Gerrit Software Code Review Data from Android

Murtuza Mukadam
Concordia University
Montreal, QC, Canada
m_mukada@concordia.ca

Christian Bird
Microsoft Research
Redmond, WA, USA
cbird@microsoft.com

Peter C. Rigby
Concordia University
Montreal, QC, Canada
peter.rigby@concordia.ca

***Abstract*—Over the past decade, a number of tools and systems have been developed to manage various aspects of the software development lifecycle. Until now, tool supported code review, an important aspect of software development, has been largely ignored. With the advent of open source code review tools such as Gerrit along with projects that use them, code review data is now available for collection, analysis, and triangulation with other software development data. In this paper, we extract Android peer review data from Gerrit. We describe the Android peer review process, the reverse engineering of the Gerrit JSON API, our data mining and cleaning methodology, database schema, and provide an example of how the data can be used to answer an empirical software engineering question. The database is available for use by the research community.**

I. Introduction

The tools used to support software projects have provided a rich source of data for software engineering research. For example, common tools include source code management systems, build systems, bug databases, and test infrastructures. The results of such research have included insight into software development practices (e.g. [5]) and tools to aid practitioners (e.g. [6]). However, there are few mining scripts and datasets available for studying tool supported peer review.

Software inspection has been an engineering "best practice" for over 35 years [3]. Email based Open Source Software (OSS) peer review has been extensively studied and been shown to be effective [7]. It is important to understand how tool supported peer review is impacting the effectiveness of this engineering practice. In this data paper, we describe how we mine the Gerrit [4] peer review system to extract reviews done by Android developers. We provide a dataset that includes information about which software changes are reviewed (and implicitly, which are not), who typically looks at such changes, how long reviews take, and what types of discussions and feedback are given during code review.

The paper is structured as follows: we describe the Gerrit based peer review process used by the Android project (section II), the source of the data, the methods we used to collect the data along with challenges and limitations (section III), a description of the data schema (section IV), and finally we show an example of how such data can be used to answer questions relating to review practices and discuss future avenues of research with the data (section V).

II. Description of Project and Data

Android is an operating system developed with a goal to create real-world products which improves the experience for users using mobile and tablet devices. It was initiated by Android Inc. , and was bought by Google in 2005. Open Source Software and was initiated by a group of companies known as Open Handset Alliance in 2007, which is led by Google [1]. The Android community uses the free web based software code review tool Gerrit [4]. We downloaded a total of 19k reviews from Gerrit.

Gerrit is integrated with git and serves as a barrier between developers' private repositories and the official, centralized Android source tree [2]. Developers make local changes and then submit these changes for review. Reviewers make comments via the Gerrit web interface. For a change to be merged into the Android source tree, it must be approved and verified by a senior developer. Android is an example of a review-then-commit policy [7] that has additional change approval steps [2]:

1) "Verified" - Before a review beings, someone must verify that the change merges with the current master branch and does not break the build. In many cases, this step is done automatically.
2) "Approved" - While anyone can comment on the change, someone with appropriate privileges and expertise must approve the change.
3) "Submitted/Merged" - Once the change has been approved it is merged into Google's master branch so that other developers can get the latest version of the system.

The example in Figure 1 illustrates a review in Android.[1] A Gerrit review begins when the owner (Shuo Gao) posts a patch to be reviewed. Reviewers are assigned (Jeffrey Brown, Christophe Bransiec, *etc.*) so that they can take part in the reviewing of the patch uploaded by the owner. Unassigned reviewers can also make comments. Reviewers can provide comments on individual lines that have changed (2 comments) or they can provide general comments (Jean Baptiste Queru comments "Patch Set 1: Verified"). Reviewers can approve (Christophe Bransiec gives a value +1) or reject (Jeffrey Brown, a value of -2) the uploaded patch. The bot (Deckard Autoverifier) comments, "Patch Set 2 is verified". A patch set encapsulates details regarding the author, committer and also the inline comments made by the reviewers. Multiple patch sets can be uploaded during a review (2 patch sets have been uploaded).

[1] https://android-review.googlesource.com/#/c/41591/

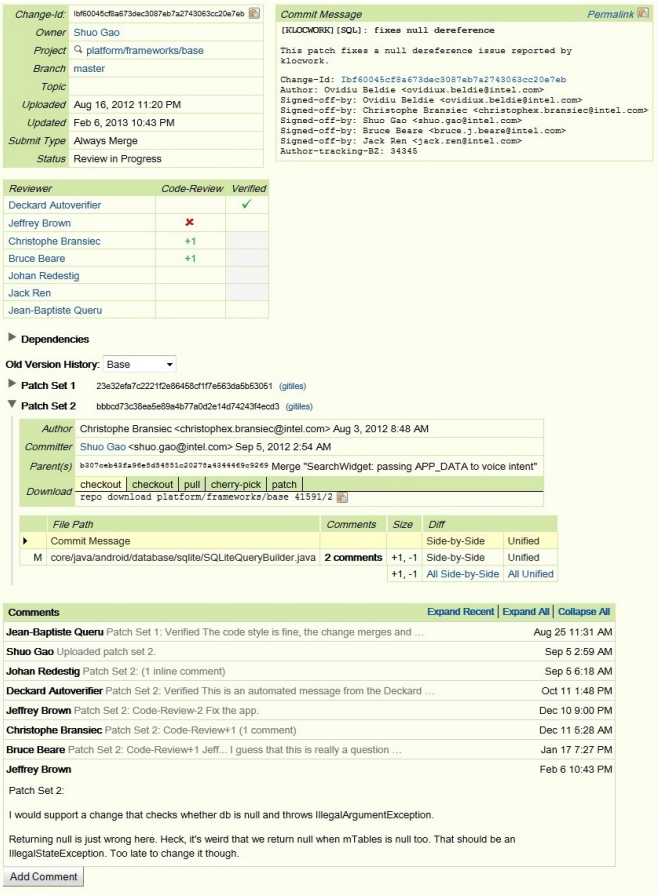

Fig. 1. Gerrit Android Review Number 41591

III. Extraction Method

Android review data is stored in Gerrit.[2] We were able to avoid screen-scraping by observing how Gerrit web pages are constructed. Gerrit works by initially sending a web page skeleton and some Javascript to the browser. The Javascript then makes a number of web requests back to the Gerrit server and requests information about code reviews, which is returned in JSON format. The page DOM is then modified by the Javascript to display the code review information. When we developed our script, the Gerrit REST API provided limited information. However, the current Gerrit API provides an interface to JSON formatted review data.[3] This JSON data must still be parsed.

We used the developer tools within the Chrome web browser to inspect these web requests (inspecting header fields and POST data) to the Gerrit server along with the responses in an effort to reverse engineer the types of web methods available and the structure of the JSON data returned by the requests. We also determined which fields in the displayed web pages corresponded to what fields within the JSON. The JSON returned was fairly complex, deep, and redundant (it was not uncommon for a single JSON response for a code review to exceed 50 kilobytes). In addition, many web requests were needed to obtain all information about an individual review.

[2]https://android-review.googlesource.com

[3]https://gerrit-review.googlesource.com/Documentation/rest-api.html


```
currentPatchSetId: {changeId:{id:41591}, patchSetId:2}
  changeId: {id:41591}
    id: 41591
  patchSetId: 2
dependsOn: []
messages: [{key:{changeId:{id:41591}, uuid:AAAI6X///9M=}, author:
neededBy: []
patchSets: [{id:{changeId:{id:41591}, patchSetId:1}, revision:{id
  0: {id:{changeId:{id:41591}, patchSetId:1}, revision:{id:23e32e
    createdOn: "2012-08-17 03:20:48.036000000"
    draft: false
    id: {changeId:{id:41591}, patchSetId:1}
      changeId: {id:41591}
        id: 41591
      patchSetId: 1
    revision: {id:23e32efa7c2221f2e86458cf1f7e563da5b53051}
    uploader: {id:1014559}
```


Fig. 2. JSON Response from the server

One review might contain many rounds of *patch sets* (an author may submit one set of changes, get feedback, submit a set of revised changes, etc.). Obtaining the information for each patch set requires an additional web request, and gathering the reviewer comments for each file within each patch set requires yet another. Thus, a review might require over twenty to thirty individual web requests. In an effort to avoid overloading the Gerrit server (and also avoid our IP being blacklisted from the site), we throttled our mining by delaying one second between requests.

We developed a Python script that made use of the various web methods and extracted the relevant data from the JSON responses. We also created a database schema based on the information returned from the server and the data was stored in a Microsoft SQL Server database for later analysis. To enable broad use of the data, we provide an SQL Server database backup file as well as a simple XML dump of the data.[4]

Data Extraction Details and Example

We reverse engineered the JSON requests to get the "ChangeDetailService" and "PatchDetailService". In Figure 2, we show a snippet of the JSON returned when we sent a request for the ChangeDetailService for the review in Figure 1. We store the raw data from each JSON request, so that we can re-process the data without sending requests to the server.[5] Our Python script then extracts data from the JSON into a database. For example, the patch id (["result"]["patchSet"]["id"]["patchSetId"]=2), change type (['result']['patches']='A'), lines added (['result']['patches']=19), and lines removed (['result']['patches']=0).

A. Challenges and Limitations

We describe some of the data limitations and some challenges we overcame while cleaning anomalies from the data. We hope that as this dataset becomes more widely used for answering empirical software engineering research questions, other challenges and limitations will be identified and removed from the data.

Challenge: Gerrit JSON API

The Gerrit JSON API is the only way to get all information from Gerrit. While the API is intended for public use, it is

[4]Script and data is available at https://github.com/mmukadam/gerrit-miner.git

[5]Please contact us for a dump of the raw data

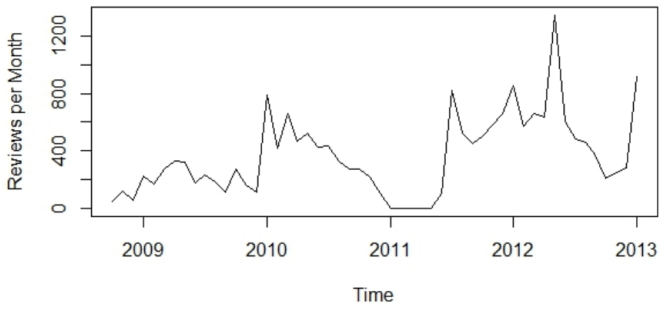

Fig. 3. Number of Reviews started per Month

not formally defined and there is no documentation for it. As we discussed above we reverse engineered the JSON requests and responses. However, the API can change between versions of Gerrit and the location and names of services can also change. For example, while mining Android, the location of the "ChangeDetailService" was moved from `/gerrit/rpc/ChangeDetailService` to `/gerrit_ui/rpc/ChangeDetailService`

Challenge: A Bot and other Anomalies

Looking for anomalies in the data, we noticed that a bot called "Deckard Autoverifier" was involved in 1.8k reviews. Qualitative analysis revealed that the bot was responsible for ensuring that the change merged with the master branch without conflict and that it did not break the build – it was "Verified". Since the bot is responsible for automatically verifying new changes, we expected there to be one verification for each patch set. However, there are 19k reviews and the bot is involved in only 1.8k. A mailing list discussion[6] revealed that the "Deckard Autoverifier" cannot verify inter-project dependencies, so many verifications must be done manually. For example, on AOSP "Jean-Baptiste Queru" manually verifies all new changes. Since Queru does many manual verifications, he will have commented on an artificially large number of review. Depending on the goal of future analysis, verifications with no other comments may be tagged or removed.

Challenge: Collecting all Reviews

Reviews are identified by an id number; however, not every review number contains a valid id. We download all reviews between review number 1 to 51750, which resulted in 19k reviews. Figure 3 plots the number of reviews per month. While the number of reviews can fluctuate drastically in a given month [7], there is evidence on the Android mailing lists that the Gerrit database has been cleaned at various points in time, removing stalled reviews, but leading to missing data. For example, many reviews are missing from August 2011 until the start of January 2012. A solution would be to regularly mine the Android Gerrit data.

[6]https://groups.google.com/forum/#!msg/android-contrib/cEFSGewsqUQ/umd5FKrv4_QJ

IV. Data Storage and Schema

The database schema is depicted in Figure 4. In general, each table has an *Id* column that is unique for each entry in the table (e.g., the *Review* table has *ReviewId*, the *Person* table has *PersonId*). Foreign key relations are indicated by the presence of a column in one table that has the same name as the primary key of another table.

Reviews

The *Review* table contains information about the review itself. This includes the review identification number used by Gerrit (the primary key), the person that created the review and typically made the change (*OwnerId*), the creation time (*CreatedOn*), the last time that any activity occurred on the review (*LastUpdatedOn*), a one line description of the change (*Subject*) along with a description (*message*), the project that the review is for, and the branch within git that the change was made on. The *Status* can be "open", meaning that the review is active and the change has not yet been accepted,"merged", meaning that the review has passed and the change has been merged into the codebase, and "abandoned", indicating that the review has not passed and is no longer active.

People

Many tables include references to people (reviewers, authors, *etc.*) through the use of a *PersonId*. The *Person* table maps this id to the person's *Name* and *Email* address. We have observed that some automated system accounts also add information to reviews. For example, one "bot" adds comments to a review that describe changes to the review. These accounts can be identified due to the *IsBotAccount* being set to 1.

Patch Sets

A change for review is made up of a set of files that correspond to a commit. In Gerrit parlance, this is called a patch set. As an author responds to feedback, he may submit multiple patch sets until the final patch set is accepted. The *PatchSet* table includes information including whether this is the first, second, third, *etc.* patch set for a review (*PatchSetNumber*), the number of files in the patch set (*NumberOfFiles*), when it was created (*CreatedOn*), and the revision within the git repository that contains the versions of the files in the patch set (*GitRevision*).

Patch Set Files

Each file within a patch set has an entry in the *PatchSetFile* table. This includes the path of the file (*Path*), how many lines were added and deleted, and the *ChangeType*, which indicates if the file was added, removed, or modified. We do not store this in the database. It is easy to obtain this with the *GitRevision* from the corresponding *PatchSet* entry.

Comments

The information about each comment is in the *Comment* table. This includes the text of the comment (*Message*), when the comment was made (*WrittenOn*), who wrote the comment (*AuthorId*), and which patch set the comment is relevant to

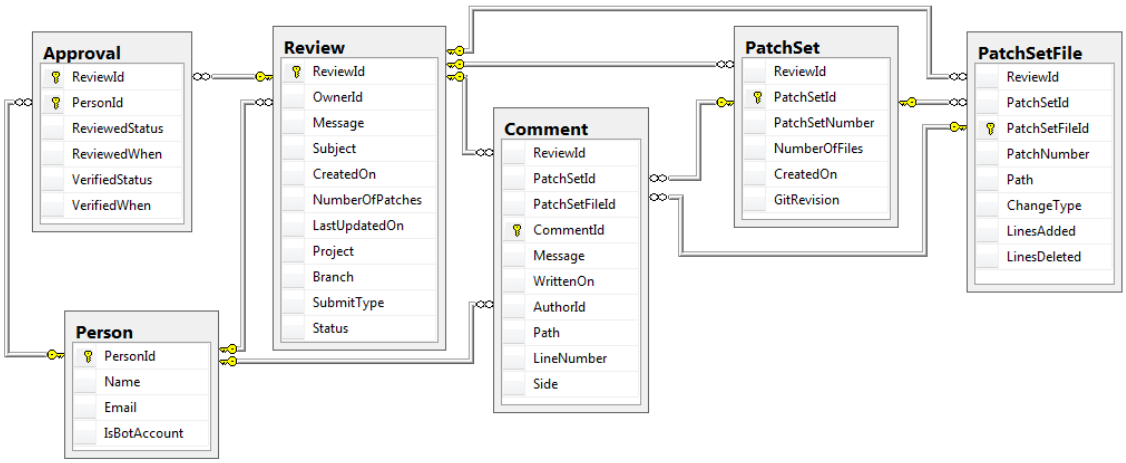

Fig. 4. Database Schema

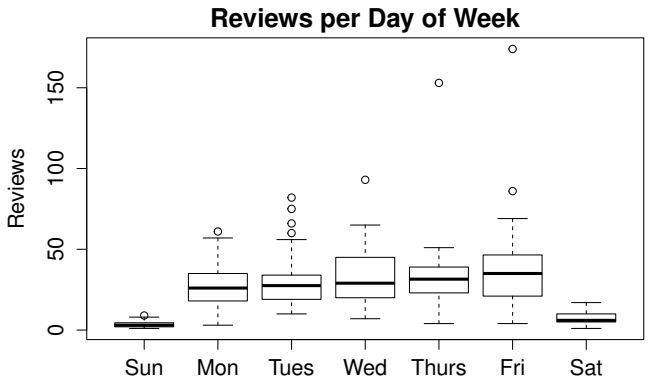

Fig. 5. Number of reviews submitted per day of the week over the last half of 2012 in Android.

(*PatchSetId*). Comments can refer to a particular location within a file in the patch set, in which case the file path (*Path*), line within the file (*LineNumber*), and whether the comment refers to the version of the file prior to or after the change (*Side*) are indicated. Otherwise, these fields will be NULL.

Approvals

Each review is given certain points(-2,-1,0,1,+2) based on whether reviewers judge that it should be accepted or rejected. This information is stored in the *Approval* table, with an entry for each person involved in a review. This is the only table that doesn't have a single field primary key, as the *ReviewId* and the *PersonId* uniquely identify the entry and act as a conjugate primary key. The information in this table indicates whether the reviewer has signed off on the review (*ReviewedStatus*) and/or verified that the change does not cause problems (*VerifiedStatus*) and stores when these occurred (*ReviewedWhen* and *VerifiedWhen*).

V. Future Work

In a forthcoming paper in preparation for submission, we have used this data to quantitatively characterize code review in Android and compare Android code review to review in other contexts. However, here we present a simple illustration of the types of questions that can be answered using this data. We want to know how developers apportion their work over the course of their work week.

Figure 5 shows boxplots that describe the distribution of the number of code reviews submitted per weekday over the past six months in the Android project. Using t-tests (since the submission numbers per day follow normal distributions) we determined that there is no statistical difference between Monday, Tuesday, Wednesday, Thursday, and Friday. However, they all show a statistically significant increase over both Saturday and Sunday and Saturday has more than Sunday to a statistically significant degree. Thus, one can reasonably conclude that the contributors to Android work during the week and take weekends off.

We are currently using this data set, other OSS datasets, and datasets from software firms to understand how different software development environments affect peer review practices. Other research avenues include using this new dataset to improve models of defect prediction, identifying attributes of changes that lead to many comments from reviewers or many iterations of change submission, and characterizing review patterns of developers who join software projects.

References

[1] Android. Android Open Souce Project. http://http://source.android.com/index.html.
[2] Android. Submitting patches. http://source.android.com/source/submit-patches.html.
[3] M. Fagan. Design and Code Inspections to Reduce Errors in Program Development. *IBM Systems Journal*, 15(3):182–211, 1976.
[4] Gerrit. Web based code review and project management for git based projects. http://code.google.com/p/gerrit/.
[5] A. Hindle, D. M. German, and R. Holt. What do large commits tell us?: a taxonomical study of large commits. In *MSR*, 2008.
[6] R. Holmes and A. Begel. Deep intellisense: a tool for rehydrating evaporated information. In *MSR*, 2008.
[7] P. C. Rigby, D. M. German, and M.-A. Storey. Open Source Software Peer Review Practices: A Case Study of the Apache Server. In *30th ICSE*, pages 541–550, 2008.