Rebasing in Code Review Considered Harmful: A Large-scale Empirical Investigation

Matheus Paixao and Paulo Henrique Maia
Distributed Software Engineering Research Group
State University of Ceara
Fortaleza, Brazil
{matheus.paixao,pauloh.maia}@uece.br

***Abstract*—Code review has been widely acknowledged as a key quality assurance process in both open-source and industrial software development. Due to the asynchronicity of the code review process, the system's codebase tends to incorporate external commits while a source code change is reviewed, which cause the need for rebasing operations. External commits have the potential to modify files currently under review, which causes re-work for developers and fatigue for reviewers. Since source code changes observed during code review may be due to external commits, rebasing operations may pose a severe threat to empirical studies that employ code review data. Yet, to the best of our knowledge, there is no empirical study that characterises and investigates rebasing in real-world software systems. Hence, this paper reports an empirical investigation aimed at understanding the frequency in which rebasing operations occur and their side-effects in the reviewing process. To achieve so, we perform an in-depth large-scale empirical investigation of the code review data of 11 software systems, 28,808 code reviews and 99,121 revisions. Our observations indicate that developers need to perform rebasing operations in an average of 75.35% of code reviews. In addition, our data suggests that an average of 34.21% of rebasing operations tend to tamper with the reviewing process. Finally, we propose a methodology to handle rebasing in empirical studies that employ code review data. We show how an empirical study that does not account for rebasing operations may report skewed, biased and inaccurate observations.**

***Index Terms*—Rebasing, Code Review, Empirical Software Engineering**

I. Introduction

In software development, code review is a process in which source code changes proposed by developers are peer-reviewed by other developers before being incorporated into the system [1]. The code review process has been empirically observed to successfully assist developers in finding defects [2], [3], transferring knowledge [1], [4] and improving the overall quality of a software system. Given its benefits, code review has been widely adopted by both industrial and open-source software development communities. For example, large organisations such as Google, Microsoft and Facebook use code review systems on a daily basis [5], [6], [7], [8].

Nevertheless, code review is costly. Developers need to constantly context switch between implementation and reviewing tasks, which has been observed to be one of the major challenges in peer code review [9]. Moreover, depending on the process followed by each organisation and its underlying structure, senior developers may spend more time reviewing code than writing it [10], [11]. In addition, developers may experience confusion throughout various stages of the code review process, which have the potential to incur delays, incorrect solutions and increased development effort [12], [13]. In this context, researchers have been proposing tools and methods to improve the code review process and optimise developers' time during review. Examples include review automation [14], reviewer recommendation [15], [16], integration of static analysers [17], [18] and testing tools [19].

Moreover, we observe a myriad of empirical studies that attempt to measure and analyse the effect of code review on many aspects of software quality, such as defects [20], coding conventions [21], design [22], build analysis [23], etc. In addition, researchers have leveraged code review data to empirically study other aspects of software engineering, such as architectural changes [24], [25].

Modern code review is asynchronous [1], where developers can submit their code changes for review and immediately start working on a different task while the review is carried out. Similarly, reviewers may receive a review notification and choose to accommodate it in the best time during their daily routine. However, this asynchronicity has side-effects.

In the scenario where the system's codebase changes in the repository during the course of a review, developers need to update their local copy to carry on with the review. Modern code review systems, such as Gerrit, opt to perform a *rebasing* operation instead of *merging* when updating a change locally [26]. This is due to the benefits commonly presented by rebasing, such as cleaner version history and simplified branches [27]. However, when applied during code review, rebasing may present side-effects (see Section III). Depending on how the codebase changes during review, a rebasing operation may invalidate the code developers and reviewers have been working on, which might cause re-work, reviewing fatigue [28], and confusion [12], [13] for developers.

Rebasing operations may alter the system's codebase during code review. Hence, the source code changes observed during the review may not be due to the review itself but rather to external changes. As a researcher, this may represent a severe threat to the validity of an empirical study. Previous investigations have mentioned the problems presented by rebasing in empirical studies [29], [30], [31], yet no in-depth analyses have been performed. In spite of the potential issues rebasing

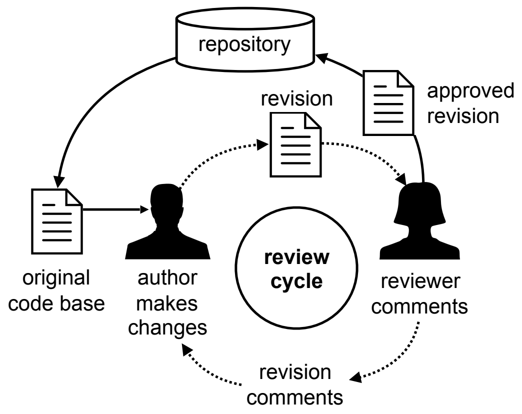


Fig. 1. The modern code review process.

may cause for both software engineering practitioners and researchers, to the best of our knowledge, there is no study to date that is dedicated to characterise and investigate rebasing and its side-effects in the reviewing process.

Thus, we list the main contributions of this paper as follows:

1) The first in-depth empirical study on the effects of rebasing in the code review process of real-world systems.
2) The identification that rebasing is performed in an average of 75.35% of code reviews, where a median of 13 files and 385 lines of code are modified due to rebasing.
3) The observation that an average of 34.21% of rebasing operations tend to cause re-work for developers and severely affect empirical studies that employ code review data.
4) A methodology to handle rebasing operations in empirical studies that employ code review data.

The rest of this paper is organised as follows. Section II presents the background for the code review process while Section III discusses the side-effects caused by rebasing operations. Section IV presents our 4 research questions and the methodology we employ to answer each of them. Section V details the results for our research questions and Section VI discusses the implications of our observations for both software engineering practitioners and researchers. Section VII presents and evaluates a methodology to handle rebasing in empirical studies that employ code review data. Finally, Sections VIII, IX and X discusses threats to the validity, related work and conclusions, respectively.

II. Code Review Background and Terminology

The modern code review process is asynchronous and commonly built on top of a decentralised version control system, such as git [1]. Moreover, a dedicated tool is employed to assist developers when visualising a review, providing feedback and committing the changes. The standard modern code review process is represented in Figure 1.

A developer starts a review by modifying the original codebase in the repository and submitting a new revision in the form of a commit. Other developers of the system will serve as reviewers by inspecting the submitted source code and providing feedback in the form of comments. Next, the review's author will modify the current revision according to the received feedback and submit the improved code as a new revision. This procedure is repeated until the last revision is either merged or abandoned, where the first indicates the code change was incorporated into the system and the latter indicates the code change was rejected.

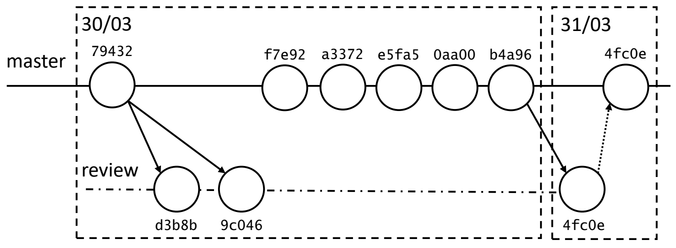


Fig. 2. Lifecycle of code review 17890 from egit.

In this paper, we use *code review* to indicate a set of sequential source code submissions that were manually inspected by developers and later merged or abandoned. In this context, a revision represents the source code submission at each different iteration of the review process. Finally, the developers involved in writing the code for review are considered to be the review's authors, and the developers responsible for inspecting the code are considered to be the reviewers.

III. The Side-effects of Rebasing in Code Review

To demonstrate the scenario in which a rebasing operation is needed alongside its side-effects in the reviewing process for both practitioners and researchers, we make use of a real code review (number 17890[1]) from the egit software system (see Section IV-A). Egit employs Gerrit as its code review tool, and the lifecycle for review 17890 is depicted in Figure 2.

The figure depicts egit's master branch and the developer's local copy, which is indicated by the 'review' timeline. As one can see in Figure 2, at the time the review started, the system's codebase in the master branch was represented by commit 79432. Hence, this is the commit whose review's first revision was based upon, i.e., the first revision's parent. Right after the first revision (commit d3b8b) was submitted, a reviewer quickly observed that the proposed source code change was prone to a *NullPointerException*. Next, the review's author checked whether the codebase had changed in the master branch. Since this was not the case, the developer simply fixed the issue in the local copy and submitted the second revision, as represented by commit 9c046. In Gerrit, new revisions are always created by *amending* the previous revision [26]. Thus, although revision 9c046 is a follow up of revision d3b8b, they are not connected in the version control system. In regards to the system's version control repository, both first and second revisions share the same parent commit, i.e., 79432.

According to Figure 2, the system's master branch remained the same while the first and second revisions occurred. Hence, the review's author did not have to update the local codebase to apply the changes suggested by the reviewers. In this case,

[1]https://git.eclipse.org/r/#/c/17890/

a rebasing operation was *not* necessary, and any difference in the source code between the first and second revisions are due to changes performed by the review's author.

Differently, between the second and third revisions, 5 commits were integrated into the master branch, each of which contains the results of other code reviews that were running in parallel to review 17890 in egit. These commits are not related to the code review at hand, yet they influence review 17890 because they modified the master's branch during the reviewing process. These are considered *external* commits to code review 17890. In this case, the review's author needed to *rebase* the master branch into the local copy to obtain the up to date version of the code from which the third revision could be based upon, as enforced by Gerrit's workflow. Hence, despite being sequentially related in the code review, the second and third revisions were based on different versions of the codebase and do not share the same parent commit.

For this particular code review, in the first two revisions, the author made modifications to parts of a large class named CheckoutDialog. However, in between the second and third revisions, the 5 external commits that have been integrated into the codebase moved large pieces of code from CheckoutDialog to the BranchSelectionAndEditDialog class. Hence, when the rebasing was performed, all the work performed in the first and second revision was invalidated and lost.

This rebasing operation caused re-work for all developers involved in the review. First, the review's author had to locate the pieces of code in the new class to adapt the original source code changes to the external changes in the codebase that were incorporated by the external commits. Similarly, the reviewers had to review virtually the same code change as in the second revision, but now into a different context.

As a researcher, rebasing may pose a serious threat to empirical studies that employ code review data. In the case of naively studying code review 17890 as a simple sequence of its revisions (d3b8b, 9c046 and 4fc0e), one would observe source code changes that are not only due to the revisions themselves, but actually results of the 5 external commits integrated into the codebase during the reviewing process.

IV. Experimental Methodology

Our goal is to study rebasing and its side-effects in the code review process. Thus, we ask the following research questions:

RQ1: *How often do developers need to perform rebasing during code review?* **(Sanity check)** As discussed in Section III, rebasing may occur during the code review process. This research question studies code review data from real-world software systems to investigate how often rebasing occurs in practice. We first need to establish the frequency in which rebasing occurs to justify the need for this empirical study.

RQ2: *For how long is the codebase exposed during the reviewing process?* **(Codebase exposure)** The results obtained in this research question indicate the time frame in which the codebase is exposed to external commits, i.e., code changes that will cause the need for rebasing.

RQ3: *How does the codebase change between rebasing?* **(Codebase change)** For the code reviews in which developers performed a rebasing operation, we measure the codebase change between the beginning of the code review to the time the rebasing was performed. This indicates the changes in the source code that are not due to the code review itself, but actually to external commits.

RQ4: *How often does rebasing tamper with the code review process?* **(Side-effects of rebasing)** A certain code review is tampered by rebasing operations when the code changes performed in the external commits invalidate the code changes currently at review, as presented in Section III. The observations drawn from this research question will shed light on the side-effects caused by rebasing for both software engineering practitioners and researchers.

Next, we present the dataset used in this study followed by the experimental methodology employed to answer each of the research questions above. In addition, we provide a complete replication package [32] for our study, including the raw data for all steps of our experimental methodology and analyses.

A. Code Review Data in CROP

In this paper, we make use of CROP, a curated open-source dataset of code review data. We have developed CROP to support our previous work in architectural changes during code review [24], [25], and it comprises code review data from 11 software systems developed by two well-known open-source communities: Eclipse [33] and Couchbase [34]. At the time we developed CROP, we selected the software systems with most code review data from each of these two communities. Given a certain software system, CROP provides its complete reviewing history. For each code review, we have access to the review's data and metadata, such as the description and comments from developers alongside information about the review's author and timestamps, for example. In addition, CROP provides a complete copy of the entire codebase for each revision and its respective parent, which represent the system's codebase at the time of review. For the interested reader, we refer to CROP's original paper [35] and website [36].

For this empirical study, we employ the code review data of all systems contained in CROP. By including in our study all available systems from a dataset that was not originally designed for this paper, we avoid 'cherry-picking' and selection biases in our experiment. Despite considering all software systems in CROP, not all code reviews from each system suit our study. For instance, abandoned reviews represent changes that were not integrated into the system. Therefore, these reviews do not present any influence on the system's codebase regardless of rebasing. In addition, code reviews that are composed of a single revision are not affected by rebasing. Hence, the inclusion criteria for this empirical study consisted of merged code reviews composed of at least two revisions.

Table I presents details about each system and the set of code reviews we consider in this study. In total, we study 11 software systems, 28,808 code reviews and 99,121 revisions. The reviewing time span for the systems under analysis range from 3 to 8 years. Finally, we study systems that provide a

TABLE I
SOFTWARE SYSTEMS CONSIDERED IN THIS PAPER. WE PRESENT THE SYSTEMS' IDS USED IN THIS STUDY, DESCRIPTION, CORE PROGRAMMING LANGUAGE, TIME SPAN OF REVIEWING HISTORY, NUMBER OF REVIEWS AND REVISIONS CONSIDERED, AND AVERAGE NUMBER OF COMMITS PER DAY.

Systems	ID	Description	Programming Language	Time Span	Number of Reviews	Number of Revisions	Commits per Day
linuxtools	S1	C/C++ IDE for linux developers	Java	06/12 to 11/17	3,438	10,635	2.85
platform.ui	S2	Building blocks for user interfaces in Eclipse	Java	02/13 to 11/17	2,985	11,188	4.47
egit	S3	Integration of jgit into the Eclipse IDE	Java	09/09 to 11/17	2,899	9,827	1.70
jgit	S4	Java implementation of Git	Java	10/09 to 11/17	2,533	9,961	1.97
testrunner	S5	Tool used to run Couchbase's tests	Python	11/10 to 11/17	7,050	20,138	3.98
ns_server	S6	Couchbase's server supervisor	JavaScript	05/10 to 11/17	6,218	25,246	4.18
ep-engine	S7	Couchbase's eventual persistency implementation	C++	02/11 to 11/17	3,990	18,415	2.44
indexing	S8	Couchbase's indexes implementation	Go	03/14 to 11/17	1,964	6,695	1.75
java-client	S9	Couchbase's driver implementation in Java	Java	11/11 to 11/17	766	2,362	0.37
jvm-core	S10	Low-level API mostly used by java-client	Java	04/14 to 11/17	698	2,097	0.47
spymemcached	S11	Implementation of a memory caching system	Java	05/10 to 07/17	257	972	0.25

varied set of functionalities alongside implementations in up to 5 different programming languages.

B. *Identifying Rebasing Operations*

In RQ1, we investigate the occurrence of rebasing during code review. To do so, we rely on the revisions' commit ids and their respective parents to identify the rebasing operations. Consider the example provided in Figure 2. The first revision is represented by commit d3b8b, whose parent is commit 79432. The master branch has not incorporated any external commit between the first and second revisions. Hence, the second revision (9c046) presents the same parent as the first revision. In this case, since the parent commit for both revisions is the same, a rebase operation *has not* been performed.

Between the second and third revisions, 5 external commits have been integrated into the master branch. As a result of the codebase change, the third revision, commit 4fc0e, presents a parent commit (b4a96) that is different from the two previous revisions. The change in the parent commit between revisions indicates that a rebasing operation *has occurred*. By following this procedure for all code reviews in CROP, we can identify all instances of rebasing operations in our dataset.

C. *Measuring Codebase Exposure During Code Review*

In RQ2, we investigate codebase exposure during the reviewing process. To achieve this, we observe how long it commonly takes from the beginning of a code review to its latest rebasing operation. For this study, we employ number of days as a proxy to measure the time of codebase exposure during code review. Codebase exposure can be computed for each revision within a code review. Since we aim at computing the complete time frame in which the codebase is exposed during code review, we chose to consider the latest revision to compute codebase exposure.

Consider the example presented in Figure 2. To measure the codebase exposure during this review, we first identify the parent commit of its first revision, i.e., 79432. Through *git log*, we notice that this commit was integrated into the codebase on the 30/03. Next, we identify the parent commit of the revision in which the latest rebasing operation has occurred. In this case, developers only performed a rebasing operation on the third revision, whose parent commit is b4a96. Since this commit was also integrated into the codebase on the 30/03, we observe that the codebase was exposed for 1 day.

D. *Measuring Codebase Change During Code Review*

In RQ3, we investigate how the codebase evolves during the course of a code review. The codebase change represents the source code that was integrated into the codebase as a result of all external commits that occurred during the reviewing process. To achieve this, we employ a mix of repository activity and churn metrics.

First, we identify the parent commit of the first revision and latest rebasing operation. For the example in Figure 2, we identify 79432 and b4a96, respectively. Next, we measure the repository activity between these two commits by observing the number of commits integrated into the master branch between them, which in this case, equals 5. In addition, we measure the source code churn through the following metrics: number of files changed, number of hunks, and number of lines changed. Note that all these churn metrics have been reported by developers as highly influential in a review's response time and reviewer fatigue [9]. For this particular review, we observe that the 5 external commits changed 17 files in the codebase by performing 46 hunks and changing a total of 818 lines. These metrics indicate not only the number of commits that caused the rebasing but also the amount of change in the codebase that is not due to the code review itself.

E. *Identifying Code Review Tampering due to Rebasing*

Every rebasing operation has the potential to tamper with a code review. As presented in Section III, external commits may invalidate the changes proposed by a review. Hence, in RQ4, we investigate how often external commits tamper with a code review when rebasing is performed.

To properly measure code review tampering, one would need to employ robust dynamic and static program analysis techniques. However, such techniques would incur drawbacks in this study. First, we cannot guarantee that all commits of all systems are compilable and have a test suite from which we can perform dynamic analysis. Thus, any compilation and/or testing issue that we observe during code review cannot be directly linked to either the external commits or the code review. In addition, we consider 11 software systems implemented in 5 different programming languages in this study. As a result, we could not find equivalent static analysers for all languages which would yield comparable results. Choosing a specific language and/or a set of systems to carry on this investigation would result in considerable threats to the study's generalisation and overall validity.

Thus, we employ version control analysis to identify code review tampering. We consider code review tampering to occur when external commits modify files that are currently involved in a code review, as described as follows. For each code review, we record the files involved in each of its revisions, as indicated by the CROP dataset. In the example presented in Section III, we observe that, during the course of its three revisions, CheckoutDialog.java and BranchSelectionAndEditDialog.java were the files modified by the review's author. Next, we employ *git diff* to identify the list of files modified in the codebase between the review's first revision and latest rebasing operation, i.e., commits 79432 and b4a96 in our example. In the case we observe at least one file being modified by both sets of revisions and external commits, we consider the respective code review to be tampered with. In our example, commit e5fa5, which was integrated into the code between the review's second and third revisions, moved code from CheckoutDialog.java to BranchSelectionAndEditDialog.java. Therefore, we consider this review to be tampered by the rebasing operation.

Hence, by following this language agnostic and scalable procedure, we can investigate code review tampering for all systems in our dataset. Although we are aware that code review tampering may occur through means other than modifying the same file, this procedure provides a safe under-approximation of code review tampering.

V. Experimental Results

A. RQ1: How often do developers need to perform rebasing during code review?

For this research question, we consider all reviews composed of more than one revision, as presented in Table I. For each system, we computed the number of reviews in which we identified at least one rebasing operation, following the procedure described in Section IV-B. The first two columns of Table II present (i) the number of reviews with a rebasing operation and (ii) the percentage of rebasing occurrence regarding all reviews considered in the study. Considering linuxtools, for example, we identified 2,376 code reviews in which a rebasing operation was performed, which accounts for 69.11% of all reviews composed of more than one revision in linuxtools.

When considering all systems, the average percentage of rebasing occurrence is 75.35%. This indicates that, on average, a rebasing operation is likely to occur in 3 out of 4 code reviews. For platform.ui and indexing, developers performed rebasing in 93.74% and 91.85% of reviews, respectively.

In addition, we computed the rebasing ratio for each review in each system. This metric indicates how often developers need to perform rebasing in a single code review. Consider

TABLE II
Rebasing during code review. For each system, we report the number of reviews with rebasing operations, the percentage of rebasing occurrence, and the average rebasing ratio.

Systems	Reviews With Rebasing	Rebasing Occurrence	Average Rebasing Ratio
linuxtools	2,376	69.11%	79.09%
platform.ui	2,798	93.74%	81.87%
egit	2,464	84.99%	79.04%
jgit	1,941	76.63%	79.43%
testrunner	4,918	69.76%	77.29%
ns_server	5,397	86.80%	89.61%
ep-engine	2,976	74.56%	70.52%
indexing	1,804	91.85%	86.88%
java-client	412	53.79%	68.09%
jvm-core	397	56.88%	72.33%
spymemcached	182	70.82%	70.43%

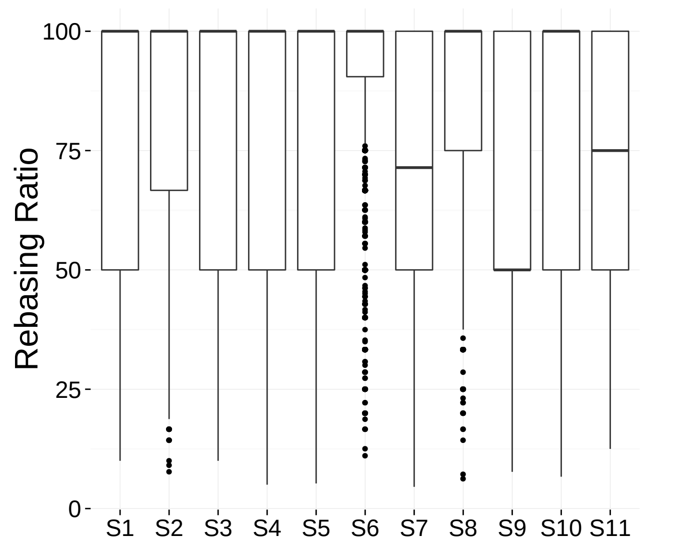


Fig. 3. Distribution of rebasing ratio for each system under study. The rebasing ratio indicates the percentage of revisions within a code review where developers performed a rebasing operation.

the example depicted in Figure 2, which is composed of 3 revisions. Every revision after the first one is prone to a rebasing operation due to changes in the codebase, which, in this case, accounts for 2 revisions. For this particular review, we only identified 1 rebasing operation. Hence, the rebasing ratio of this code review is $\frac{1}{2} \times 100\% = 50\%$.

According to our analysis, the overall rebasing ratio for all systems is 77.72%. This indicates that, on average, developers need to perform rebasing in about 75% of all revisions they work on. Figure 3 depicts boxplots with the complete distribution of the rebasing ratio for each system, as represented by their respective id (see Table I).

As one can see from Figure 3, the boxplots are highly skewed towards upper values of rebasing ratio. For instance, 8 out of the 11 systems under study present a median rebasing ratio of 100%. This indicates that, in most systems, in more than half of the code reviews, developers had to perform a rebasing operation for every single revision as the review progressed. When considering the distribution for ns_server,

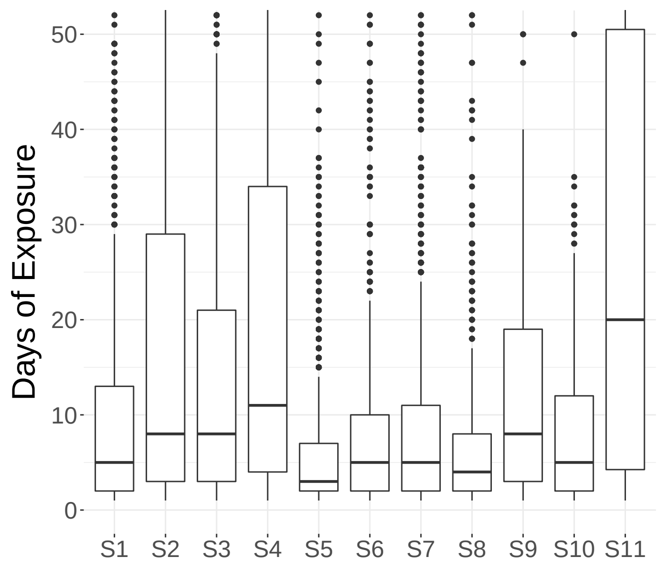


Fig. 4. Codebase exposure during code review. The boxplots indicate the number of days the codebase remained exposed during code review.

for example, all code reviews with a rebasing ratio lower than 75% are considered outliers in the distribution.

As an answer to RQ1, our analysis indicate that developers need to perform rebasing in an average of 75% of all code reviews. Moreover, developers need to perform a rebasing operation in about 75% of all revisions within a single code review. Hence, we observe that developers commonly perform rebasing as part of their code review process.

B. RQ2: For how long is the codebase exposed during the reviewing process?

According to a recent empirical study [15], code reviews commonly take from 2 to 5 days for completion. During this period, the system's codebase is exposed to changes other than the ones being reviewed, which might eventually cause developers to perform a rebasing operation. This research question aims at investigating the exposure time of the systems' codebase during the course of a code review. To do so, we measure the exposure time in number of days, as described in Section IV-C and displayed in Figure 4. Each system is represented by their respective id, as described in Table I. Notice that the boxplots are zoomed to display a maximum of 50 days of exposure for better visualisation. Nevertheless, our replication package [32] provides the complete set of results.

Consider linuxtool (S1), for example. A median value of 5 days of exposure indicates that the codebase has been exposed for more than 5 days during the course of more than half of its code reviews. For all systems under study, the median value of codebase exposure lies between 3 and 20 days. The systems with highest median values of codebase exposure are spymemcached and jgit with 20 and 11 days, respectively. The systems with smallest median values of codebase exposure are textrunner and indexing with 3 and 4 days, respectively.

Although not displayed in Figure 4, we observed a heavy-tailed distribution of codebase exposure for all systems. For instance, reviews number 1614 and 103921 from jgit, present codebase exposure of 1,574 and 1,506 days, respectively.

Besides separately analysing each system, we additionally combined the codebase exposure computed for all reviews of all systems into a single distribution. Hence, as an answer to RQ2, when considering all reviews in our dataset, the median value of codebase exposure is 6 days. Since all of the systems under study present an average of 2.22 commits per day (see Table I), one can see that the systems' codebases are significantly exposed to external commits during code review.

C. RQ3: How does the codebase change between rebasing?

To investigate codebase change during the reviewing process, we measure the number of commits, number of files changed, number of hunks and number of lines changed for each review and each system, as described in Section IV-D. Figures 5(a) and 5(b) display the codebase change during code review in regards to number of commits and files changed, respectively. Similarly to RQ2, all boxplots are zoomed for better visualisation. Due to space constraints, we omit the boxplots regarding number of hunks and lines changed from the paper. Nevertheless, for the full results, the interested reader can access our replication package [32].

The median number of commits between rebasing varies from 2 to 11 for the software systems under study. Nevertheless, we have commonly observed cases in which the codebase has integrated more than 20 external commits during the course of a code review. When considering all reviews from all systems, the median number of external commits integrated into the codebase during code review is 5.

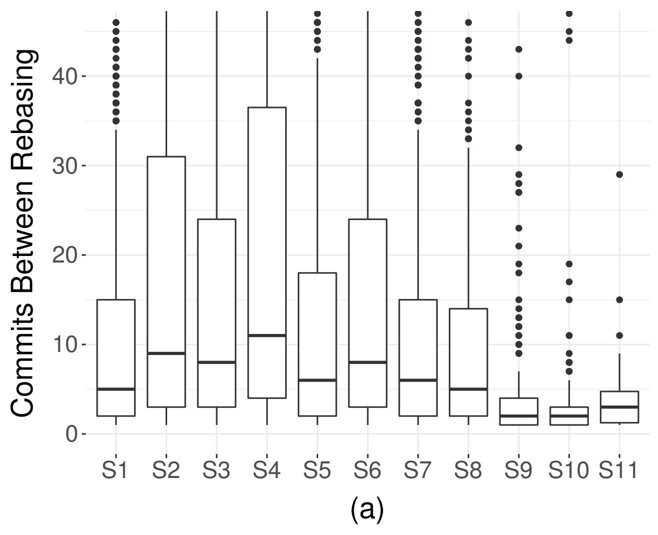
As one can see from Figure 5(b), the number of files changed in the codebase during review is commonly higher than 10. When considering linuxtools (S1) and platform.ui (S2), the median number of files changed is 22 and 36, respectively. In some cases, the distribution of files changed outside review reaches from 100 to 150 files. For all reviews and all systems, 13 files represent the median number of files changed by external commits during a code review.

For number of hunks and number of lines changed, we continue to observe linuxtools and platform.ui being the systems with the largest codebase change during review. When considering all reviews in our dataset, the median number of hunks and lines changed in the codebase during the reviewing process is 32 and 385, respectively.

As an answer to RQ3, the codebase of the systems under study integrate a median of 5 external commits, which cause a median change of 13 files during the course of a code review. Each of these external commits and changes will cause the need for a rebasing operation with the potential to tamper with the reviewing process.

D. RQ4: How often does rebasing tamper with the code review process?

Code review tampering occurs when external commits incorporated into the codebase during the course of a review invalidate the changes proposed in the code review. In this



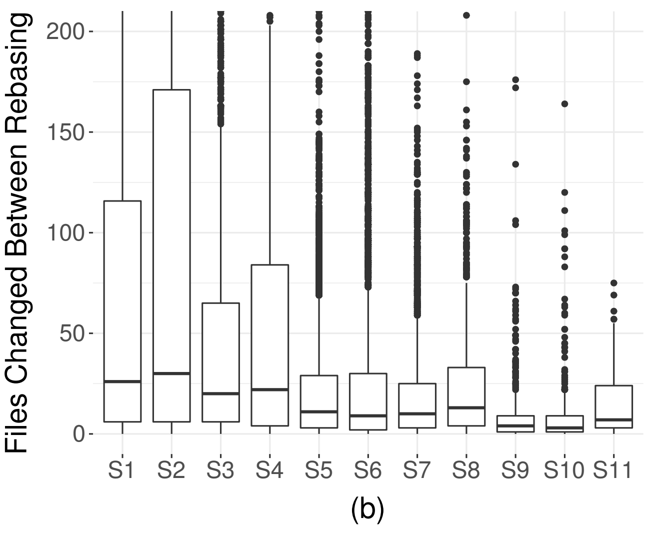


Fig. 5. Codebase change during code review. Figure (a) indicates the number of external commits integrated into the codebase during the course of a review. Figure (b) indicates the number of files changed as a result of the external commits.

TABLE III
CODE REVIEW TAMPERING. FOR EACH SYSTEM, WE REPORT THE NUMBER OF REVIEWS TAMPERED BY REBASING, THE PERCENTAGE OF TAMPERING OCCURRENCE, AND THE AVERAGE TAMPERING RATIO.

Systems	Reviews Tampered by Rebasing	Tampering Occurrence	Average Tampering Ratio
linuxtools	533	22.43%	43.86%
platform.ui	593	21.19%	60.62%
egit	1,060	43.01%	57.75%
jgit	845	43.53%	58.41%
testrunner	1,083	22.02%	70.14%
ns_server	1,525	28.25%	67.25%
ep-engine	1,617	54.35%	64.84%
indexing	539	29.87%	63.13%
java-client	93	22.57%	47.96%
jvm-core	62	15.61%	44.00%
spymemcached	93	51.09%	57.25%

paper, we identify code review tampering by observing when external commits modify the same files involved in the code review, as described in Section IV-E. The first two columns of Table III present (i) the number of reviews tampered by rebasing and (ii) the percentage of review tampering regarding all reviews in each system. Consider testrunner, for example, we identified 1,083 code reviews tampered by rebasing, which accounts for 22.02% of all reviews in which a rebasing operation has occurred.

The systems which we observed the most and least amount of code review tampering are ep-engine and jvm-core, with 54.35% and 15.61% of reviews being tampered due to rebasing, respectively. When considering all systems, the average tampering occurrence is 34.21%. This indicates that 35.21% of all code reviews in our dataset were affected by external commits being incorporated into the codebase during review.

In addition, we computed the tampering ratio for each review. This metric computed the percentage of files involved in a certain review that were modified by external commits. Consider the example in Figure 2. All revisions within code review 17890 modified two files: CheckoutDialog.java and CheckoutDialog.java. We observed that the code changes caused by the 5 external commits incorporated during the review modified both files involved in the review. Hence, the tampering ratio for this code review is $\frac{2}{2} \times 100\% = 100\%$. The third column of Table III depicts the average tampering ratio for each system considered in this study. In platform.ui, for instance, when considering the reviews in which tampering occurred, an average of 60.62% of the files involved in the review were modified by external commits.

As one can see from Table III, the values for tampering ratio tend to be high. On average, when considering all systems, 57.74% of the files involved in tampered reviews were affected by external changes. This indicates that, when a review is tampered by rebasing, about half of the files involved in the review tend to be altered by commits outside the reviewing process. We observed considerably high levels of tampering ratio for a few systems in our dataset, such as ns_server and testrunner, which presented an average tampering ratio of 67.25% and 70.14%, respectively.

Furthermore, we present the distribution of tampering ratio for each system in Figure 6. Overall, the boxplots are skewed towards upper values of tampering ratio, indicating that most files involved in a review tend to be modified by external changes when tampering occurs. Moreover, for some systems, such as testrunner and ns_server, more than half of tampered reviews have more than half of its files modified by external changes.

As an answer to RQ4, 34.21% of rebasing operations tend to tamper with the reviewing process. Moreover, according to our analyses, rebasing operations that tamper with a code review tend to do so by modifying an average of 60.62% of files involved in the review. One should notice that each of these instances of code review tampering have the potential to cause re-work and generally disrupt the reviewing process, as described in Section III.

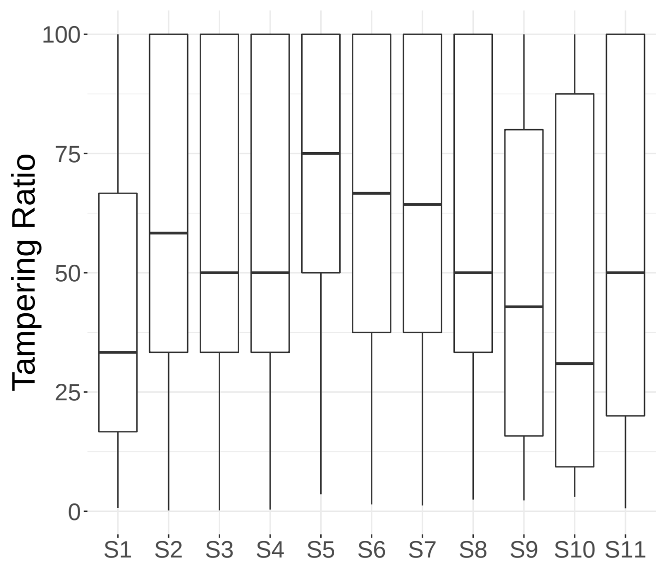


Fig. 6. Code review tampering due to rebasing operations

VI. Discussion

A. *Contributions for Software Engineering Practitioners*

Software developers who submit code for review and serve as a reviewer of source code changes from their peers on a daily basis need to understand the side-effects that rebasing may have on their workflow. As an author of a code review who will need to perform rebasing in between revisions, one must be aware that external changes might invalidate the existing code written for review. In this scenario, one should first understand the differences in the codebase to identify any possible conflict with the existing code under review. To avoid re-work and possible defects in the case of code tampering by rebasing, the review's author should plan the changes that will be necessary to adapt the revision to the new codebase.

When reviewing a revision which performed a rebasing operation, reviewers may waste time by looking at changes that actually came from external commits and not from the code review itself, which may cause reviewing fatigue [28] and lead to an overall poor (and sometimes unnecessary) reviewing process. In this case, reviewers need to be able to differentiate between the code being submitted for review and the code that is a result of external commits.

Tools and automation are a key factor to assist in the tasks listed above. Code review systems, such as Gerrit, should be able to automatically identify code review tampering to better advise developers when rebasing is necessary. In addition, visualisation techniques can be employed to assist reviewers in differentiating new code from rebased code during review.

A recent empirical study has provided a comprehensive framework of confusion during code review [13]. The authors observed complex changes, lack of context, change impact and version control issues as causes for the confusion. We have noticed many of the causes for confusion mentioned in the related paper may be due to rebasing operations being performed during review. Nevertheless, a follow-up study is necessary to fully identify to what extent rebasing operations are linked to developers confusion in code review.

B. *Contributions for Software Engineering Researchers*

Code review data has been employed in a plethora of empirical studies that not only evaluate the code review process itself [1], [2], [3], [4], [5], [6], [7], [8], [9], [10], [11], [12], [13], [14], [15], [16], [20], [21], [22], [23] but also use code review data to study other software engineering practices [24], [25]. In this context, our analyses indicate that the understanding of rebasing and its effects in the code review process is paramount for good quality empirical studies involving code review data. Next, we list the main take away points for researchers when performing empirical studies that employ code review data.

1) Researchers cannot ignore code reviews in which rebasing operations have been performed.

To avoid the issues and difficulties caused by rebasing in empirical studies that involve code review data, one could filter out all reviews and revisions with rebasing. However, our investigation suggests that rebasing operations are performed in an average of 75.35% of all code reviews. Moreover, rebasing tend to be performed for an average of 77.72% of all revisions within a code review. Hence, to filter out reviews and revisions with rebasing, one would incur serious threats to the study's validity since most of the dataset would not be studied.

2) External commits happen often, and their side-effects in the codebase may bias a study's methodology and observations.

Our analyses suggest that a median of 5 external commits are incorporated into a software system's codebase during the course of a review. These commits modify a median of 13 files and change a median of 385 lines of code. Hence, if rebasing operations are not identified and handled in an empirical study, researchers may incur noise in their code review datasets. As a result, observations drawn from studies that employ code review data, yet do not consider rebasing, are prone to biases and inaccuracies.

3) External commits tend to modify and tamper with files involved in a code review.

Our observations suggest that an average of 34.21% of code reviews are tampered by rebasing, i.e., external commits incorporated during the reviewing process modified files involved in a code review. This indicates that external commits not only affect the codebase as a whole but also the files involved in the code review itself.

Based on the points and analyses presented above, we advocate that all empirical studies that involve code review data must handle rebasing operations as part of their methodology's design. This will ensure that we, as a community, will be able to perform better empirical studies with the ability to achieve scientifically sound and unbiased observations.

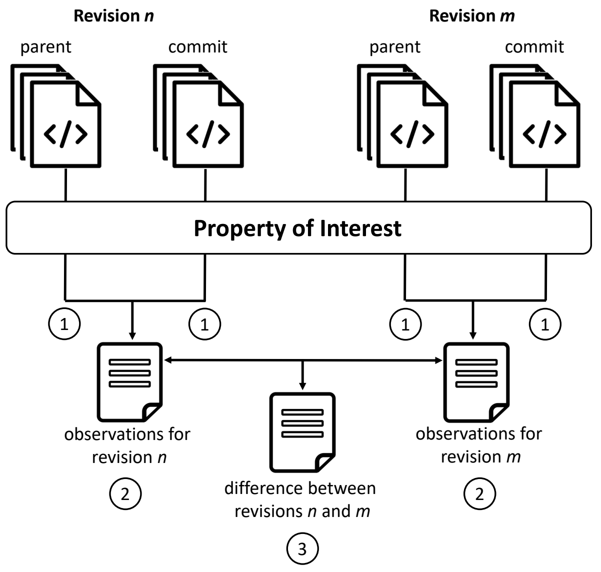

Revision n
Revision m
parent
commit
parent
commit
Property of Interest
1
1
1
1
observations for revision n
2
difference between revisions n and m
3
observations for revision m
2

Fig. 7. Methodology to handle rebasing operations in empirical studies that employ code review data.

VII. Handling Rebasing in Code Review Data

In this section, we propose a methodology to handle rebasing operations in empirical studies that employ code review data. Next, we compare our methodology to a naive one, where rebasing operations are not properly handled. The results aim at showcasing the threats and biases researchers may incur to their empirical studies if rebasing is not considered.

Thus, Figure 7 presents our proposed methodology to handle rebasing in empirical studies that employ code review data. The numbers in the figure indicate the order of steps necessary to replicate the approach. Consider an empirical study in which one needs to measure the difference between two revisions (n and m) according to a certain property of interest, such as code readability, testing coverage, code smells, architectural quality, build time, energy consumption, etc.

The first step consists in separately computing the property of interest for each revision and its respective parent. Due to external commits that may have altered the system's codebase between revisions n and m, one cannot compare their code directly, otherwise, external changes will be observed in the revisions' codebases. Hence, in the second step, one must compute the difference in the property of interest between each revision and its parent to extract the correct observations for a revision. In the third step, one can compute the difference in observations between n and m, which represent the change in the codebase between revisions according to the property of interest employed in the empirical study.

To demonstrate the usage of the proposed methodology, we devised a simple empirical study regarding software refactoring. Given a certain code review, our goal is to measure the number of refactoring operations performed during the course of the review. In this context, we use RefMiner [37], a tool that identifies refactoring operations between two commits of a software system implemented in Java. Hence, we use RefMiner to identify the refactoring operations performed between the first and last revision of each code review. For this study, we excluded (i) all the systems not implemented in Java and (ii) platform.ui due to constant crashes in RefMiner while executed in this system. We compare the results achieved by our methodology to the results achieved by a naive one, where rebasing operations are not properly handled.

TABLE IV
For each system, we present the number of refactorings identified by the naive and the proposed methodologies. In addition, we report the reviews' error ratio between the two.

Systems	Refactorings in Naive Methodology	Refactorings in Proposed Methodology	Reviews' Error Ratio
linuxtools	45,710	3,665	39.00%
egit	28,420	1,239	52.56%
jgit	38,436	1,506	47.49%
java-client	2,291	303	20.62%
jvm-core	2,034	216	15.47%
spymemcached	1,017	121	26.84%

In the naive methodology, we compare the codebase of the first and last revisions of each code review directly. Consider the example depicted in Figure 2. We apply RefMiner to commits 4fc0e and d3b8b, and record the number of refactoring operations reported. For this code review, we observed a total of 16 refactoring operations being performed between the codebase of the first and last revisions. However, we cannot differentiate which ones are due to the code review itself and which ones are due to external commits.

Hence, in Step 1, we compute the refactoring operations for the first revision and its parent (d3b8b and 79432) followed by the refactoring operations for the last revision and its parent (4fc0e and b4a96). As a result, in Step 2, we identified that 2 refactoring operations were applied in both the first and second revisions. In Step 3, we compared the refactoring operations identified in the two revisions. After analysis, we noticed that the refactoring operations reported in both revisions are the same. Hence, no refactoring operation has been added and/or removed during the course of code review 17890. This indicates that the true number of refactoring operations performed by this code review is 2.

After a qualitative analysis of these results, we confirmed that the other 14 refactoring operations reported by the naive methodology were due to the 5 external commits incorporated into the system's codebase between the first and last revision. Hence, an empirical study that does not handle rebasing would have observed that 14 refactoring operations were performed in code review 17890, where, in reality, only 2 actually were.

Table IV presents the results for all code reviews and all systems under study. Consider egit, for example. The naive methodology identified 28,420 refactoring operations, where our proposed methodology identified 1,239 refactoring operations. This represents a total of 27,181 wrongfully identified refactoring operations in egit's code reviews when rebasing is not handled in the empirical study's design.

In addition, we report the reviews' error ratio for the naive methodology. This metric indicates the proportion of code reviews in which the naive methodology wrongfully identified refactoring operations, in comparison to our proposed methodology. In jgit, for example, the naive methodology identified wrong refactoring operations in 47.49% of all reviews. When considering all systems under study, the average reviews' error ratio is 33.66%. This indicates that, for this particular empirical study, not handling the rebasing operations would have caused inaccurate observations in 33.66% of the dataset.

We do not believe the presented methodology will be promptly applicable for all possible empirical studies that employ code review data. Nevertheless, its basic ideas and future extensions might serve as a starting point for the development of good practices when mining code review data in the presence of rebasing.

VIII. Threats to the Validity

Construct: Our experimental methodology may be only valid for code review data extracted from Gerrit. To mitigate this threat, we designed an experimental methodology that is based upon basic constructs that can be found in the vast majority of open-source and industrial code review systems in use nowadays. Hence, this assures that our methodology is applicable beyond code review data from Gerrit.

External: Previous empirical studies have shown that different code review communities behave differently with regards to Git usage and practices [38], [39]. Hence, our results may only generalise for software systems that employ similar code review practices like the ones in the communities we studied. To alleviate this threat, we employed a curated dataset of code review data that include 11 systems developed by two different well-known open-source communities: Eclipse [33] and Couchbase [34]. In addition, we included all the 11 software systems available in the dataset to avoid 'cherry-picking' and enhance the study's generalisability.

Internal: Our method for identifying code review tampering does not cover all possible tampering scenarios. We are aware that code review tampering cannot be fully identified through version control analysis alone. However, this method enables the analysis of all 11 systems in our dataset, which would not be possible for more sophisticated dynamic and static analysers. Furthermore, our method represents a safe under-approximation of code review tampering from which we can draw reliable scientific observations. Finally, we include advanced methods for code review tampering identification in our directions for future work.

IX. Related Work

Bird et al. [29] investigate git repositories and list potential issues a researcher might face when performing empirical studies that employ data from git. In addition, the authors present guidelines on how to possibly deal with some of the main perils of git mining. Rebasing is mentioned as a threat in regard to 'squashing' several commits into a single change, yet its side-effects on tampering with other developers' work are neither mentioned or investigated.

Kalliamvakou et al. [30], [40] focus on potential threats when performing studies that employ data from the Github platform. In this case, the authors provide comprehensive strategies for a researcher to alleviate the identified perils. The latter paper provides details on how Github handles rebasing, where the discussion focuses on losing commit history. Similarly to the work by Bird et al. [29], no investigation or detailed empirical study is performed to evaluate rebasing.

German et al. [31] perform a study of how the Linux project uses git. Alongside its main results, the authors present several challenges they faced while executing the study. Rebasing is largely mentioned throughout the paper as the main cause of lost history between repositories. The authors report that rebasing can be observed in about 20% of Linux's commits. However, there is no in-depth investigation of the side-effects of rebasing in the empirical study.

X. Conclusion

Code review is a process in which source code changes proposed by developers are peer-reviewed by other developers before being incorporated into the system. Code review has been widely adopted in both industrial and open-source software development due to its positive impact on quality assurance and overall software quality. In addition, researchers tend to heavily employ code review data in empirical studies that span across multiple areas within software engineering.

Source code changes are reviewed asynchronously to enhance development and reviewing productivity. However, this feature incurs side-effects to the code review process. While a source code change is reviewed, the system's codebase may incorporate external commits, which will cause the need for rebasing. External changes that alter source code files under review have the potential not only to cause re-work and fatigue to practitioners but also to pose serious threats for researchers.

Hence, this paper reported the first in-depth large-scale empirical study of rebasing in real-world software systems. We analysed code review data from 11 systems and 28,808 code reviews. Through version control analysis and mining software repositories, we studied the frequency in which rebasing occurs and its side-effects in the reviewing process.

Our observations indicate that developers need to perform rebasing operations in an average of 75.35% of code reviews. In addition, external commits modify a median of 13 files and 385 lines of code during the course of a code review. Finally, our data suggests that an average of 34.21% of rebasing operations tend to tamper with the reviewing process by invalidating the source code changes currently under review.

These results indicate that rebasing operations may affect a developer's work routine by causing re-work of source code changes and fatiguing reviewers. Moreover, empirical studies that employ code review data may be prone to severe threats to validity if rebasing is not considered. Finally, we proposed and evaluated a methodology to handle rebasing operations in code review data to assist researchers in obtaining bias-free and scientifically sound observations.

As future work, we lay out the usage of static and dynamic analysis to identify code review tampering due to rebasing. In addition, we plan to extend our dataset and analyses to include code review data from other communities.

REFERENCES

[1] A. Bacchelli and C. Bird, "Expectations, outcomes, and challenges of modern code review," in *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, may 2013, pp. 712–721.

[2] M. Mantyla and C. Lassenius, "What types of defects are really discovered in code reviews?" *IEEE Transactions on Software Engineering*, vol. 35, no. 3, May 2009.

[3] M. Beller, A. Bacchelli, A. Zaidman, and E. Juergens, "Modern code reviews in open-source projects: which problems do they fix?" in *Working Conference on Mining Software Repositories (MSR)*, 2014.

[4] P. C. Rigby and C. Bird, "Convergent contemporary software peer review practices," in *Foundations of Software Engineering (ESEC/FSE)*, 2013.

[5] C. Sadowski, J. van Gogh, C. Jaspan, E. Soderberg, and C. Winter, "Tricorder: Building a Program Analysis Ecosystem," in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 1. IEEE, may 2015, pp. 598–608.

[6] A. Bosu, M. Greiler, and C. Bird, "Characteristics of Useful Code Reviews: An Empirical Study at Microsoft," in *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*, vol. 2015-Augus. IEEE, may 2015, pp. 146–156.

[7] D. G. Feitelson, E. Frachtenberg, and K. L. Beck, "Development and Deployment at Facebook," *IEEE Internet Computing*, vol. 17, no. 4, pp. 8–17, jul 2013.

[8] J. Shimagaki, Y. Kamei, S. McIntosh, A. E. Hassan, and N. Ubayashi, "A study of the quality-impacting practices of modern code review at Sony mobile," in *Proceedings of the 38th International Conference on Software Engineering Companion - ICSE '16*. New York, New York, USA: ACM Press, 2016, pp. 212–221.

[9] O. Kononenko, O. Baysal, and M. W. Godfrey, "Code Review Quality: How Developers See It," in *Proceedings of the 38th International Conference on Software Engineering - ICSE '16*. New York, New York, USA: ACM Press, 2016, pp. 1028–1038.

[10] G. Gousios, A. Zaidman, M.-A. Storey, and A. van Deursen, "Work Practices and Challenges in Pull-Based Development: The Integrator's Perspective," in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 1. IEEE, may 2015, pp. 358–368.

[11] T. Baum, O. Liskin, K. Niklas, and K. Schneider, "Factors influencing code review processes in industry," in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering - FSE 2016*. New York, New York, USA: ACM Press, 2016, pp. 85–96.

[12] F. Ebert, F. Castor, N. Novielli, and A. Serebrenik, "Confusion Detection in Code Reviews," in *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, sep 2017, pp. 549–553.

[13] ——, "Confusion in Code Reviews: Reasons, Impacts, and Coping Strategies," in *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2019, pp. 49–60.

[14] V. Balachandran, "Reducing human effort and improving quality in peer code reviews using automatic static analysis and reviewer recommendation," in *35th International Conference on Software Engineering (ICSE)*, may 2013.

[15] 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," in *International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, March 2015.

[16] M. B. Zanjani, H. Kagdi, and C. Bird, "Automatically recommending peer reviewers in modern code review," *IEEE Transactions on Software Engineering*, vol. 42, no. 6, Jun. 2016.

[17] A. Rice, E. Aftandilian, C. Jaspan, E. Johnston, M. Pradel, and Y. Arroyo-Paredes, "Detecting argument selection defects," *Proceedings of the ACM on Programming Languages*, vol. 1, pp. 1–22, 2017.

[18] J. Bader, S. Chandra, E. Lippert, and A. Scott. (2018) Getafix: How facebook tools learn to fix bugs automatically. Accessed: 2019-07-29. [Online]. Available: https://code.fb.com/developer-tools/getafix-how-facebook-tools-learn-to-fix-bugs-automatically/

[19] K. Mao. (2018) Sapienz: Intelligent automated software testing at scale. Accessed: 2019-07-29. [Online]. Available: https://code.fb.com/developer-tools/sapienz-intelligent-automated-software-testing-at-scale/

[20] S. McIntosh, Y. Kamei, B. Adams, and A. E. Hassan, "The impact of code review coverage and code review participation on software quality: a case study of the qt, vtk, and itk projects," in *Working Conference on Mining Software Repositories*, ser. MSR, 2014.

[21] S. Panichella, V. Arnaoudova, M. Di Penta, and G. Antoniol, "Would static analysis tools help developers with code reviews?" in *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. IEEE, mar 2015, pp. 161–170.

[22] R. Morales, S. McIntosh, and F. Khomh, "Do code review practices impact design quality? a case study of the qt, vtk, and itk projects," in *International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, 2015.

[23] R. Wen, D. Gilbert, M. G. Roche, and S. McIntosh, "BLIMP Tracer: Integrating Build Impact Analysis with Code Review," in *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, sep 2018, pp. 685–694.

[24] M. Paixao, J. Krinke, D. Han, C. Ragkhitwetsagul, and M. Harman, "Are developers aware of the architectural impact of their changes?" in *International Conference on Automated Software Engineering*, ser. ASE, 2017.

[25] ——, "The Impact of Code Review on Architectural Changes," *IEEE Transactions on Software Engineering*, 2019.

[26] Gerrit. (2019) Gerrit's documentation and user guide. Accessed: 2019-07-29. [Online]. Available: https://gerrit-review.googlesource.com/Documentation/intro-user.html

[27] S. Chacon and B. Straub, *Pro Git*. Berkeley, CA: Apress, 2014. [Online]. Available: http://link.springer.com/10.1007/978-1-4842-0076-6

[28] S. Haiduc, J. Aponte, L. Moreno, and A. Marcus, "On the Use of Automated Text Summarization Techniques for Summarizing Source Code," in *2010 17th Working Conference on Reverse Engineering*. IEEE, oct 2010, pp. 35–44.

[29] C. Bird, P. C. Rigby, E. T. Barr, D. J. Hamilton, D. M. German, and P. Devanbu, "The promises and perils of mining git," in *2009 6th IEEE International Working Conference on Mining Software Repositories*. IEEE, may 2009, pp. 1–10.

[30] E. Kalliamvakou, G. Gousios, K. Blincoe, L. Singer, D. M. German, and D. Damian, "An in-depth study of the promises and perils of mining GitHub," *Empirical Software Engineering*, vol. 21, no. 5, pp. 2035–2071, oct 2016.

[31] D. M. German, B. Adams, and A. E. Hassan, "Continuously mining distributed version control systems: an empirical study of how Linux uses Git," *Empirical Software Engineering*, vol. 21, no. 1, pp. 260–299, feb 2016.

[32] M. Paixao and P. H. Maia. (2019) Replication package for the paper: "rebasing in code review considered harmful". Accessed: 2019-07-29. [Online]. Available: https://zenodo.org/record/3354510

[33] Eclipse. (2019) Eclipse projects. Accessed: 2019-07-29. [Online]. Available: https://eclipse.org/projects

[34] Couchbase. (2019) Couchbase open source projects. Accessed: 2019-07-29. [Online]. Available: https://developer.couchbase.com/open-source-projects

[35] M. Paixao, J. Krinke, D. Han, and M. Harman, "CROP: Linking Code Reviews to Source Code Changes." New York, New York, USA: ACM Press, 2018, pp. 46–49.

[36] ——. (2019) Code review open platform. Accessed: 2019-07-29. [Online]. Available: https://crop-repo.github.io/index.html

[37] N. Tsantalis, M. Mansouri, L. M. Eshkevari, D. Mazinanian, and D. Dig, "Accurate and efficient refactoring detection in commit history," in *Proceedings of the 40th International Conference on Software Engineering - ICSE '18*. New York, New York, USA: ACM Press, 2018, pp. 483–494.

[38] S. Just, K. Herzig, J. Czerwonka, and B. Murphy, "Switching to Git: The Good, the Bad, and the Ugly," in *2016 IEEE 27th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, oct 2016, pp. 400–411.

[39] V. Kovalenko, F. Palomba, and A. Bacchelli, "Mining file histories: should we consider branches?" in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering - ASE 2018*. New York, New York, USA: ACM Press, 2018, pp. 202–213.

[40] E. Kalliamvakou, G. Gousios, K. Blincoe, L. Singer, D. M. German, and D. Damian, "The promises and perils of mining GitHub," in *Proceedings of the 11th Working Conference on Mining Software Repositories - MSR 2014*. New York, New York, USA: ACM Press, 2014, pp. 92–101.