CROP: Linking Code Reviews to Source Code Changes

Matheus Paixao
University College London
London, United Kingdom
matheus.paixao.14@ucl.ac.uk

Jens Krinke
University College London
London, United Kingdom
jens.krinke@ucl.ac.uk

Donggyun Han
University College London
London, United Kingdom
d.han.14@ucl.ac.uk

Mark Harman
Facebook and University College London
London, United Kingdom
mark.harman@ucl.ac.uk

ABSTRACT

Code review has been widely adopted by both industrial and open source software development communities. Research in code review is highly dependant on real-world data, and although existing researchers have attempted to provide code review datasets, there is still no dataset that links code reviews with complete versions of the system's code base mainly because reviewed versions are not kept in the system's version control repository. Thus, we present CROP, the **C**ode **R**eview **O**pen **P**latform, the first curated code review repository that links review data with isolated complete versions (snapshots) of the source code at the time of review. CROP currently provides data for 8 software systems, 48,975 reviews and 112,617 patches, including versions of the systems that are inaccessible in the systems' original repositories. Moreover, CROP is extensible, and it will be continuously curated and extended.

CCS CONCEPTS

• **Software and its engineering → Software libraries and repositories**;

KEYWORDS

Code Review, Repository, Platform, Software Change Analysis

ACM Reference Format:
Matheus Paixao, Jens Krinke, Donggyun Han, and Mark Harman. 2018. CROP: Linking Code Reviews to Source Code Changes. In *MSR '18: 15th International Conference on Mining Software Repositories, May 28–29, 2018, Gothenburg, Sweden.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3196398.3196466

1 INTRODUCTION

In software development, code review is an asynchronous process in which changes proposed by developers are peer reviewed by other developers before being incorporated into the system [1]. The modern code review process has been empirically observed to successfully assist developers in finding defects [3, 10], transferring knowledge [1, 16] and improving the general 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 and Facebook use code review systems on a daily basis [5, 9].

In addition to its increasing popularity among practitioners, code review has also drawn the attention of software engineering researchers. There have been empirical studies on the effect of code review on many aspects of software engineering, including software quality [11, 12], review automation [2], and automated reviewer recommendation [20]. Recently, other research areas in software engineering have leveraged the data generated during code review to expand previously limited datasets and to perform empirical studies. As an example, in a recent study we used code review data to analyse whether developers are aware of the architectural impact of their changes [14].

Code review research relies heavily on data mining. In this context, some researchers have attempted to mine code review data and have made their datasets available for the community [6, 7, 13, 19]. However, code review data is not straightforward to mine (see Section 2.2), mostly due to difficulties in linking the reviews to their respective source code changes in the repository. This limits the potential research that can be carried out using existing code review datasets. In fact, to the best of our knowledge, there is no curated code review dataset that identifies and provides the complete state of the system's source code associated with a set of code reviews.

Based on this observation, we introduce CROP, the **C**ode **R**eview **O**pen **P**latform: a curated open source repository of code review data[1] that provides, not only the review's metadata like existing datasets, but also links, to each code review, a complete state of the system's source code at the time of review. For each code review in CROP, one will be able to access the source code that represents the complete state of the system when the review was carried out. Thus, researchers will now have the opportunity to analyse code review data in combination with, for example, source code analysis performed by static and dynamic techniques such as profiling, testing and building. The combination of code review data and source code analysis will facilitate research in areas that previously required a significant amount of human participation, as outlined in Section 4.

Gerrit [15] is a popular open source code review tool that has been widely used in research [4, 11, 14, 19]. In addition, notable open source organisations adopted Gerrit as their code review tool, including Eclipse, OpenStack and Couchbase. Thus, since CROP focuses on curating code review data from open source software

[1] https://crop-repo.github.io/

MSR '18, May 28–29, 2018, Gothenburg, Sweden

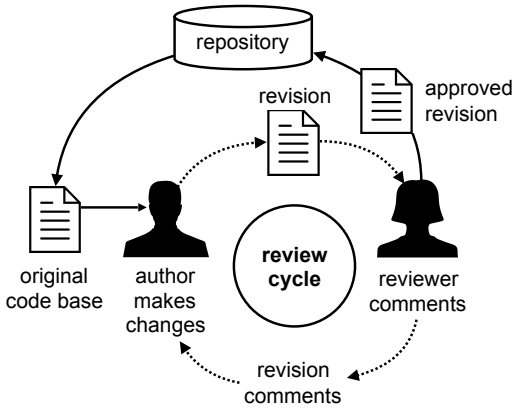

Figure 1: Code review process in Gerrit.

systems, we chose to mine data from projects that adopted Gerrit as their code review tool.

At the time or writing, we have mined the Gerrit code review data for 8 software systems, accounting for a total of 48,975 reviews and 112,617 patches (see Gerrit's structure in Section 2.1). In addition, we have mined and linked 225,234 complete source code versions to each of the 112,617 patches.

2 HARVESTING THE CROP

In this section we first describe the code review process employed by Gerrit followed by a discussion of the challenges of mining this data. Finally, we detail the approach we used to first mine the code review data and later link it to the system's source code.

2.1 Code Review in Gerrit

The Gerrit system is built on top of git, and its code review process is outlined in Figure 1. A developer starts a review by modifying the original code base in the repository and submitting a new **revision** in the form of a commit, where the commit message is used as the revision's description. For each new review, Gerrit creates a *Change-Id* to be used as an identifier for that review during its reviewing cycle. Other developers of the system will serve as reviewers by inspecting the submitted source code and providing feedback in the form of comments. Patches that improve upon the current revision according to the received feedback are submitted as new revisions of the same commit. Finally, the latest revision is either *merged* or *abandoned*, where the first indicates the review was incorporated to the system and the latter indicates the commit was rejected.

2.2 Challenges in Mining Gerrit Review Data

Gerrit provides RESTful APIs that one can use to access the review's metadata for a project, such as author, description, comments etc. However, linking the reviews to changes in the system's source code is far from straightforward.

As previously mentioned, Gerrit is built on top of git. Thus, the git repository of the system would be the obvious first choice to access the versions of the source code that correspond to the code reviews. However, the system's git repository is an unreliable record because Gerrit constantly *rewrites* and *deletes* history information.

When a new review is submitted by a developer, Gerrit creates a temporary branch in the git repository to be used for review. Every improved revision submitted by a developer is committed to this branch and *replaces* the previous revision through a *commit amend* operation. Therefore, given a merged review, the review's revision history is lost and only the source code of the latest accepted revision can be accessed. Moreover, when developers opt to abandon a review, the current revision is simply deleted from the repository.

In addition to the issues of lost history described above, the system's git repository might also contain inconsistencies if we fail to fully account for the overall review process: code review is a laborious task, and it is common for some reviews to take a few days to complete one iteration of the core cycle [8, 17, 18]. Between the time a comment is initially submitted and the time the revision is finally merged to the system's repository, other developers might have merged and/or committed other changes to the repository. In this case, each new revision submitted during the review needs to be *rebased* to be up-to-date. Thus, when one reverts the system back to the merged review, the source code will reflect not only the changes due to one but also all the other changes that were merged to the repository while the revision was open. These difficulties in isolating the source code changes associated with a specific review pose serious threats to the validity of empirical studies that use code review data.

2.3 Mining Code Review Data From Gerrit

We performed a preliminary analysis of the different open source communities that adopted Gerrit and selected the data source we would use for CROP's development. As a result, we identified the Eclipse and Couchbase communities as those that provided all the data we needed to build CROP. The data mining process we employed is outlined in Figure 2.

As one can see from the figure, our mining framework consists of 4 sequential phases. The framework is written in Python, and we made it available online[2]. Given a certain Eclipse or Couchbase project, the review harvester explores Gerrit's RESTful API to download the code reviews' metadata for the project. The API returns the data in JSON files that are kept to be used later.

In Phase 2, the snapshot harvester downloads the complete source code of the project for each code review. Both the Eclipse and Couchbase communities have a server that is separated from their git repositories and the Gerrit system where they keep complete snapshots of their projects for every commit ever performed in the project. These snapshots include the complete code base, i.e., source code, testing code, build files and so on. Thus, for each review, we iterate over all revisions and download the project's snapshots that correspond to the code base both *before* and *after* each revision.

As a result of this process, we were able access versions of the project's code base that would otherwise have been lost in the official git repository, such as reviews that were abandoned and intermediate revisions that were submitted during the review process. Moreover, by downloading the *before* and *after* versions of the code base for each revision, we check guarantee that the observed changes in the code were specifically attributable to the revision.

After downloading all the code reviews' metadata and the project's snapshots (Phases 1 and 2), Phases 3 and 4 handle the data. The discussion grinder processes the code review data stored in the JSON

[2] https://github.com/crop-repo/crop

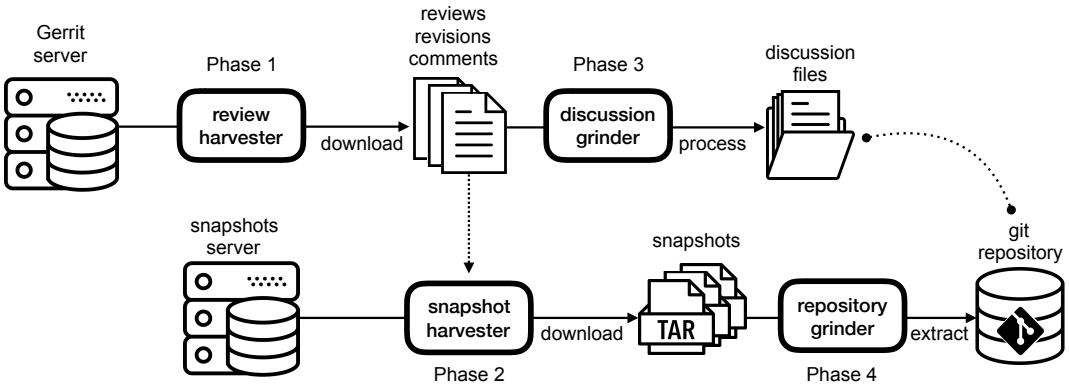

Figure 2: The framework employed to mine code review data from Gerrit and link it to complete versions of the code base.

files and creates discussion files for each revision. Discussion files are text files that present, among other information, the review's author, description and comments in a format that is easy to read and analyse (see Section 3 for more details).

In Phase 4, all downloaded snapshots are extracted to a new git repository in order to reduce the disk space occupied by CROP. The repository grinder creates a new git repository and then iterates over each snapshot, automatically extracting and committing the snapshot to the new repository. At the end, every snapshot will be accessible through the new git repository. If CROP would have included the snapshots' raw data from the 8 projects we have mined, the repository would have a size of 4.2TB. Instead, this approach reduced the size of CROP to 7.8GB, which accounts for a 99.8% reduction rate.

3 STORING THE CROP

The CROP repository is organised as three major directories. A CROP user starts at the metadata directory, where he or she will find a CSV file for each project in CROP. The CSV files contain metadata information about the project's code review and the necessary information to access the project's code base. Since we mined the projects' snapshots revision by revision, each line in the CSV corresponds to a project's revision.

For each revision, we create an **id** to serve as a unique identifier to the revision. The **review_number** column indicates the review to which the revision belongs. The **revision_number** denotes the number of the revision within the review. **Author** and **Status** indicate the revision's author and status, respectively. The **change-id** is the unique identifier that Gerrit creates, as explained previously. We provide a **URL** that can be used to access the revision's data in the Gerrit's web view. In the previous section, we showed how we created a new git repository to store the project's code base for each revision. Accordingly, the **before_commit_id** indicates the commit id in the new git repository that should be used to access the project's code base as it was before the revision was submitted. Similarly, one should refer to **after_commit_id** in order to access the code base as it was after the revision was submitted.

The git repositories we created to re-build the projects' reviewing history are contained in the git_repos directory. Each repository has a single master branch, where the before and after versions of the source code for each revision were committed sequentially, based on the review and revision numbers. Such versions are accessible through the commit ids provided in the projects' CSV file, as discussed above.

Table 1: Statistics about each project currently in CROP

Systems	Time Span	#Reviews	#Revisions	kLOC	Language
jgit	Oct-09 to Nov-17	5382	14027	200	Java
egit	Sep-09 to Nov-17	5336	13211	157	Java
linuxtools	Jun-12 to Nov-17	5105	15336	270	Java
platform-ui	Feb-13 to Nov-17	4756	14115	637	Java
ns_server	Apr-10 to Nov-17	11346	34317	253	JavaScript
testrunner	Oct-10 to Apr-16	7335	17330	117	Python
ep-engine	Feb-11 to Nov-17	6475	22885	68	C++
indexing	Mar-14 to Nov-17	3240	8316	107	Go

We store the discussion files for each revision in the discussion directory. This directory follows a tree structure, organised by review number, in which the discussion files for each revision are contained in the directory of its respective review. A discussion file presents reviewing data in the following order: first, the description of the revision is presented, which denotes the commit message of the revision. Such a message includes the revision's **change-id** and **Author**. The comments that were made during review by other developers are presented next. In the discussion file we include the author of the comment and the respective message.

4 GRINDING THE CROP

For the first iteration of CROP, we mined data from the four projects with most reviews at the time of mining [3] in the Eclipse and Couchbase communities. Table 1 reports statistics concerning the data collected for each of these 8 systems, where the Eclipse projects are presented in the upper section of the table and the Couchbase projects in the lower section. As one can see from the table, all projects have more than 3.5 years of reviewing history, where the data for egit spans more than 8 years. In addition, each project has more than 3,000 reviews and more than 8,000 revisions. In total, CROP provides comprehensive code review data linked to versions of the code base for 48,975 reviews and 112,617 revisions. Finally, these 8 projects are developed in a wide range of programming languages that include Java, C++, JavaScript, Python, Go and others.

[3] At the time of writing, papyrus is the project with most code reviews in Eclipse. However, it was not the case when we started mining.

4.1 Research Directions

As previously mentioned, there has been a good number of empirical studies that explored Gerrit data for different purposes [4, 7, 11–14, 19]. Hence, the data provided by CROP can now be used for extension, replication and validation of existing studies in code review, such as the ones mentioned above, as well as the formulation of new studies.

We envision the data provided by CROP being leveraged in a series of studies that span, not only code review, but also other areas of software engineering research. By linking code reviews to complete versions of the code base, we can now fully assess the impact of code review in the context of the code about which the reviews are written. Researchers may want to evaluate how code review influences building and testing, for example. Since CROP provides data for all revisions within a review, one might evaluate how the quality of a patch evolves from when it is first submitted to when it has finally been merged. We also provide source code for abandoned reviews, which enables the evaluation of, for example, whether the rejection of code reviews is somehow correlated to quality indicators in the submitted source code. By profiling developers' patterns and behaviour for a certain system, researchers may assess the effect of code reviews on the knowledge transferred between developers as the system evolves.

During code review, developers are constantly providing reasoning and rationale for the changes they make in the system, both when they submit code for review and when they inspect code from their peers. Thus, we see code review data as a scientifically valuable source of knowledge regarding motivation for and explanation of software changes. This knowledge can be leveraged to answer questions that previously required interactions with developers, such as interviews and surveys. One might use code review data to assess how developers react to the introduction and removal of code smells, for example. Similarly, one may investigate how developers deal with code duplication. By analysing different systems, one may be able to study how different teams reason about and discuss their maintenance activities, such as refactoring and bug fixing.

CROP is an ongoing research project, where we will periodically update the code review data to reflect the evolution of the systems in the dataset. In addition, we will be constantly mining and including reviewing data for other open source systems. Finally, CROP's code base is open[4] for contributions.

5 CONCLUSION

Code review has been widely adopted in the industrial and open source communities due to a number of benefits, such as knowledge transfer, defect detection, and code improvement. Although research in code review is highly dependent on datasets, there is currently no curated dataset that provides code review data that is linked to complete versions of the code at the time of reviewing. To address this limitation, we introduced CROP in this paper: the **C**ode **R**eview **O**pen **P**latform, an open source repository of code review data that provides links between code reviews and changes in the system's code base. We mined data of 8 software systems, accounting for 48,975 reviews and 112,617 revisions, where we provide not only code review information, but also links to versions of the source code that we archived and which would otherwise no longer be available in the systems' original repositories. By exploring the data provided in CROP, researchers can now study the effects of code review on the code base. Moreover, code review generates valuable and detailed information about software changes that can be leveraged in the development of other research areas in software engineering, such as code smells, code cloning and refactoring.

[4] https://github.com/crop-repo/crop

REFERENCES

[1] Alberto Bacchelli and Christian Bird. 2013. Expectations, outcomes, and challenges of modern code review. In *35th International Conference on Software Engineering (ICSE)*.

[2] Vipin Balachandran. 2013. 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)*.

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

[4] Amiangshu Bosu, Jeffrey C. Carver, Munawar Hafiz, Patrick Hilley, and Derek Janni. 2014. Identifying the Characteristics of Vulnerable Code Changes: An Empirical Study. In *International Symposium on Foundations of Software Engineering*.

[5] Joe Brockmeier. 2011. A Look at Phabricator: Facebook's Web-Based Open Source Code Collaboration Tool. (2011). http://readwrite.com/2011/09/28/a-look-at-phabricator-facebook/ Accessed in February, 2018.

[6] Georgios Gousios and Andy Zaidman. 2014. A Dataset for Pull-based Development Research. In *Working Conference on Mining Software Repositories (MSR)*.

[7] Kazuki Hamasaki, Raula Gaikovina Kula, Norihiro Yoshida, A. E. Camargo Cruz, Kenji Fujiwara, and Hajimu Iida. 2013. Who Does What During a Code Review? Datasets of OSS Peer Review Repositories. In *Working Conference on Mining Software Repositories (MSR)*.

[8] Yujuan Jiang, Bram Adams, and Daniel M. German. 2013. Will my patch make it? and how fast? Case study on the Linux kernel. In *Working Conference on Mining Software Repositories (MSR)*.

[9] Niall Kennedy. 2006. Google Mondrian: web-based code review and storage. (2006). https://www.niallkennedy.com/blog/2006/11/google-mondrian.html Accessed in February, 2018.

[10] M.V. Mantyla and C Lassenius. 2009. What Types of Defects Are Really Discovered in Code Reviews? *IEEE Transactions on Software Engineering* 35, 3 (May 2009).

[11] Shane McIntosh, Yasutaka Kamei, Bram Adams, and Ahmed E Hassan. 2014. 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 (MSR)*.

[12] Rodrigo Morales, Shane McIntosh, and Foutse Khomh. 2015. 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*.

[13] Murtuza Mukadam, Christian Bird, and Peter C. Rigby. 2013. Gerrit software code review data from Android. In *Working Conference on Mining Software Repositories*.

[14] Matheus Paixao, Jens Krinke, DongGyun Han, Chaiyong Ragkhitwetsagul, and Mark Harman. 2017. Are Developers Aware of the Architectural Impact of Their Changes?. In *International Conference on Automated Software Engineering (ASE)*.

[15] Shawn Pearce. 2006. Gerrit Code Review for Git. (2006). https://www.gerritcodereview.com Accessed in: April 2017.

[16] Peter C. Rigby and Christian Bird. 2013. Convergent contemporary software peer review practices. In *Foundations of Software Engineering (ESEC/FSE)*.

[17] Peter Weißgerber, Daniel Neu, and Stephan Diehl. 2008. Small patches get in!. In *International Workshop on Mining Software Repositories (MSR)*.

[18] X. Xia, D. Lo, X. Wang, and X. Yang. 2015. Who should review this change? Putting text and file location analyses together for more accurate recommendations. In *International Conference on Software Maintenance and Evolution (ICSME)*.

[19] X. Yang, R. G. Kula, N. Yoshida, and H. Iida. 2016. Mining the Modern Code Review Repositories: A Dataset of People, Process and Product. In *Working Conference on Mining Software Repositories (MSR)*.

[20] Motahareh Bahrami Zanjani, Huzefa Kagdi, and Christian Bird. 2016. Automatically Recommending Peer Reviewers in Modern Code Review. *IEEE Transactions on Software Engineering* 42, 6 (June 2016).