# Peer Review on Open-Source Software Projects: Parameters, Statistical Models, and Theory

PETER C. RIGBY, Concordia University
DANIEL M. GERMAN, LAURA COWEN, and MARGARET-ANNE STOREY,
University of Victoria

Peer review is seen as an important quality-assurance mechanism in both industrial development and the open-source software (OSS) community. The techniques for performing inspections have been well studied in industry; in OSS development, software peer reviews are not as well understood.

To develop an empirical understanding of OSS peer review, we examine the review policies of 25 OSS projects and study the archival records of six large, mature, successful OSS projects. We extract a series of measures based on those used in traditional inspection experiments. We measure the frequency of review, the size of the contribution under review, the level of participation during review, the experience and expertise of the individuals involved in the review, the review interval, and the number of issues discussed during review. We create statistical models of the review efficiency, review interval, and effectiveness, the issues discussed during review, to determine which measures have the largest impact on review efficacy.

We find that OSS peer reviews are conducted asynchronously by empowered experts who focus on changes that are in their area of expertise. Reviewers provide timely, regular feedback on small changes. The descriptive statistics clearly show that OSS review is drastically different from traditional inspection.

Categories and Subject Descriptors: D.2.8 [**Software Engineering**]: Metrics; K.6.3 [**Management of Computing and Information Systems**]: Software Management—*Software development; Software maintenance; Software process*

General Terms: Management, Measurement

Additional Key Words and Phrases: Peer review, inspection, open-source software, mining software repositories

**ACM Reference Format:**
Peter C. Rigby, Daniel M. German, Laura Cowen, and Margaret-Anne Storey. 2014. Peer review on open-source software projects: Parameters, statistical models, and theory. ACM Trans. Softw. Eng. Methodol. 23, 4, Article 35 (August 2014), 33 pages.
DOI: http://dx.doi.org/10.1145/2594458

## 1. INTRODUCTION

Over the years, formal peer reviews (inspections) have been perceived as a valuable method for improving the quality of a software project. Inspections typically require periodic group reviews. Developers are expected to prepare for these meetings by studying the artifact under review, and then they gather to discuss it [Fagan 1976; Laitenberger and DeBaud 2000; Kollanus and Koskinen 2009].

In practice, industrial adoption of software inspection remains low, as developers and organizations complain about the time commitment and corresponding cost required

This article is a revised and extended version of a paper presented at the 2008 International Conference on Software Engineering (ICSE) in Leipzig, Germany [Rigby et al. 2008]. The work is largely based on Rigby's dissertation at the University of Victoria [2011] and on an earlier technical report [Rigby and German 2006].
 1049-331X/2014/08-ART35 $15.00
DOI: http://dx.doi.org/10.1145/2594458

for inspection, as well as the difficulty involved in scheduling inspection meetings [Johnson 1998]. These problems are compounded by tight schedules that make it easy to ignore peer reviews.

Given the difficulties with adoption of inspection techniques in industry, it is surprising that most large, successful projects within the open-source software (OSS) community have embraced peer review as one of their most important quality-control techniques. Despite this adoption of peer review, there are very few empirical studies examining the peer review techniques used by OSS projects. There are experience reports [Lussier 2004; Raymond 1999], descriptions at the process and policy level [Dinh-Trong and Bieman 2005; Mockus et al. 2000], and empirical studies that assess the level of participation in peer reviews [Asundi and Jayant 2007; Lee and Cole 2003]. There has also been some more recent work that examines OSS review when conducted on bug-tracking tools, such as Bugzilla [Breu et al. 2010; Jeong et al. 2009; Nurolahzade et al. 2009]. In this article, we examine the widely used email-based style of OSS peer review.

A review begins with an author creating a patch (a software change). The author can be anyone from an experienced core developer to a novice programmer who has fixed a trivial bug. The author's patch, which is broadcast on the project's mailing list, reaches a large community of potentially interested individuals. The patch can be ignored[1], or it can be reviewed with feedback sent to the author and also broadcast to the project's community. The author, reviewers, and potentially other stakeholders (e.g., nontechnical users) discuss and revise the patch until it is ultimately accepted or rejected.

The study of software inspection for the last 35 years has produced a set of parameters of inspection that can be measured in order to determine the efficacy of an inspection technique. For example, the expertise of reviewers, the review interval, and the effectiveness of review. Our goal is to measure similar parameters to provide an empirically examination of peer review across six successful, mature OSS projects and to model the efficiency and effectiveness of each projects' review technique. By measuring these parameters of review, we expand what is known about OSS and can identify techniques from OSS peer review that might be valuable and transferable to industry or vice versa.[2] We have chosen the Apache httpd server (referred to as Apache in this work), Subversion, Linux, FreeBSD, KDE, and Gnome to study, because each project is widely used, large, and mature. While future work may study unsuccessful or small projects, we feel that it is important to first understand the review techniques originating in successful projects that manage a large number of reviews and have a high degree of collaboration. This article replicates and augments the parameters measured in Rigby et al.'s [2008] case study of the Apache server. A further contribution of this work is statistical models of the peer review processes that help us determine which parameters have the largest impact on review efficacy.

The article is organized as follows. Section 2 lays out our research questions and discusses related work. Our questions mirror those asked of traditional inspection techniques. In Section 3, we present the types of review that we extracted from the review policies of 25 successful OSS projects. Section 4 describes our quantitative research methodology, data, and data extraction techniques. Section 5 answers our research questions by extracting a series of measures on archival data. In Section 6, we develop statistical models to determine which measures have the largest impact on review efficiency and effectiveness. In Section 7, we introduce our theory of OSS peer review, describe our findings in the context of the software inspection literature, and discuss the validity of our study.

[1]Contributions that receive no review responses are not studied in this work.

[2]A summary of our findings for practitioners can be found in IEEE Software [Rigby et al. 2012].

## 2. RESEARCH QUESTIONS AND RELATED WORK

To facilitate comparison of our findings with those of the last 35 years of inspection research, we base our research questions upon those that have previously been asked and answered for inspection techniques (e.g., [Ackerman et al. 1989; Cohen 2006; Porter et al. 1998, 1997]). Each set of research questions are operationalized as a set of measures. Since these measures depend on the type of data used and are often proxies for the actual quantity we wish to measure, the measures are introduced, along with any limitations, in the section in which they are used. Each measure will provide us with a variable to use in our statistical models of review efficacy.

*Q1 Frequency and Activity.* What is the frequency of review? Can reviewing keep up with development or do we see an increase in the proportion of unreviewed contributions when more commits are being made?

OSS peer review policies enforce a review around the time of a commit. For pair programming, reviews are conducted continuously [Cockburn 2004], while for inspections, reviews are usually conducted infrequently on completed work products [Fagan 1986]. As development activity increases, so does the number of contributions and commits. In OSS, if the level of reviewing does not increase with development activity, this could mean that contributions could go unreviewed. To study this concern, we correlate review frequency to development activity.

*Q2 Participation.* How many reviewers respond to a review? How much discussion occurs during a review? *What is the size of the active reviewer group?*

In his experience-based analysis of the OSS Linux project, Raymond [1999] coined Linus's Law as "Given enough eyeballs, all bugs are shallow." It is important to gauge participation during peer reviews to assess the validity of this statement. Earlier research into the optimal number of inspectors has indicated that two reviewers perform as well as a larger group [Buck 1981; Porter et al. 1998; Sauer et al. 2000]. Previous OSS research has found that there are on average 2.35 reviewers who respond per Linux review [Lee and Cole 2003]. Similar findings were attained when replications were performed on other projects [Bird et al. 2006, 2007]. The amount of discussion is also measured to gauge participation by counting the number of messages exchanged during a review. We add a new measures to gauge the size of the reviewer group (i.e., the number of people participating in reviews on a regular basis) at monthly intervals.

*Q3 Expertise and Experience.* For a given review, how long have the authors and reviewers been with the project? How much work has a developer done on the project?

Expertise has long been seen as the most important predictor of review efficacy [Porter et al. 1998; Sauer et al. 2000]. We measure how much experience and expertise authors and reviewers have based on how long they have been with the project and the amount of work they do. Based on the experiences of prominent OSS developers [Fielding and Kaiser 1997; Fogel 2005; Raymond 1999], developers self-select work that they find interesting and for which they have the relevant expertise. Two papers [Asundi and Jayant 2007; Rigby and German 2006] indicate that a large percentage of review responses are from the core group (i.e., experts). We expect OSS reviews to be conducted by expert developers who have been with the project for extended periods of time.

*Q4 Churn and Change Complexity.* How does the size of change (churn) affect peer review?

Mockus et al. [2002] found that the size of a change, or churn, for the Apache and Mozilla projects were smaller than for the proprietary projects they studied, but they

did not understand or investigate why. We investigate the relationship between OSS review policy and practice with the size of the artifact under review. We want to understand whether the small change size is a necessary condition for performing an OSS-style review.

*Q5 Review Interval.* What is the calendar time to perform a review?

The review interval, or the calendar time to perform a review, is an important measure of review effectiveness [Kollanus and Koskinen 2009; Porter et al. 1998]. The speed of feedback provided to the author of a contribution is dependent on the length of the review interval. Review interval has also been found to be related to the overall timeliness of a project. For example, Votta [1993] has shown that 20% of the interval in a traditional inspection is wasted due to scheduling. Interval is one of our response variables, and in Section 6, we create a statistical model to determine how the other variables influence the amount of time it takes to perform a review.

*Q6 Issues.* How many issues are discussed during a review?

Fagan [1976] measured the effectiveness of inspection by counting the number of defects, and then, based on the cost of fixing defects later in the development cycle, he determined the time savings of finding defects early. Variations on Fagan's measure of the number of defects found have been used in many subsequent studies of review effectiveness [Votta 1993; Knight and Myers 1993]. Research on inspection has established inspection as an effective defect removal technique, with the result that researchers usually calculate the number of defects found instead of the ultimate cost savings to determine the effectiveness of a review process [Kollanus and Koskinen 2009]. We do not have data that would allow us to measure the costs savings on OSS projects, so we create a proxy measure of the number of defects found.

OSS developers do not record the number of defects found during review, so we measure: the number of issues found during review. An issue, unlike a true defect, includes false positives and questions. For example, an issue brought up by a reviewer may actually be a problem with the reviewer's understanding of the system instead of with the code. In Section 6, we statistically model the number of issues discussed per review to understand which of the variables previously described have the greatest impact on review effectiveness.

## 3. BACKGROUND: REVIEW PROCESSES AND CASE STUDIES

Before we quantify OSS review processes, we study the review processes that are used on successful OSS projects. The accurate determination of success in OSS is difficult to quantify [German 2007], and there are thousands of potentially successful OSS case studies [Freshmeat 2009; SourceForge 2010; Howison et al. 2006]. When it is not obvious which cases must be examined, Yin [2003] recommends collecting "limited information on a large number of cases as well as intensive information on a smaller number."

We use two approaches to identify potential case studies. First, we use an online "catalogue" of thousands of OSS applications and sample the 19 highest ranked projects [Freshmeat 2009]. Second, we examine six high-profile projects which have obvious measures of success (e.g., dominance in a particular software domain). We manually classify the review processes of these 25 projects. We visited each project's main development page and searched for links relating to peer review or patch submission. We noted the type of project, the review types and policies used on the project, and any other observations, such as the size of the development team and the governance structure. Figure 1 illustrates the steps in the two most common review types. Table I shows the types of review policies used on each project. Since our unit of analysis is the individual peer reviews, the pertinent "limited information" is the

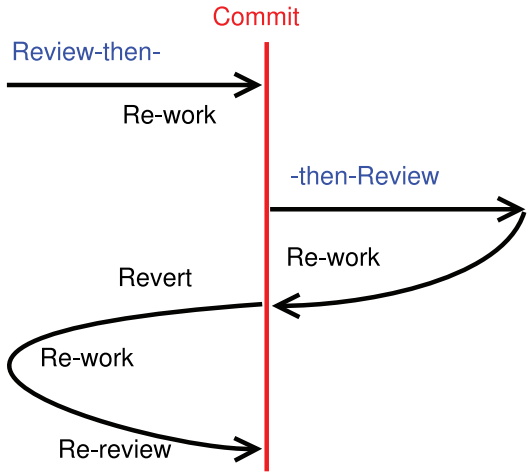


Fig. 1. Review processes: RTC and CTR.

Table I. Review Types Used by the 25 Projects Examined

| Project | Review Types |
|---|---|
| Apache | RTC, Lazy RTC, CTR |
| Subversion | RTC, CTR |
| Linux | Maintainer RTC |
| FreeBSD | Maintainer RTC, CTR |
| KDE | RTC, CTR, ReviewBoard tool |
| GNOME | RTC, CTR, Bugzilla |
| Mozilla | "Strong" RTC in Bugzilla |
| Eclipse | RTC in Bugzilla |
| GCC | Maintainer RTC |
| cdrtools | Small, stable no formal review |
| Postgresql | RTC in Commitfest |
| VLC | RTC |
| MPlayer | RTC, CTR |
| Clam AntiVirus | Commercially run, no policy |
| MySQL | "Strong" RTC |
| PHP | Informal RTC, CTR, Bugzilla |
| PHPMyAdmin | Informal RTC, CTR |
| NTop | Informal RTC, Trac tool |
| TightVNC | RTC using Sourceforge tools |
| GTK+ | Bugzilla |
| LibJPEG | Small, stable no formal review |
| WebGUI | Commercially run, no policy |
| NMap | RTC |
| Samba | RTC, CTR |

review process, which we summarize next. Rigby [2011, Appendix A] contains the details of the attributes and context for each project.

### 3.1. Review Process

Although the level of formality of the review processes varies among OSS projects, the general steps involved in review are as follows:(1) the author submits a contribution

(a patch or diff) by emailing it to the developer mailing list, (2) one or more people review the contribution, (3) it is modified until it reaches the standards of the community, and (4) it is committed to the code base. This style of review is called review-then-commit (RTC). In contrast to RTC, some projects allow trusted developers to commit contributions (i.e., add their contributions to the shared code repository) before they are reviewed. According to the policy statements of OSS project, the main or core developers for the project are expected to review all commits. This style of review is called commit-then-review (CTR). Figure 1 illustrates the differences between RTC and CTR. CTR is less commonly used than RTC but has consistent application across project. For RTC, there are five variants: informal, strong, maintainer RTC, lazy, and tracker-based.

—*Informal RTC* exists when there is no explicit policy for review, but contributions are sent to the mailing lists where they are discussed. This is the most common type of review in OSS.
—In contrast, *strong* RTC occurs when all developers must have their code reviewed before committing it regardless of their status within the community. For example, on the MySQL and Mozilla projects, all developers, including core-developers, must have two reviewers examine a change before it can be committed. When a project uses strong RTC, CTR is not used.
—*Maintainer RTC* occurs on large projects that have explicit code ownership. For example, GCC, Linux, and FreeBSD use this style of review. In this case, developers must get the approval of the code's maintainer before committing any changes in that part of the system.
—*Lazy RTC*, as used on Apache and Subversion, occurs when a core developer posts a change to the mailing lists, asking for feedback within a certain time period. If nobody responds, it is assumed that other developers have reviewed the code and implicitly approved it.
—*Tracker-Based RTC* occurs when the review is conducted on a Web-based tracking tool (e.g., Bugzilla, ReviewBoard) instead of on a mailing list. Tracker-based review is outside the scope of this work. However, bugtracker review is used by several projects that have been studied by other researchers, including Eclipse [Breu et al. 2010], Mozilla [Jeong et al. 2009; Nurolahzade et al. 2009], KDE, and GNOME. On some projects, such as Apache and Linux, a bugtracker is used, but according to project policy, all reviews are still performed on the developers mailing list; the bugtracker is simply for reporting bugs.

Aside from the actual processes of review, there are two policies that apply to all changes to OSS projects. First, a contribution must be small, independent, and complete. Reviewers do not want to review half-finished contributions or contributions that involve solutions to multiple unrelated problems (e.g., a change that involves fixing a bug and correcting the indentation of a large section of code). Second, on projects with a shared repository, if any core developer feels that a contribution is unacceptable, he or she can place a veto on it, and the contribution will not be committed or, in the case of CTR, it will be removed from the shared repository.

### 3.2. Detailed Case Study Replications

Having developed a broad understanding of the review policies on successful OSS projects, we used theoretical sampling to sequentially select six well-known projects to examine intensively [Yin 2003]. The projects are presented in order in which they were studied and represent both literal and contrasting replications.

*Apache* is a the most widely used httpd server [Netcraft 2010]. Apache is a successful, medium-sized project that is run by a foundation. It has been the focus of many empirical investigations because early on it formalized and enforced its project

policies [Mockus et al. 2000; Gutwin et al. 2004; Bird et al. 2006; Rigby et al. 2008]. Some OSS projects state that they are doing "Apache Style" development [Fogel 2005].

*Subversion* is a version control system that was designed to replace CVS. Subversion is a good first test of our evolving theory because it borrowed many of its policies from the Apache project and several of the original Subversion developers were also involved in the Apache project [Fogel 2005]. It is also run by a foundation and is similar in size to Apache.

*FreeBSD* is both a UNIX based kernel and a UNIX distribution. FreeBSD is similar to the previous projects in that it is governed by a foundation and has similar styles of review to Apache and SVN. However, it is a much larger project than either of the previous cases and tests the impact of size on peer review.

*Linux Kernel* is a UNIX based kernel, like FreeBSD, but also contrasts sharply with the first three projects on the governance dimension. It is governed by dictatorship instead of by a foundation. This difference in governance means that Linux can only use RTC and that patches are passed up a "chain-of-trust" to the dictator.

*KDE* is a desktop environment and represents not a single project, as was the case with all the previous projects, but an entire ecosystem of projects. KDE also contrasts with the other projects in that end-user software as well as infrastructure software is developed. By being a composite project the relationship between each individual subproject is less well defined. We are interested in understanding how a diverse and large community like KDE conducts peer review. The exact review process varies among subprojects.

*GNOME*, like KDE, is a desktop environment and an ecosystem of projects. Developers on this project write infrastructure and end user software. For both KDE and GNOME, reviews were at one point conducted exclusively over email. However, many reviews on GNOME are now being performed in Bugzilla, and on KDE, in Bugzilla or ReviewBoard, depending on the subproject.

## 4. METHODOLOGY AND DATA SOURCES

We scope this work around the parameters of peer review that have been measured in the past. We only consider patch contributions that receive a response, because a contribution to the mailing list that is ignored by the developer's peers is not a review. For a qualitative perspective, see Rigby and Storey [2011] and Rigby's dissertation [2011], where they interviewed core developers and manually inspected 500 review threads to understand the interactions, roles, and types of review on the the same set of OSS project that are quantitatively examined in this work.

OSS developers rarely meet in a synchronous manner, so almost all project communication is recorded [Fielding and Kaiser 1997]. The OSS community fosters a public style of discussion, where anyone subscribed to the mailing list can comment. Discussions are usually conducted on a mailing list as an email *thread*. A thread begins with an email that includes, for example, a question or a patch contribution. As individuals reply, the thread becomes a discussion about a particular topic. If the original message is a patch contribution, then the discussion is a review of that contribution.

We define a review on an OSS project to be an email thread discussion which contains a patch (i.e., a change to the software system). All 25 projects we examined required a patch in order to start a review. The patch contribution may be any kind of software artifact. New artifacts are also patch contributions that contain only added lines. A patch contribution that is posted to a mailing list but that does not receive a response from other developers is the is equivalent to a co-located inspection meeting where only the author shows up to the review meeting—there is no discussion, so it is not a review. Contributions that receive no comments from other developers are excluded from our dataset.

Table II. Project Background Information

| Project | Period | Years | Patches | RTC | $\frac{RTC}{Patches}$ | Commits | CTR | $\frac{CTR}{Commits}$ |
|---|---|---|---|---|---|---|---|---|
| Apache (ap) | 1996–2005 | 9.8 | 4.6K | 3.4K | 74% | 32K | 2.6K | 8% |
| Subversion (svn) | 2003–2008 | 5.6 | 2.9K | 2.8K | 97% | 28K | 2.2K | 8% |
| Linux (lx) | 2005–2008 | 3.5 | 50K | 28K | 56% | 118K | NA | NA |
| FreeBSD (fb) | 1995–2006 | 12 | 73K | 25K | 34% | 385K | 24K | 6% |
| KDE (kde) | 2002–2008 | 5.6 | 22K | 8K | 36% | 565K | 15K | 3% |
| Gnome (gn) | 2002–2007 | 5.8 | 12K | 8K | 67% | 450K | NA | NA |

*Note*: The time period we examined in years, the number of patches submitted, the number of RTC-style reviews, the proportion of patches that have a review discussion, the number of commits, the number of CTR-style reviews, and the proportion of commits that have a review discussion.

*RTC.* For review-then-commit, we identify contributions on the mailing list by examining threads looking for diffs. A diff shows the lines that an author has changed, including lines around the change to help reviewers understand the change [Eggert et al. 2002]. We examine only email threads that contain at least one diff, while in previous work, we considered an email thread to be a review if the email subject contained the keyword "[PATCH]" [Rigby et al. 2008]. This technique works well on the Apache project, as developers usually include the keyword in the subject line; however, other projects do not use this convention. For consistent comparison with the other projects, we re-ran the analysis on Apache, this technique identified an additional 1,236 contributions or an additional 60% of the original sample.

*CTR.* Every time a commit is made, the version control system automatically sends a message to the "version control" mailing list containing the change log and diff. Since the version control system automatically begins each commit email subject with "cvs [or svn] commit:", all replies that contain this subject are reviews of a commit. In this case, the original message in the review thread is a commit recorded in the version control mailing list.

*Extraction Tools and Techniques.* We created scripts to extract the mailing lists and version control data into a database. An email script extracted the mail headers including sender, in-reply-to, and date. The date header was normalized to coordinated universal time (UTC). Once in the database, we threaded messages by following the references and in-reply-to headers. Unfortunately, the references and in-reply-to headers are not required in RFC standards, and many messages did not contain these headers [Resnick et al. 2001]. When these headers are missing, the email thread is broken, resulting in an artificially large number of small threads. To address this problem, we use a heuristic based on the date and subject to join broken threads. For further details of our data extraction techniques, please see Rigby [2011, Appendix B].

### 4.1. Project Background Information

The purpose of Table II is to show that we have sampled a variety of projects and to give a sense of the size of our data sets. Comparisons of raw, absolute values among projects is meaningless. For example, the time periods that we had available for study vary drastically—we studied 12 years of FreeBSD development, but only 3.5 years of Linux development. More meaningful comparisons can be made at the level of individual reviews. When we answer our research questions, individual reviews are our unit of analysis unless otherwise noted.

*Ignored Contributions.* One commonality across projects is that a large number of patch contributions that are ignored. Rigby and Storey [2011] have examined this issue

and found that interviewed developers ignore any patch that is not interesting to them: they ignore a patch rather than rush through a review missing potentially important flaws. An ignored patch may be re-posted at a later time.

The number of ignored patches varies from 66% on FreeBSD to 3% on Subversion. This variation can be partially explained by the culture on the projects. For example, on Subversion, one developer set a community norm by reviewing each patch that was submitted to the mailing list and this culture was adopted by the project [Fogel 2005; Rigby and Storey 2011]. In contrast, interviews with FreeBSD developers indicated that they would only respond to knowledgeable contributors and that a substantial problem for the project's developers was finding someone with the appropriate expertise to review each contribution. KDE and Gnome in particular, have drastically more patch contributions and commits than reviews, indicating that many reviews occur in the bug repository or reviewing tool. Tracker-based review is becoming increasingly popular on projects with a large number of nontechnical users, but it is outside the scope of this article and has recently been examined by other researchers, such as Nurolahzade et al. [2009].

Since OSS developers follow a community norm to only comment when some new insight can be added [Hambridge 1995; Fogel 2005; Rigby and Storey 2011], a response to a commit indicates some potential problem has been found. We see that 8%, 8%, and 6% of commits contain at least one potential problem for Apache, Subversion, and FreeBSD, respectively. Given that we selected successful mature projects for study, it would be surprising to see higher percentages of commits containing potential defects. For KDE, only 3% of changes are reviewed after commit. Since subprojects can vary their review practices, it is clear that some do not use CTR and so none of their changes are reviewed after commit.

Despite the many ignored contributions, this article studies drastically more reviews than any earlier work. For example, Porter et al. [1998, 1997] studied 88 reviews at Lucent and Cohen studied 2.5K peer reviews at Cisco [2006].

*Plotting the Data.* We use two types of plots: beanplots and boxplots. Beanplots show the distribution density along the y-axis and in this work contain a horizontal line that represents the median [Kampstra 2008]. Since most of the distributions in this article are nonnormal, a beanplot allows us to view any irregularities. However, when we have count data that is highly concentrated, we use a boxplot, as a beanplot will appear as one or two concentrated points. For all the boxplots in this work, the bottom and top of the box represent the first and third quartiles, respectively. Each whisker extends 1.5 times the interquartile range. The median is represented by the bold line inside the box. Since our data are not normally distributed, regardless of the style of plot, we report and discuss median values.

## 5. ARCHIVAL DATA RESULTS

We present results related to our research questions. Since each question requires a different set of measures, we describe each measure and discuss its limitations in the section in which it is used. A discussion of the validity of this study can be found in Section 7.2. Although the operationalization of the measures may be unfamiliar to readers, they are designed to mirror those used in traditional inspection experiments (e.g., Fagan [1976] and Porter et al. [1998]).

### 5.1. Frequency and Activity

*Q1.* What is the frequency of review? Can reviewing keep up with development or do we see an increase in the proportion of unreviewed contributions when more commits are being made?

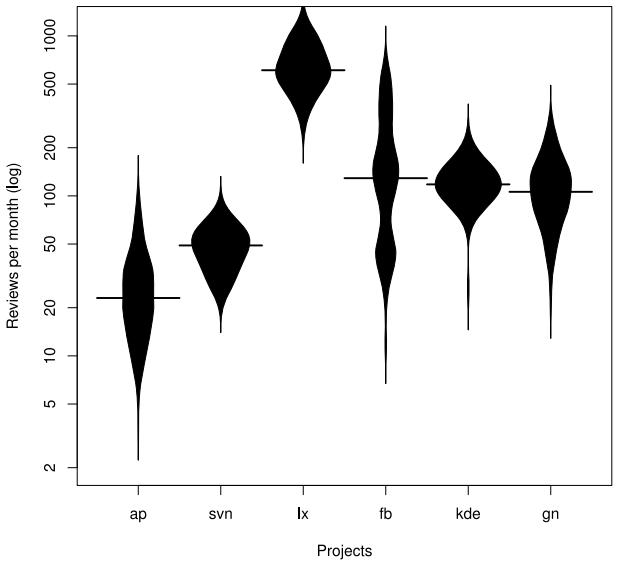


Fig. 2. RTC – reviews per month.

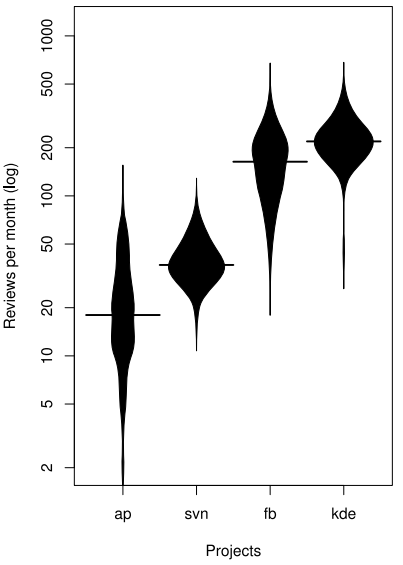


Fig. 3. CTR – reviews per month.

We measure the relationship between development activity and reviews. For this research question, the unit of analysis is the number of reviews per month. For RTC, review frequency is measured by counting the number of contributions submitted to the mailing list that receive at least one reply. For CTR, we count the number of commits that receive at least one reply on the lists. Development activity is measured as the number of commits.

Figures 2 and 3 show the number of reviewers per month for RTC and CTR, respectively. We see that RTC Linux has far more reviews per month (median of 610) than any other project. KDE, FreeBSD, and Gnome all have medians of slightly over 100 reviews per month, while the smaller projects, Apache and SVN, have around 40 reviews in the median case. A similar divide occurs when we look at CTR.

In order to determine the relationship between commit activity and review type, we conduct both Pearson and Spearman correlations.[3] We assume that commit activity is related to development activity [Nagappan and Ball 2005]. The correlation between the number of CTRs and commits is strong. Person $r = .80$, .73, .82 and Spearman $r = .75$, .69, and .84, for Apache, SVN, and FreeBSD, respectively.[4] This correlation indicates that the number of CTRs changes proportionally to the number of commits. Therefore, when there are more changes to be reviewed, there are more CTR reviews. While future work is necessary, this finding suggests that as the number of commits increases, CTR continues to be effective and does not become, as one Apache developer feared, "commit-then-whatever." The correlation for KDE is low ($r = .16$). Since KDE is a large set of related OSS projects, the lack of correlation between CTR and commits may be because not all projects use CTR.

In contrast, the correlation between the number of RTCs and commits is weak to moderate. Only two projects (Linux with $r = .55$ and FreeBSD with $r = .64$, respectively) are correlated above $r = 0.50$ with the number of commits. With RTC, the code is not yet committed, so it cannot cause bugs in the system. This contrasts with CTR,

[3]All correlations are significant at $p \ll .001$.

[4]For SVN, we eliminated one month that had 1.4k commits and only 26 reviews.

which is a review of committed code. As Rigby and Storey found [2011], interviewed developers tend to only review code that interests them and is related to their area of expertise, so many RTC contributions are ignored. Other researchers have provided quantitative evidence on the acceptance rate of RTC. Bird et al. [2007] find that the acceptance rate in three OSS projects is between 25% and 50%. Also on the six projects examined by Asundi and Jayant [2007], they found that 28% to 46% of noncore developers had their patches ignored. Estimates of Bugzilla patch rejection rates on Firefox and Mozilla range from 61% [Jeong et al. 2009] to 76% [Nurolahzade et al. 2009].

The frequency of review is high on all projects and tied to the size of the project. The frequency of CTR has a reasonably strong correlation with the number of commits, indicating that reviewers likely keep up with changes to the system. The correlation between commits and RTC is less strong and may be related to the high levels of ignored RTC contributions, as well as the conservative, self-selecting nature of the developers on the mature projects we examined.

### 5.2. Participation

*Q2.* How many reviewers respond to a review? How much discussion occurs during a review? What is the size of the active reviewer group?

It is simple to count the number of people that come to a co-located inspection meeting. Ascertaining this measure from mailing list-based reviews is significantly more difficult. The first problem is that developers use multiple email addresses. These addresses must be resolved to a single individual. We use Bird et al.'s [2006] name aliasing tool, which calculates the Levenshtein edit distance between normalized pairs of email addresses. The tool is conservative, so we manually inspect and divide the clusters of aliased email addresses that it creates.

The remaining problems relate to the data available for each review type. In mailing list-based reviews, it is only possible to count reviewers who respond to a contribution. So, if an individual performed a review and did not find any issues or found the same issue as other reviewers, this individual would not be recorded as having performed a review. If an individual is performing reviews regularly over a period of time, he or she will eventually be the first person to find an issue and will respond to a contribution (if a reviewer never responds, the reviewer is not helping the software team). We define the active *reviewer group* as all individuals who have participated in a review over a given time period. Since reviews are completed quickly, we define it on a monthly basis (i.e., number of reviewers per month). We have three measures to gauge participation in reviews: the number of developers per review (roughly equivalent to the number of people who actively participate in an inspection—see Figures 4 and 5), the number of emails per review (the amount of discussion per review—see Figures 6 and 7), and the reviewer group or the number of people who performed at least one review in a given month (roughly equivalent to the pool of reviewers who participate in inspections). For each measure, the author of the patch is not counted.

Figure 4 shows that, for RTC, all the projects have, with the exception of Gnome, a median of two reviewers per review. The number of reviewers for CTR is one in the median case (see Figure 5). The median number of messages ranges from three to five for RTC and from two to five for CTR, with RTC having more discussion during review. Despite the large differences in the frequency of reviews across projects, the amount of discussion surrounding a review appears to consistently involve few individuals and have a limited number of exchanges.

Surprisingly, the number of reviewers per month (the reviewer group, which excludes authors) is very similar to the number of reviews per month. The size of the reviewer group ranges from a median of 16 (Apache CTR) to 480 (Linux RTC) reviewers, while the

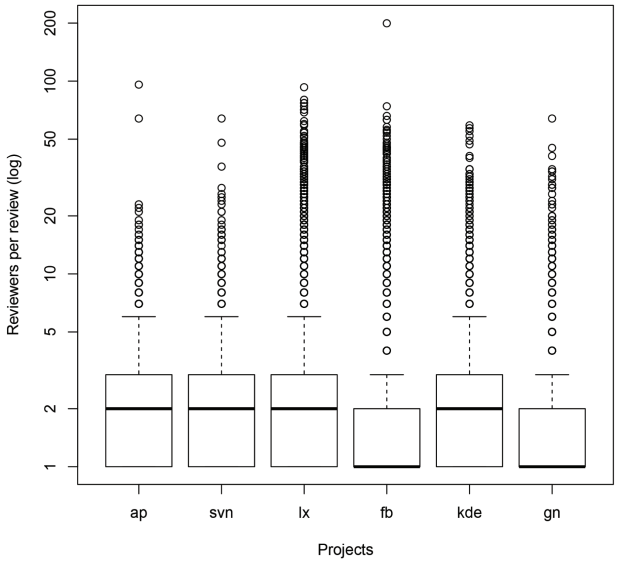


Fig. 4. RTC – reviewers per review.

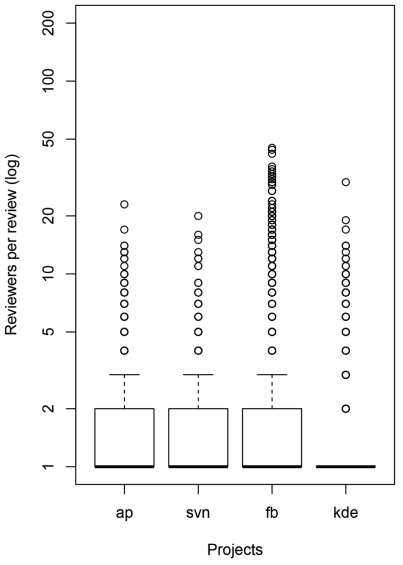


Fig. 5. CTR – reviewers per review.

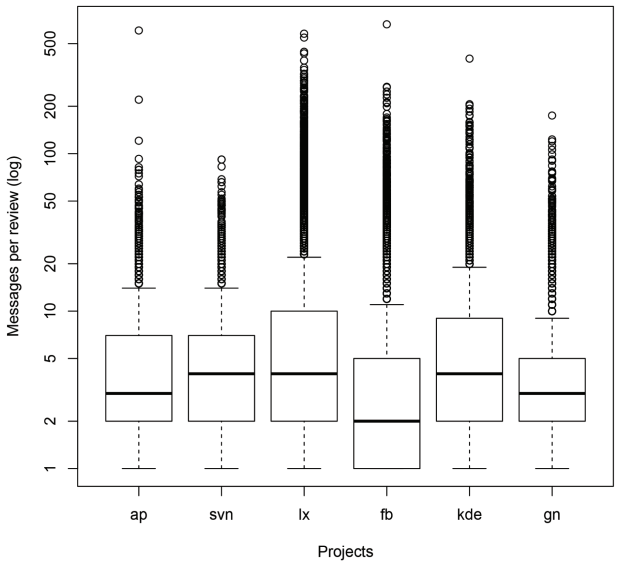


Fig. 6. RTC – messages per review.

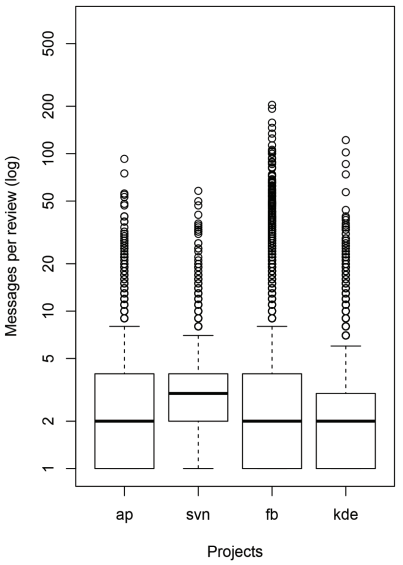


Fig. 7. CTR – messages per review.

number of reviews ranges from 18 (Apache CTR) to 610 (Linux RTC, see Figures 2 and 3) reviews. It is clear from these numbers, and Figures 8 and 9, that most reviewers do not partake in very many reviews. In the median case, a reviewer participates in between two and three reviews per month. However, the top reviewers, that is the reviewers in the 95$^{th}$ percentile, participate in between 13 (Apache CTR) and 36 (Linux RTC) reviews per month. Although 13 Apache CTR reviews may not appear to be many reviews, proportionally, the top reviewers are involved in 72% of all Apache CTR reviews.

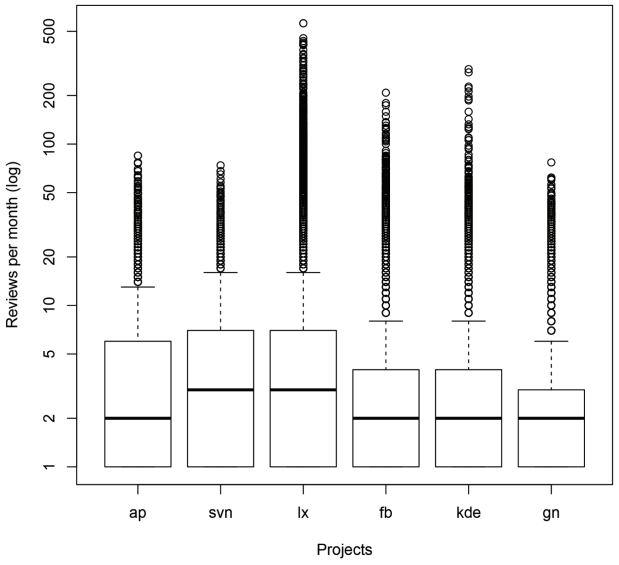


Fig. 8. RTC – number of reviews a developer is involved in per month.

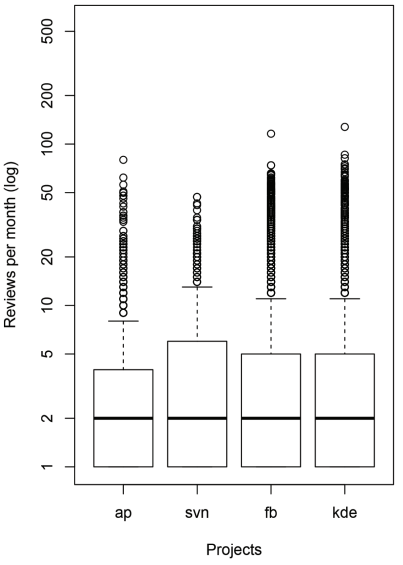


Fig. 9. CTR – number of reviews a developers is involved in per month.

The consistently small number of reviews a developer partakes in likely relates to inherent human limits and developer expertise. Interviewed developers stated that they specialize in a particular aspect of a large system, which in turn limits the number of reviews in which they are able to participate [Rigby and Storey 2011].

Overall, we find that few individuals are involved and few messages are exchanged during each review. Most developers do few reviews per month, and only the top reviewers participate in a large number of reviews per month. However, there is a very large number of developers who participate in only one review in a given month, indicating that a proportionally large number of developers see each review.

### 5.3. Experience and Expertise

*Q3.* For a given review, how long have the authors and reviewers been with the project? How much work has a developer done on the project?

Expertise has long been seen as the most important predictor of review efficacy [Porter et al. 1998; Sauer et al. 2000]. We measure expertise as the amount of work a developer has done. We also measure experience as the length of time a developer has been with the project. In the case of the reviewers, both the average and maximum experience and expertise value is calculated. Since a small group of experts will outperform a larger group of inexperienced reviewers [Sauer et al. 2000], it is important to record the level of experience of the most experienced reviewer. This maximal value will be unaffected by a large number of inexperienced reviewers supported by one experienced reviewer.

*5.3.1. Experience.* The experience of an author or reviewer is calculated as the time between a developer's first message to the mailing list and the time of the current review. Figures 10 and 11 show the distribution of author experience as well as the experience of the top reviewer. For RTC, the median author experience ranges from 275

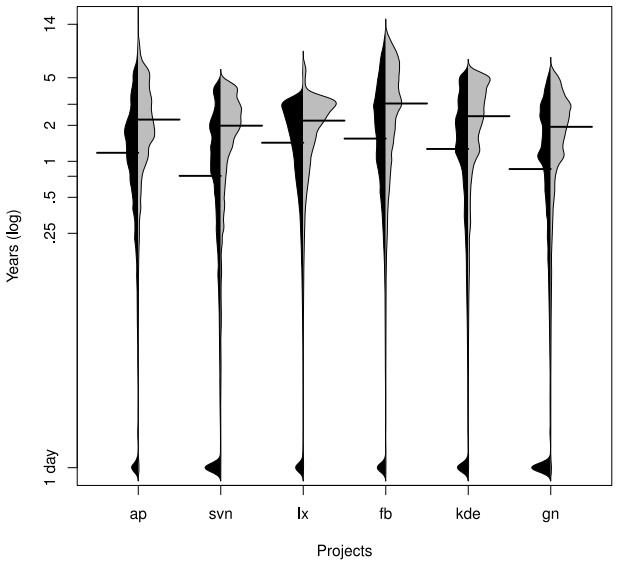


Fig. 10. RTC – author (left) and reviewer (right) experience in years.

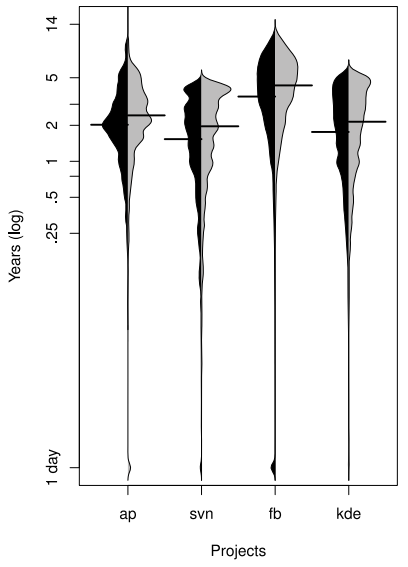


Fig. 11. CTR – author (left) and reviewer (right) experience in years.

to 564 days, and the median top reviewer experience ranges from 708 to 1,110 days.[5] There are also a large number of authors who have made only one contribution, so they have less than one day of experience on the project.[6] For CTR, the median author experience ranges from 558 to 1,269 days, and the median top reviewer experience ranges from 715 to 1,573 days. Given that different time periods were examined for the projects (i.e., Linux = 3.5 years to FreeBSD = 12.0 years), it may not be appropriate to compare across projects. Regardless of time period, both authors and reviewers appear to have been on the mailing lists for extended periods of time, and reviewers typically have more experience than authors. It is also clear that both authors and reviewers for RTC have been with a project for a shorter period of time, and they are likely less experienced than reviewers and authors involved in CTR. This result is not surprising because CTR requires the author to have commit privileges, and experienced core-developers are often required to monitor the commit mailing list, making it more likely that experienced developers are involved in CTR than RTC.

*5.3.2. Expertise.* Mockus and Herbsleb [2002] find evidence that links the amount of work a developer performs in a particular area to his or her level of expertise. They measure the number of commits a developer made to a particular file in the system. We extend and simplify their measure. The amount of "work" done by an author or reviewer is calculated on a monthly basis as the number of review messages, reviewed commits, and unreviewed commits a developer has made before the current review, regardless of area or files modified.[7]

Let the function $work(m, i)$ be the number of times an individual $i$ has made a commit or participated in a review in month $m$. We use the following linear decay function to weight the measure. We divide the life of the project into monthly intervals

[5]In all our measures, the correlation between the average and maximum reviewer is at or above $r = .89$. Due to this high correlation, we use the maximal reviewer in the remainder of this work.
[6]We add one day to our measure of experience to avoid taking the $log(0)$.
[7]In Rigby [2011], the granularity is also calculated at the file level.

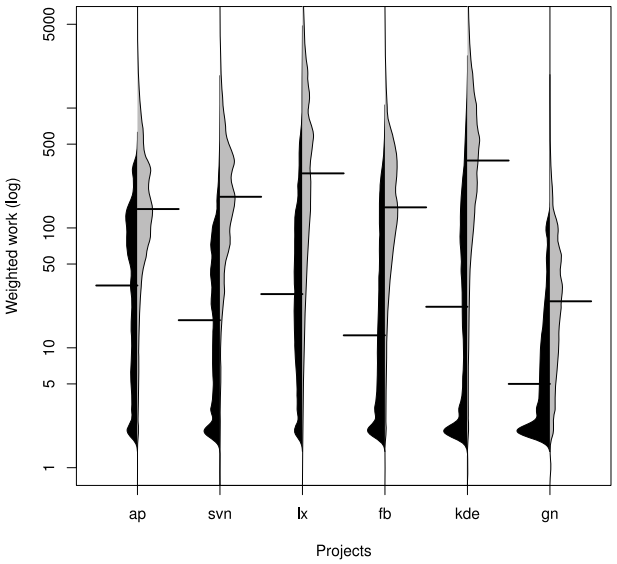


Fig. 12. RTC – author (left) and reviewer (right) weighted work expertise.

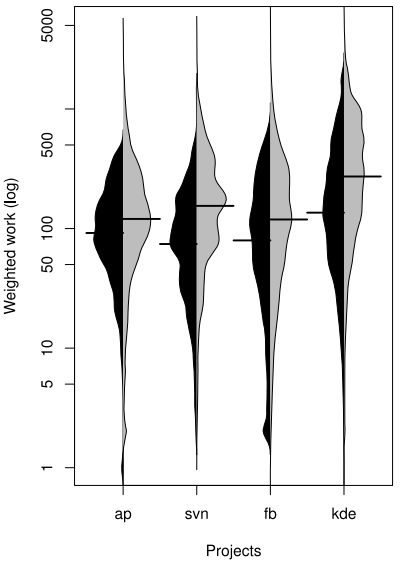


Fig. 13. CTR – author (left) and reviewer (right) weighted work expertise.

and divide the work by the number of months since a given review. So, the weighted work measure is

$$work_w(m,i) = \sum_{w=1}^{m} \frac{work(w,i)}{m-w+1},$$

where $m$ is the calendar month in which the review occurred.

By using this weighting, we factor in the diminishing effect that inactivity and time has on expertise and status within the community. This weighting allows the expertise associated with files developed by one individual but then maintained by another to be appropriately divided. For the remainder of this article, the "work" measure refers to the weighted version.

From Figures 12 and 13, we can see that reviewers have more active work expertise than do authors. Individuals involved in CTR have more expertise than those involved in RTC. The direct interpretation of these findings is difficult because expertise is weighted. A possible simplified interpretation is, in the case of Apache RTC, authors having the equivalent of 111 fewer commits and review messages than do reviewers.

In summary, we find that authors have less experience and work expertise than reviewers and that RTC involves less experienced individuals with less work expertise than CTR.

### 5.4. Churn and Change Complexity

*Q4.* How does the size of change (churn) affect peer review?

The change size or churn of the artifact under review is a common measure in the inspection literature [Laitenberger and DeBaud 2000]. Mockus et al. [2002] found that changes to Apache and Mozilla were smaller than changes to the industrial projects they examined, but they did not explain why this was the case. Software changes are contributed in the form of diffs, which are fragments that indicate what has changed between two versions of a file [Eggert et al. 2002]. We examine the size of changes

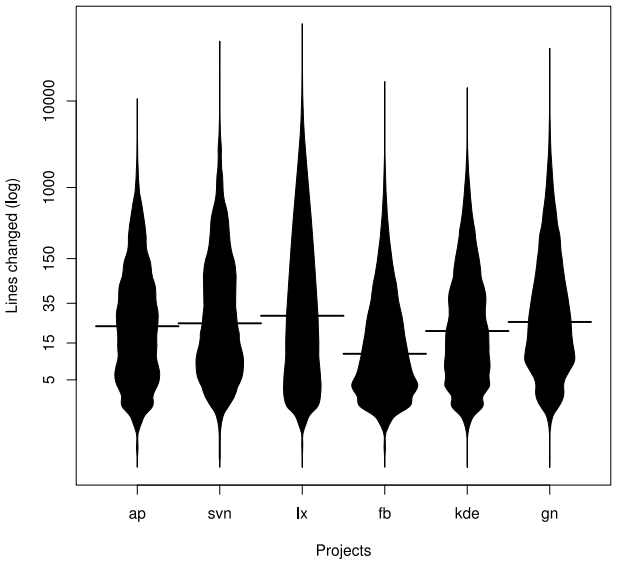


Fig. 14. RTC churn.

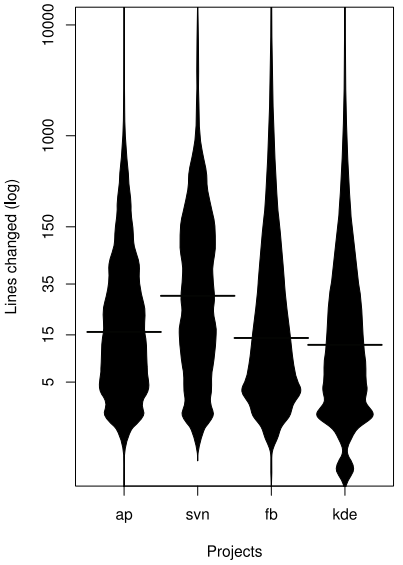


Fig. 15. CTR churn.

that are reviewed. The change size is measured by summing the number of added and deleted lines in a software change or patch.

Figures 14 and 15 show the churn size of reviewed contributions. For RTC, the median churn ranges from 11 to 32 changed lines, while the 75$^{th}$ percentile is between 37 and 174. For CTR, the median churn ranges from 12 to 35 changed lines, and the 75$^{th}$ percentile is between 46 and 137. OSS projects have a policy that requires contributions to be small, independent and complete, so it is not surprising that the change sizes are smaller than those in industrial development. This policy allows developers to periodically and quickly review a large number of contributions [Lussier 2004; Rigby and Storey 2011].

In Rigby's dissertation [2011], he also measured the complexity of a change. The change complexity measures he consider were the following: the number of modified lines (churn), the number of modified files in a change, the number of distinct diffs per review, the distance between contiguous change blocks within a diff, the depth of indentation in a change [Hindle et al. 2008], and the directory distance between files in a diff. Other works have found high correlation between complexity measures, such as McCabe's cyclomatic complexity, and the size of a file. Similarly, Rigby found high correlations between each change complexity measure and the size of the change. This co-linearity would make the interpretation of statistical models difficult. Since churn is the most parsimonious measure of change complexity and it has been used in previous inspection experiments, we use it as a proxy measure of change complexity.

### 5.5. Review Interval

*Q5.* What is the calendar time to perform a review?

Porter et al. [1997] define review interval as the calendar time to perform a review. The full interval begins when the author prepares an artifact for review and ends when all defects associated with that artifact are repaired. The pre-meeting interval, or the time to prepare for the review (i.e., reviewers learning the material under review), is also often measured.

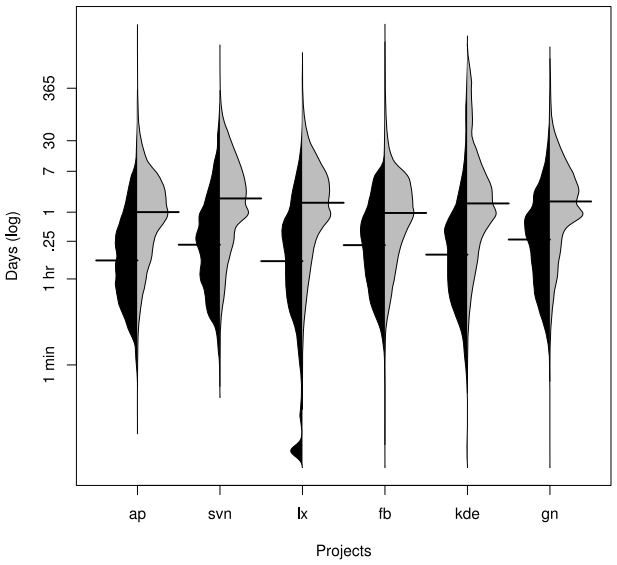


Fig. 16. RTC – First response (left) and full review interval (right) in days.

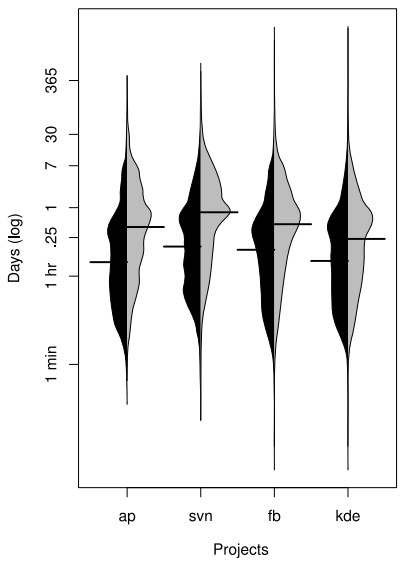


Fig. 17. CTR – First response (left) and full review interval (right) in days.

*RTC.* For every contribution submitted to the mailing list, we determine the difference in time from the first message (the author's diffed contribution) to the final response (the last review or comment). The review interval is short on all projects. For RTC, on the right of Figure 16, we see that the median interval is between 23 and 46 hours, while 75% of contributions are reviewed in 3 to 7 days. We also measure the how long it takes the first reviewer to respond. This time is roughly equivalent to what is traditionally called the pre-meeting time. First-response time, on the left of Figure 16, is between 2.3 and 6.5 hours in the median case and between 12 and 25 hours at the $75^{\text{th}}$ percentile. Only 2% to 5% of RTC reviews begin one week after the initial post, indicating that initially ignored contributions are rarely reviewed.

*CTR.* Since the contribution is committed before it is reviewed, we need to know not only the time between the commit and the last response, the full CTR review interval (on the right of Figure 17), but also the time between the commit and the first response. This first response, shown on the left of Figure 17, indicates when the issue first issue was discovered; the ensuing discussion may occur after the problematic commit is reverted. Ideally, the amount of time defective, unreviewed code is in the system should be short. The first review happens very soon after a commit: 50% of the time it occurs between 1.9 and 4.0 hours, and 75% of the time it happens between 9.4 and 12.8 hours. The discussion or full interval of the review lasts longer, with a median value of between 5.6 and 19.4 hours. In 75% of cases, the review takes less than between 19.8 hours and 2.7 days, depending on the project. Only between 2% and 3% of issues are found after one week has passed.

These data indicate that reviews, discussion, and feedback happen quickly. The following quotation from the Apache mailing list discussion in January 1998 supports these findings.

> "I think the people doing the bulk of the committing appear very aware of what the others are committing. I've seen enough cases of hard to spot typos being pointed out within hours of a commit." [Hartill 1998]

### 5.6. Issues and Defects

*Q6.* How many issues are discussed during a review?

Traditionally, the number of defects found during inspection, or the inspection effectiveness, was manually recording during inspection meetings [Fagan 1976]. OSS developers do not record the number of defects found in a review. Interviews with OSS developer indicate that they are interested in discussing a solution to defects and implementing the solution, rather than simply identifying defects [Rigby and Storey 2011]. The transition from inspection as purely a defect finding activity to reviews that include group problem solving has been discussed in Rigby et al. [2012]. Our goal in this section is to identify a proxy measure for review effectiveness. Next, we describe a measure of the number of issues discussed during review. A count of issues discussed will include false positives, such as reviewer questions that do not lead to a defect being found. The measure is a count of the number of times diffed code is interleaved with reviewer comments and is based on norms in the OSS community.

In the following Subversion review, we see that the reviewer has posted two issues (potential defects) under the diffed code which contains the issue. This practice is known as 'bottom-posting' and has been considered a best practice on email forums since the early days of Usenet [Jar 2013]. As part of Rigby and Storey's [2011] grounded theory study of 500 review threads on the same OSS projects we examine here, they found bottom-posting to be an enforced, universal practice on OSS projects.

```
> +  if (strcmp(status, "failure") == 0)
> +    return svn_ra_svn__handle_failure_status(list, pool);
> +
What happens to err here. If an error was returned, status is garbage.

> +  if (err && !SVN_ERR_IS_LOCK_ERROR(err))
> +    return err;
> +
Parsing a tuple can never result in a locking error, so the above is
bogus.
```

As a reviewer finds issues with the code, he or she writes the rationale under the section of code that is problematic, removing sections that are unrelated to the issue. This pattern may happen many times during a single email, indicating multiple issues with the code. To create a proxy measure of the number of issues discussed, we count the number of times the replied-to text of a message containing a diff (i.e., lines in a diff starting with '>') are broken by new text (i.e., lines that do not start with '>'). Each time this break occurs, we consider the break to be an issue that the reviewer has found. In the preceding example, there would be two issues.

Responses to reviewer comments constitute a discussion of an existing issue and not a new issue, so responses to responses (i.e., when the diff has two or more '>') are not counted as new issues. In the following example, we can see the author responding to acknowledge that the reviewer is correct. These responses are not counted as two new issues.

```
> > +  if (strcmp(status, "failure") == 0)
> > +    return svn_ra_svn__handle_failure_status(list, pool);
> > +
> What happens to err here. If an error was returned, status is garbage.

Oops.
```

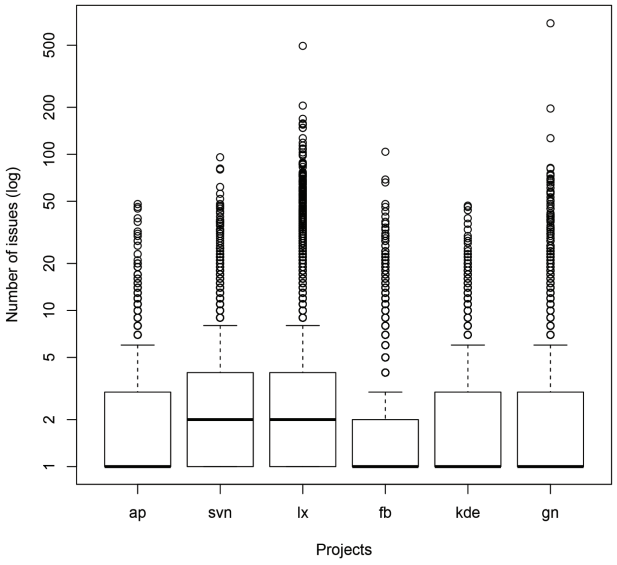


Fig. 18. RTC – number of Issues.

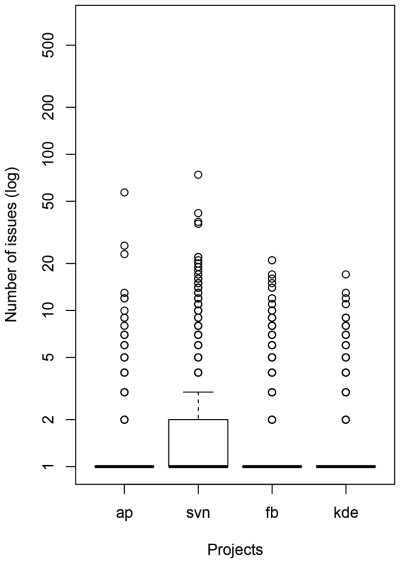


Fig. 19. CTR – number of issues.

```
> > +  if (err && !SVN_ERR_IS_LOCK_ERROR(err))
> > +    return err;
> > +
> Parsing a tuple can never result in a locking error, so the above is
> bogus.

Double oops.
```

For RTC, Figure 18 shows the number of issues discussed in a review is one or two in the median case and between two and four at the 75$^{th}$ percentile. For CTR, Figure 19 shows the number of issues is one in the median case and between one and two at the 75$^{th}$ percentile. Few issues are discussed per review. Patch contributions that do not receive a response have no discussion and are not considered reviews (See Section 4). Since each patch contribution must receive at least one response to be considered a review, the minimum number of issues per review is one.

In summary, we do not have the traditional measure of number of defects found per review. However, interviews with OSS developers have indicated that they are less interested in counting defects and more interested in discussing solutions to issues found in the code. We have developed a proxy measure of the *number of issues discussed*. We see that a small number of issues are being discussed per review. Since the size of contributions is small (between 11 and 32 lines changed, see Section 5.4), we would not expect there to be a large number of issues found per review. OSS projects appear to frequently and incrementally review small sections of code, rather than inspect large work products after they are completed.

### 5.7. Summary of Quantitative Results

The goal of this section was to quantify and establish the parameters of peer review in OSS. We summarize our findings here and in Table III.

Table III. Summary of Quantitative Results

| Research Question | Measures | Min | Max |
|---|---|---|---|
| **Q1: Frequency** – Section 5.1 | | | |
| | Number of reviews per month | 18 | 610 |
| **Q2: Participation** – Section 5.2 | | | |
| | Number of reviewers per review | 1 | 2 |
| | Number of messages per review | 3 | 5 |
| | Number of reviewers per month | 16 | 480 |
| **Q3: Experience and Expertise** – Section 5.3 | | | |
| | Author experience RTC (days) | 275 | 564 |
| | Top review experience RTC (days) | 708 | 1110 |
| | Author experience CTR (days) | 558 | 1269 |
| | Top review experience CTR (days) | 715 | 1573 |
| | Units of expertise (weighted amount of work) do not have easily interpretable units (see Section) 5.3.2. | | |
| **Q4: Churn and Change Complexity** – Sections 5.4 | | | |
| | Patch change size in lines (churn) | 11 | 32 |
| **Q5: Interval (efficiency)** – Section 5.5 | | | |
| | First response RTC (hours) | 2 | 3 |
| | Full interval RTC (hours) | 23 | 46 |
| | First response CTR (hours) | 2 | 4.0 |
| | Full interval CTR (hours) | 6 | 19 |
| **Q6: Issues (effectiveness)** – Section 5.6 | | | |
| | Number of issues discussed | 1 | 2 |

*Note*: The median value for a project is represented in each case. For example, Apache CTR has a median of 18 reviews per month, while Linux has a median of 610 reviews per month.

*Q1 Frequency.* While all projects conducted frequent reviews, the number of reviews varies with the size of the project. Reviews increase with the number of commits (CTR), but not necessarily with the number of patch submissions (RTC).

*Q2 Participation.* There are hundreds of stakeholders subscribed to the mailing lists that contains the contributions for review; however, there is a small number of reviewers and messages per review. The size of the reviewer group varies with the size of project. With the exception of core developers, most reviewers participate in few reviews.

*Q3 Experience and Expertise.* Authors tend to have less experience and expertise than reviewers, and individuals involved in CTR have more experience and expertise than those in RTC. Developers involved in review usually have at least one year of experience, and in the case of core reviews, usually multiple years.

*Q4 Churn and Change Complexity.* The changes made to patches are small, which likely makes providing feedback quicker. The size of a change (churn) is a simple measure of change complexity [Rigby 2011].

*Q5 Interval (Efficiency).* The review interval is very short on OSS projects. Facilitated by the small size of a change, reviews are conducted very frequently and quickly. For CTR, where code is already committed, the first response indicates the time it takes to find the initial defect. Very few reviews last longer than one week.

*Q6 Issues (Effectiveness).* There are few issues discussed per review. This finding is consistent with small change sizes and high frequency of review.

## 6. MODELS OF EFFICIENCY AND EFFECTIVENESS

The goal of a software process is to facilitate the production of high-quality software in a timely manner. Review techniques have historically been compared and statistically modeled based on their effectiveness (e.g., how many issues or defects they find) and efficiency (e.g., how long it takes to find and remove those issues) [Kollanus and Koskinen 2009; Porter et al. 1998]. We use statistical models to determine the impact that each of our explanatory variables has on the review interval and number of issues found in review. We also isolate the impact of the review process (i.e., RTC or CTR) and the project.

For efficiency (see Section 6.1), measured by the review interval, we use a multiple linear regression model with constant variance and normal errors [Crawley 2005, p. 120]. Our model of efficiency explains 29% of the variance. For effectiveness (see Section 6.2), measured by the number of issues found, we use a generalized linear model (GLM) with Poisson errors and a logarithmic link function. A quasi-Poisson model and a pseudo R-squared measure is used to correct for over-dispersion [Heinzl and Mittlböck 2003; Maindonald and Braun 2003, p. 219]. Our model of effectiveness explains 51% of the deviance. Explanatory variables shown in the previous sections to be correlated with other variables in the model (e.g., number of replies and reviewers) were replaced with the most parsimonious variable (i.e., the measure that is easiest to calculate). We grouped our explanatory variables into four categories: expertise, experience, complexity, and participation. We applied a log transformation to explanatory variables that were shown (in previous sections) to be right skewed with a long tail. To represent our models, we use the R language notation [R Development Core Team 2011]. For example, the formula $y \sim a + b * c$ means that "the response y is modeled by explanatory variables a, b, c and the interaction between b and c".

### 6.1. Efficiency: Review Interval

Our multiple linear regression model of efficiency, review interval, has the following form.[8]

$$\begin{aligned}\log(\text{interval}) \sim\ & \log(\text{auth experience} + 1) + \log(\text{auth expertise} + 1) \\ & + \log(\text{rev experience} + 1) + \log(\text{churn} + 1) \\ & + \log(\text{reviewers}) + \text{review type} * \text{project}.\end{aligned}$$

We used the Akaike Information Criterion (AIC) for model selection [Burnham and Anderson 2002]. AIC for linear regression models is defined as $nlog(RSS/n) + 2k$, where n is the sample size, $RSS$ is the residual sums of squares, and $k$ is the number of parameters in the model [Burnham and Anderson 2002, p. 63]. AIC differences are defined as $\Delta_i = AIC_i - AIC_{min}$ over all models in the candidate set. From the set of candidate models, the model with the lowest AIC or $\Delta_i = 0$ is selected as the best model [Burnham and Anderson 2002, p. 71]. The actual value of AIC is meaningless, as it is a relative measure. AIC favours the model with the highest likelihood but penalizes models with more parameters. Model weights can be used to evaluate the relative likelihood of a model within a candidate set. Akaike weights are defined as

$$w_i = \frac{exp(-1/2\Delta_i)}{\sum_{r=1}^{R} exp(-1/2\Delta_r)},$$

where $R$ is the number of models in the candidate set. Note that $\sum_{i=1}^{R} w_i = 1$. Finally, it may be that several models are plausible in that there is not an obvious best model in

[8]To avoid taking the logarithm of zero, we add one to some variables.

Table IV. Model Selection Results for the Set of Candidate Models

| Models | $k$ | $R^2$ | AIC | $\Delta$ AIC | AIC weights |
|---|---|---|---|---|---|
| 1 | 15 | 0.29 | 477854.48 | 0.00 | 1.00 |
| 2 | 16 | 0.29 | 477856.47 | 1.99 | 0.37 |
| 3 | 8 | 0.28 | 478464.10 | 609.62 | 0.00 |
| 4 | 7 | 0.28 | 479112.28 | 1257.80 | 0.00 |
| 5 | 7 | 0.26 | 482534.22 | 4679.74 | 0.00 |
| 6 | 6 | 0.24 | 484759.53 | 6905.05 | 0.00 |

1: author experience, author expertise, reviewer experience, churn, reviewers, review type, project, review type $\times$ project
2: Model 1 including reviewer expertise
3: Model 1 excluding project by review type interaction
4: Model 3 excluding project
5: Model 3 excluding review type
6: Model 4 excluding review type

*Note*: The number of parameters in the model ($k$), $R^2$, the Akaike Information Criterion (AIC), $\Delta$ AIC, and AIC weights are provided. Models within the candidate set are defined by the explanatory variables included in the model.

the candidate set. In this instance, model averaging can be employed, where parameter estimates are obtained by averaging over the plausible models (see Burnham and Anderson [2002] for details).

Table IV shows the model selection results that lead to the chosen model previously shown. From this process, it is clear that we must include the project and review type in our model, but that reviewer expertise is not necessary. The resulting model explains the most variance, 29%, while minimizing the number of explanatory variables. The amount of variance explained is low but consistent with other models of review interval 32% [Porter et al. 1997] and 25% [Porter et al. 1998]. Additional factors that may be outside of the development process and product appear to influence the review interval. The diagnostic plots confirmed that the residuals were normally distributed and that the variance was constant.

Since both the response and explanatory variables are on a log scale, to interpret the results, we must examine rates of change. Table V shows the change in review interval for a 10%, 50%, 100%, and 200% change in the explanatory variable. For example, a 10% increase in the number of lines changed (i.e., churn) leads to a 1.2% increase in the review interval. Interpreting these proportions can be difficult. To provide a more intuitive understanding of the effect each explanatory variable has on the review interval, we describe the change that occurs in the median case for a given explanatory variable assuming that all other variables remain equal.[9]

*Author.* An addition of the equivalent of four new messages, commits, or reviews in the current month leads to a decrease of nine minutes during review. The more active an author is, the shorter the review interval, likely because the author and his or her changes are known to the reviewers and preparation time is reduced. A developer who has been on the project for an additional 60 days increases the review interval by two minutes. The longer a developer is with the project, the more complex his or her changes will be, which will likely increase the interval. The overall effect of author expertise and experience on review interval is minimal.

[9]A pairwise correlation analysis among explanatory variables indicated that the highest correlation was between author experience and author expertise at $r = .43$.

Table V. Effect of Expertise, Experience, Complexity, and Participation on the Review Interval

| | Variable | Estimate | Std. Error | 10% | 50% | 100% | 200% |
|---|---|---|---|---|---|---|---|
| Experience | log(auth experience + 1) | 0.01 | 0.004 | 0.13 | 0.53 | 0.92 | 1.46 |
| & | log(auth expertise + 1) | −0.08 | 0.004 | −0.73 | −3.06 | −5.18 | −8.09 |
| Expertise | log(rev experience + 1) | 0.13 | 0.006 | 1.23 | 5.32 | 9.27 | 15.09 |
| Complexity | log(churn + 1) | 0.13 | 0.003 | 1.20 | 5.21 | 9.07 | 14.76 |
| Participation | log(reviewers) | 1.26 | 0.009 | 12.76 | 66.66 | 139.45 | 299.06 |

*Note*: To make interpretation easier, the proportional change at 10% to 200% is shown for each variable. For example, a doubling in the number of reviewers leads to an increase of 140% in the review interval.

Table VI. The Effect of Project and Review Type on Review Interval

| | Estimate | Std. Error | Percent Difference |
|---|---|---|---|
| KDE – CTR | −0.68 | 0.04 | −49.51 |
| FreeBSD – CTR | −0.50 | 0.04 | −39.66 |
| Apache – CTR | −0.46 | 0.05 | −36.76 |
| Subversion – CTR | 0.15 | 0.05 | 15.67 |
| FreeBSD – RTC | 0.26 | 0.03 | 29.57 |
| Linux – RTC | 0.47 | 0.03 | 60.66 |
| Gnome – RTC | 0.79 | 0.04 | 119.28 |
| Subversion – RTC | 0.81 | 0.05 | 125.01 |
| KDE – RTC | 0.84 | 0.04 | 130.58 |

*Note*: The percentage difference of the review intervals is calculated for each project and review type with respect to Apache RTC. For example, the interval for FreeBSD CTR is 40% shorter than for Apache RTC, while Linux RTC is 61% longer.

*Reviewer.* The reviewer's active work expertise did not make a statistically significant contribution to the model and was dropped during model selection (see Table IV, Model 2). Reviewer experience indicated that an increase of 1/4 (or 95 days) with the project increases the review interval by 15 minutes. Reviewer experience has more impact than author experience on the review interval.

*Churn.* Doubling the number of lines changed from 17 to 34 increases the review interval by 1.8 hours. The size of the artifact has a large impact on review interval with small additions of code increasing the review interval.

*Participation.* The addition of one reviewer increases the review interval by 28 hours. The level of reviewer participation has the largest impact on review interval.

*Project and Review Type.* Having discussed the effect of the main variables, we next discuss the impact of review type and project, which are represented as dummy variables in our model. Since we take the log of the review interval as our response variable, we must use Kennedy's [1981] estimator to interpret the percentage change for each project and review type. Table VI shows the affect of the project and review type variable calculated with respect to Apache RTC. We can see that KDE CTR has the shortest review interval and is 50% shorter than Apache RTC, while KDE RTC has the longest review interval and is 131% longer than Apache RTC. Within all projects, the CTR interval is shorter than the RTC interval. CTR is 37, 109, 69, and 180 percentage points shorter than RTC for Apache, SVN, FreeBSD, and KDE, respectively.

*Summary.* The efficiency of review or the review interval is most effected by the level of participation from the community. The size of the change has the next greatest impact, followed by the reviewer's experience and the author's experience and expertise.

The impact of the reviewer's expertise is negligible and is excluded from the model. Within all projects, CTR is faster than RTC.

### 6.2. Effectiveness: Issues Discovered

We use the following log-linear model for effectiveness (number of issues found).[10]

$$\begin{aligned}\text{issues} \sim\ & \log(\text{auth experience}+1)+\log(\text{auth expertise}+1)+\log(\text{rev experience}+1)\\ & +\log(\text{rev expertise}+1)+\log(\text{churn}+1)+\log(\text{reviewers})\\ & +\text{review type}*\text{project}.\end{aligned}$$

Reviews that have no discussion are not considered in this work, so the model only characterizes contributions that have issues. We found over-dispersion in our data which occurs when there is more variability in the data than explained by the model. This will make explanatory variables look more statistically significant than they actually are [Heinzl and Mittlböck 2003], as the estimated variances are smaller than they should be. To deal with this problem, we estimate an over-dispersion parameter, $\hat{c}$, that is used to inflate the variance by adjusting standard errors by a factor of $\sqrt{\hat{c}}$ [McCullagh and Nelder 1989, p. 127]. The Pearson-based dispersion parameter is estimated using

$$\hat{c}=\frac{\chi^2}{(n-k-1)}=\sum_i\frac{\frac{(y_i-\hat{\mu}_i)^2}{\hat{\mu}}}{(n-k-1)},$$

where $k$ is the number of parameters in the saturated model and n is the sample size. This is the sum of the squared differences between observed and expected counts divided by the expected count.

In R, this technique is achieved by running a quasi-Poisson general linear model [Maindonald and Braun 2003, pp. 213–216]. Although this technique deals with the dispersion issue, it does not provide an R-squared measure (or measure of explained variation) that is adjusted for over-dispersion. Heinzl and Mittlböck [2003] evaluate a series of deviance-based pseudo R-squared measures for Poisson models with over- and under-dispersion. We implemented their most robust adjusted R-squared measure in R. The formula is shown here:

$$R^2=1-\frac{D(y:\hat{\mu})+(k-1)\hat{c}}{D(y:\bar{\mu})},$$

where $D(y:\hat{\mu})$ is the deviance of the model of interest and $D(y:\bar{\mu})$ is the deviance of the null or intercept-only model.

Our model selection process is similar to that of Section 6.1. Since we have over-dispersion, we replace AIC with a QAIC defined as $-[2logL(\hat{\beta})/\hat{c}]+2K$, where $K$ is the number of parameters in the model plus 1 to account for the over-dispersion parameter $\hat{c}$ and $L(\hat{\beta})$ is the likelihood evaluated at the maximum likelihood estimates for the model of interested [Burnham and Anderson 2002, p. 68–70]. Definitions for $\Delta$QAIC and QAIC weights simply replace AIC with QAIC.

Table VII contains the model selection results that lead to the chosen model shown previously. The model selection process makes it clear that we must keep the review type, project, and review type by project interaction. Furthermore, unlike the efficiency model, the reviewer's expertise variable contributes to this model. The resulting model is a reasonable representation of peer review effectiveness with a pseudo $R^2=.51$.

[10]To avoid taking the logarithm of zero, we add one to some variables.

Table VII. Model Selection Results for the Set of Candidate Models

| Models | $K$ | pseudo $R^2$ | QAIC | $\Delta$ QAIC | QAIC weights |
|---|---|---|---|---|---|
| 1 | 17 | 0.51 | 183082.36 | 0.00 | 1.00 |
| 2 | 10 | 0.51 | 183209.20 | 126.84 | 0.00 |
| 3 | 9 | 0.50 | 184978.05 | 1895.69 | 0.00 |
| 4 | 9 | 0.49 | 186582.38 | 3500.02 | 0.00 |
| 5 | 8 | 0.45 | 191972.32 | 8889.96 | 0.00 |

1: author experience, author expertise, reviewer experience, reviewer expertise, churn, reviewers, review type, project, project $\times$ review type
2: Model 1 excluding project by review type interaction
3: Model 2 excluding review type
4: Model 2 excluding project
5: Model 2 excluding project and review type

*Note*: The number of parameters in the model plus 1 ($K$), pseudo $R^2$, the Quasi Akaike Information Criterion (QAIC), $\Delta$QAIC, and QAIC weights are provided. Models within the candidate set are defined by the explanatory variables included in the model. The estimate of $\hat{c} = 2.4$ for the saturated model (model 1).

Table VIII. Effect of Expertise, Experience, Complexity, and Participation on the Number of Issues Discussed

| | Variable | Estimate | Std. Error | 10% | 50% | 100% | 200% |
|---|---|---|---|---|---|---|---|
| Experience | log(auth experience + 1) | 0.01 | 0.002 | 0.10 | 0.40 | 0.70 | 1.10 |
| | log(rev experience + 1) | 0.03 | 0.003 | 0.30 | 1.20 | 2.00 | 3.20 |
| Expertise | log(auth expertise + 1) | −0.03 | 0.002 | −0.20 | −1.00 | −1.80 | −2.80 |
| | log(rev expertise + 1) | 0.04 | 0.002 | 0.40 | 1.60 | 2.80 | 4.50 |
| Complexity | log(churn + 1) | 0.17 | 0.002 | 1.60 | 7.00 | 12.20 | 20.10 |
| Participation | log(reviewers) | 0.54 | 0.004 | 5.30 | 24.60 | 45.60 | 81.40 |

*Note*: The quasi-Poisson dispersion parameter is taken to be 2.4 (over-dispersion). To make interpretation easier the proportional change at 10% to 200% is shown for each variable. For example, a doubling in the number of reviewers leads to an increase of 45.6% in the number of issues discussed.

Since we are using a quasi-Poisson model, which has a log-link function and our explanatory variables are on a log scale, to interpret the results we must examine rates of change. Table VIII shows the change in the number of issues discussed for a 10%, 50%, 100%, and 200% change in the explanatory variable. For example, a 10% increase in the number of lines changed (i.e., churn) leads to a 1.6% increase in the number of issues discussed during review.

*Experience and Expertise.* Changes in both author review experience and expertise have minimal impact on the number of issues found (see Table VIII). For example, a 10% increase in the author's expertise, that is, 80 additional messages, commits, or reviews in a given month leads to only a 0.4% increase in the number of issues found.

*Churn.* Doubling the number of lines changed from 17 to 34 increases the number of issues discussed by 12%. Small additions of code increase the number of issues discussed.

*Participation.* The addition of one reviewer increases the number of issues discussed by 46%. The level of review participation has the largest impact on the number of issues discussed.

*Project and Review Type.* As with the previous model, the impact of review type and project are represented as dummy variables in the current model. Since a quasi-Poisson has a log-link function, we must use Kennedy's [1981] estimator to interpret the percentage change of each project and review type. Table IX contains the effect of the project and review type variable calculated with respect to Apache RTC. The table

Table IX. Effect of Project and Review Type on the Number of Issues Discussed

| | Estimate | Std. Error | Percent Difference |
|---|---|---|---|
| FreeBSD – CTR | −0.49 | 0.02 | −38.57 |
| Apache – CTR | −0.30 | 0.03 | −25.71 |
| KDE – CTR | −0.28 | 0.02 | −26.31 |
| KDE – RTC | −0.05 | 0.02 | −4.71 |
| FreeBSD – RTC | −0.02 | 0.02 | −2.35 |
| Subversion – CTR | 0.00 | 0.03 | 0.21 |
| Linux – RTC | 0.35 | 0.02 | 41.48 |
| Subversion – RTC | 0.43 | 0.02 | 54.20 |
| Gnome – RTC | 0.49 | 0.02 | 63.39 |

*Note*: The percentage difference of the number of issues discussed is calculated for each project and review type with respect to Apache RTC. For example, the issues discussed for FreeBSD CTR is 39% fewer than for Apache RTC, while Gnome RTC is 63% more.

shows that FreeBSD CTR has the fewest issues discussed, and Gnome RTC has the most issues discussed, with 39% fewer and 63% more issues discussed than Apache RTC, respectively. Within all projects, the number of issues discussed during CTR is less than during RTC. During CTR, there are 22, 25, 36, and 54 percentage points fewer issues discussed than for RTC on KDE, Apache, FreeBSD and Subversion, respectively.

*Summary.* Like efficiency, the effectiveness of the review is most affected by the level of participation from the community. The size of the changes also impacts the number of issues discussed during review. However, while the effects of author and reviewer experience and expertise are statistically significant, the magnitude of the influence is small. There are fewer issues discussed during CTR than during RTC. In the next section, we combine our findings and the literature to provide a theory of OSS peer review.

## 7. DISCUSSION

Based on our findings across multiple case studies, we extracted the essential attributes of OSS peer review into a theory. We have provided quantitative evidence related to the following research questions:

—the review process, policies, and project structure,
—the frequency of review, and the relationship between the frequency of review and development activity,
—the level of participation in reviews and the size of the reviewer group,
—the level of experience and expertise of authors and reviewers,
—the size and complexity of the artifact under review,
—the review interval (that is, the calendar time to perform a review), and
—the number of issues found per review.

The following statement encapsulates our understanding of how OSS review functions. OSS peer review involves (1) early, frequent reviews (2) of small, independent, complete contributions (3) that are broadcast to a large group of stakeholders, but only reviewed by a small set of self-selected experts, (4) resulting in an efficient and effective peer review technique. We dissect this statement next, showing the evidence that we have gathered. In the subsequent section, we contrast our finding with those in the inspection literature.

(1) *Early, frequent reviews*
The longer a defect remains in an artifact, the more embedded it will become and the more it will cost to fix. This rationale is at the core of the 35-year-old Fagan inspection technique [1976]. We have seen comparatively high review frequencies for all OSS projects in Section 5.1. Indeed, the frequencies are so high that we consider OSS review as a form of "continuous asynchronous review". We have also seen a short interval that indicates quick feedback (see Section 5.5). This frequent review and feedback also incorporates re-review of changes within the same discussion thread and resembles a form of "asynchronous pair programming" [Rigby and Storey 2011].

(2) *of small, independent, complete contributions*
Mockus et al. [2000] were the first to note that OSS projects have smaller change sizes than those seen on industrial projects [Dinh-Trong and Bieman 2005; Weissgerber et al. 2008]. Section 5.4 illustrates the small contribution sizes on the OSS projects we studied (between 11 and 32 lines of code modified).

We believe that small change size is essential to the divide-and-conquer style of peer review found on OSS projects. This strategy eases review by dividing the complexity of a bug fix or feature across multiple changes. Small changes sizes lead to incremental and frequent reviews. The review processes in Section 3 and interviews with core developers done by Rigby and Storey [2011] provide support for the idea that OSS developers will review only small, independent, complete contributions.

(3) *that are broadcast to a large group of stakeholders, but only reviewed by a small set of self-selected experts*
The mailing list broadcasts contributions to a potentially large group of individuals. A smaller group of reviewers conducts reviews periodically. With the exception of the top reviewers, most individuals participate in very few reviews per month. Between one and two reviewers participate in any given review (see Section 5.2). OSS developers have adopted lightweight practices that allow them to find interesting reviews in the barrage of information [Rigby and Storey 2011]. There are, however, a large number of contributions that are ignored and never reviewed and committed.

Expertise is a stronger predictor of review effectiveness than process measures [Porter et al. 1998]. As we discuss in Section 5.3, both authors and reviewers have substantial expertise and experience with reviewers typically having more experience than authors. OSS projects have stripped the process to a minimum, keeping only essential aspect—experts must review each change.

(4) *leads to an efficient and effective peer review technique.*
The review interval in OSS is on the order of a few hours to a couple of, making peer review in OSS very efficient. While it is difficult to model review interval (see Section 6.1), this interval is drastically shorter than that of traditional inspection, which is on the order of weeks [Sauer et al. 2000]. The strongest predictor of interval is the number of reviewers. Most reviews can be dealt with by two reviewers; however, since the review is broadcast, more complex and controversial issues can involve a large number of reviewers (see the outliers in Section 5.2).

The number of defects found per KLOC is not recorded on OSS projects. It is difficult to estimate this measure because it is unclear which contributions are ignored and which are reviewed without defects found. Instead, we find in the median case that there are between one and two issues discussed per review (see Section 6.2). Like efficiency, the best predictor of the number of issues found is the attention a contribution is given by the developer community. The high proportion of ignored contributions [Bird et al. 2007; Asundi and Jayant 2007]

indicates that on large, successful projects, attention from the community is the most sought-after resource [Rigby and Storey 2011].

### 7.1. Comparison with Inspection

There are few similarities between OSS review and Fagan inspections [Fagan 1976]. However, many of the advances and research findings over the last 35 years for software inspections are reflected by OSS peer review practices. In some cases, OSS review can be seen as taking an inspection finding to the extreme. In this section, we compare the inspection findings to our OSS review findings on the style of review meeting, formality of process, and the artifact and under review.

Fagan inspection is a formal process that relies on rigid roles and steps, with the single goal of finding defects [Fagan 2002]. Most of the work on inspection processes has made minor variations to Fagan's process (e.g., [Martin and Tsai 1990; Knight and Myers 1993]) but kept most of the formality, measurability, and rigidity intact [Kollanus and Koskinen 2009; Laitenberger and DeBaud 2000; Wiegers 2001].

*Inspection Meetings.* An important exception to this trend was the work done by Votta that facilitated *asynchronous inspection*. In contrast to Fagan-style inspections, where defect detection is performed only during the meeting phase, Votta [1993] showed that almost all defects can be found during the individual preparation phase, where a reviewer prepares for an inspection meeting by thoroughly studying the portion of code that will be discussed. Not only were few additional defects found during synchronous meetings, but scheduling these meetings accounted for 20% of the inspection interval. This result allowed researchers to challenge the necessity of meetings. Eick et al. [1992] found that 90% of the defects in a work product could be found in the preparation phase. Perry et al. [2002] conducted a controlled experiment in an industrial setting to test the necessity of a meeting and found that meeting-less inspections, according to the style of Votta, discovered the same number of defects as synchronous Fagan inspections. A similar result was obtained by Johnson and Tjahjono [1998]. There are many tools to support inspection in an asynchronous environment (e.g., [Mashayekhi et al. 1994; Macdonald and Miller 1999; Cohen 2006]). Mirroring these findings, OSS reviews are conducted in an asynchronous setting.

*Defect Finding vs. Group Problem Solving.* The rigid goal of finding defects and measuring success based on the number of defects found per source line lead to a mentality of "Raise issues, don't resolve them" [Johnson 1998]. This mentality has the effect of limiting the ability of the group to collectively problem solve and to mentor an artifact's author [Sauer et al. 2000]. In OSS review, author, reviewers, and other stakeholders freely discuss the best solution, not the existence of defects—there are instances where a reviewer re-writes the code and the author now learns from and becomes a reviewer of the new code.

*Number of Reviewers.* Although inspection has been repeatedly shown to be cost effective over the lifecycle of a project, individual inspection are labour intensive and costly. There has been a substantial literature on the optimal number of reviewers per review [Buck 1981; Bisant and Lyle 1989; Porter et al. 1998; Sauer et al. 2000]. The consensus is that two reviewers find an optimal number of defects. In Section 5.2, we find that there is a median of two reviewers per review. However, given the broadcast mechanism used to disseminate contributions and review discussions, there is the potential to involve a larger number of reviewers should the complexity of the contribution warrant it.

*Process vs. Expertise.* While most studies, including Votta's work on the necessity of a meeting, focused on the impact of process variations, Porter et al. [1998] examined both

the process (e.g., the number of reviews, inspections with and without re-inspection of corrected code) and the inputs to the process (e.g., reviewer expertise and artifact complexity). In terms of the number of defects found, Porter et al. concluded that the best predictor was the level of expertise of the reviewers. Varying the processes had a negligible impact on the number of defects found. This finding is echoed by others (e.g., [Sauer et al. 2000; Kollanus and Koskinen 2009]). While OSS review practices evolved organically to suit the needs of the development team, they mirror Porter et al.'s finding—OSS review has minimal process but relies heavily on self-selecting experts.

*Artifact under Review.* Traditional inspection processes required completed artifacts to be inspected at specific checkpoints. The development of these artifacts was done by individuals or relatively isolated groups of developers. The work could take many months and involve little or no external feedback until it was completed [Votta 1993]. Once complete, these large artifacts could take weeks to inspect. While the importance of shorter development iterations has become increasingly recognized [Larman and Basili 2003], OSS has much smaller change sizes than the industrial projects examined by, for example, Mockus et al. [2002]. These extremely small artifact changes (e.g., between 11 and 32 lines of code modified) and the short time in which the review occurs (hours to days) allows for early, frequent feedback—"continuous asynchronous review".

*Familiarity with Review Artifact.* Aside from the issue of timely feedback, large artifacts were also unfamiliar to the developers tasked with inspecting them, making preparing for and inspecting an artifact an unpleasant task. Not only was it tedious, but as Parnas and Weiss [1985] found, inspectors were not adequately prepared and artifacts were being poorly inspected. Parnas and Weiss suggested active reviews that increased the quality of inspections by making inspectors more involved in the review. Porter and Votta [1994] and Basili et al. [1996] and many other researchers developed and validated software reading techniques that focus developers on different scenarios and perspectives. While researchers continue to question the effectiveness of these scenarios and checklists [Hatton 2008], the main idea was to ensure that reviewers paid attention during a review and that they focused on the areas for which they had expertise.

OSS reviews take this need for focused expertise in a different direction. First, reviews of small artifacts increases the likelihood that developers will be able to understand and focus on the whole artifact. Second, the high frequency of reviews means that developers are regularly and incrementally involved in the review of an artifact, likely lessening the time required to understand the artifact under review. Third, many systems and subsystems involve multiple co-developers who already understand the system and only need to "learn" the change to the artifact under review. Finally, outsiders can learn about the system and provide expertise that is missing from the core team.

### 7.2. Limitations and Validity

*Construct Validity.* Our measures are based on those that have been used in the past to gauge the efficacy of inspection processes (e.g., [Kollanus and Koskinen 2009; Porter et al. 1998]) and should have high construct validity. However, the dataset and creation of these measures are very different from past experiments. The data were not collected for the purpose of measurement. We have discussed the limitations of each measure in the section in which it was used. Comparisons with other past data sets should be done with caution.

*Reliability.* Our study has high reliability because the data is publicly available and our measures are proxies of those used in previous work. Other researchers can replicate and expand upon our results.

*External Validity.* By examining a diverse set of OSS projects, we increase the generalizability of our results. We examined 25 projects at the review policy level. As discussed in Section 3.2, each project was theoretically sampled as a literal or contrasting replication. The first case studies are of smaller infrastructure projects, Apache and Subversion, followed by larger infrastructure projects, Linux and FreeBSD. KDE and Gnome, which include many end-user applications, serve as a counterbalance to the infrastructure projects.

There are three main limitations to the generality of this study. First, we purposefully choose to examine successful, large, mature OSS projects. While we now have a good understanding of how these projects conduct peer review, it is unclear whether these results transfer to smaller projects or help less successful projects. Second, most of the projects we examined use either C or C++. Although it is possible that different languages may change our results, the review process is not coupled to language constructions. Further, many of the review discussions quickly became abstract discussions rather than focusing on code [Rigby and Storey 2011]. Finally, KDE and Gnome conduct a significant number of reviews in a bug tracking system and some discussions occur on IRC chat channels. While there is work examining these other mediums [Jeong et al. 2009; Nurolahzade et al. 2009; Shihab et al. 2009], a comparison of review on different forums would be interesting future work.

*Internal Validity.* The descriptive statistics that we have collected clearly show that OSS review is drastically different from traditional inspection processes. Our model of efficiency explained 29% of the variance, which is inline with other models of inspection efficiency [Porter et al. 1997, 1998]. Our model of effectiveness had a pseudo-$R^2$ of 0.51. These values leave open the possibility that other factors may be important in determining the efficacy of review. Another possibility is that outliers may be affecting our results. Due to the extremely large number of reviews, it is difficult to eliminate the large number of potential outliers. All attempts to systematically eliminate outliers so far have resulted in a decrease in the variance explained by the models.

### 7.3. Concluding Remarks

This work describes the first set of case studies that systematically and comparatively quantified how peer reviews are performed on successful, mature OSS projects. The work is based on Rigby et al. [2008], Rigby's [2011] dissertation, and an earlier technical report by Rigby and German [2006]. Prior to this work, there were only anecdotal descriptions and limited quantitative analyses of OSS peer review practices. There are three contributions made in this work. First, we developed measures from email and version control archives that mirrored the measures used over the last 35 years to assess software inspection in proprietary development. From this unstructured data, we measured the frequency of review, the level of participation in reviews, the experience and expertise of the reviewers, the size of the artifact under review, the calendar time to perform a review, and the number of issues that are discussed during review. Second, we examined the review policies of a 25 successful OSS projects and conducted a detailed quantitative analysis on the Apache httpd Server, Subversion, Linux, FreeBSD, KDE, and Gnome. Finally, we created statistical models of the efficiency and effectiveness of OSS review practices. However, descriptive statistics alone are enough to see that the parameters of OSS peer review are drastically different from traditional software inspection. We summarized our observations as a theory of OSS peer review practices. A detailed qualitative analysis involving manual coding of reviews and interviews of top reviewers on the same six projects can be found in Rigby and Storey [2011].

OSS peer review and inspection as envisioned by Fagan in 1976 have little in common beyond a belief that peers will effectively find software defects in software artifacts. In

contrast to top-down development practices, OSS review practices evolved organically from the needs of the core development team. The recent inspection literature and our findings regarding OSS review imply that the formality and rigidity of Fagan inspections is unnecessary. Peer review practices that are conducted asynchronously, that empower experts, that provide timely feedback on small changes, and that allow developers to focus on their area of expertise are more efficient than formal peer review techniques and are still able to detect issues.

We have begun to investigate possible transfers of our findings to other development environments. An IEEE Software paper orients our findings toward practitioners [Rigby et al. 2012]. For example, we suggest that dividing reviews into smaller, independent, complete pieces may reduce the burden placed on any individual reviewer and divide the often unpleasant review task into more manageable chunks that can be conducted periodically throughout the day. These changes might result in industrial developers taking a more positive view of peer review. We are working with the Canadian Department of Defense to develop peer review practices that are lightweight while still retaining a high degree of traceability and quality assurance.

## REFERENCES


A. Ackerman, L. Buchwald, and F. Lewski. 1989. Software inspections: An effective verification process. *IEEE Softw.* 6, 3, 31–36.

J. Asundi and R. Jayant. 2007. Patch review processes in open source software development communities: A comparative case study. In *Proceedings of the 40th Annual Hawaii International Conference on System Sciences (HICSS'07)*. 10.

V. Basili, S. Green, O. Laitenberger, F. Lanubile, F. Shull, S. Sørumgård, and M. Zelkowitz. 1996. The empirical investigation of perspective-based reading. *Empir. Softw. Eng.* 1, 2, 133–164.

C. Bird, A. Gourley, and P. Devanbu. 2007. Detecting patch submission and acceptance in OSS projects. In *Proceedings of the 4th International Workshop on Mining Software Repositories (MSR)*. IEEE Computer Society, 4.

C. Bird, A. Gourley, P. Devanbu, M. Gertz, and A. Swaminathan. 2006. Mining email social networks. In *Proceedings of the 3rd International Workshop on Mining Software Repositories (MSR)*. ACM Press, 137–143.

D. Bisant and J. Lyle. 1989. A two-person inspection method to improve programming productivity. *IEEE Trans. Softw. Eng.* 15, 10, 1294–1304.

S. Breu, R. Premraj, J. Sillito, and T. Zimmermann. 2010. Information needs in bug reports: Improving cooperation between developers and users. In *Proceedings of the ACM Conference on Computer Supported Cooperative Work*. ACM Press, 301–310.

F. Buck. 1981. Indicators of quality inspections. Tech. rep. TR 21, IBM Syst. Commun. Division.

K. Burnham and D. Anderson. 2002. *Model Selection and Multimodel Inference: A Practical Information-Theoretic Approach*. Springer Verlag.

A. Cockburn. 2004. *Crystal Clear a Human-Powered Methodology for Small Teams*. Addison-Wesley Professional.

J. Cohen. 2006. *Best Kept Secrets of Peer Code Review*. Smart Bear Inc.

M. Crawley. 2005. *Statistics: An Introduction Using R*. John Wiley and Sons.

T. Dinh-Trong and J. Bieman. 2005. The FreeBSD project: A replication case study of open source development. *IEEE Trans. Softw. Eng.* 31, 6, 481–494.

P. Eggert, M. Haertel, D. Hayes, R. Stallman, and L. Tower. 2002. *diff – Compare files line by line*. Free Software Foundation, Inc.

S. G. Eick, C. R. Loader, M. D. Long, L. G. Votta, and S. V. Wiel. 1992. Estimating software fault content before coding. In *Proceedings of the 14th International Conference on Software Engineering*. 59–65.

M. Fagan. 1986. Advances in software inspections. *IEEE Trans. Softw. Eng.* 12, 7, 744–751.

M. Fagan. 2002. A history of software inspections. In *Software Pioneers: Contributions to Software Engineering,* Springer-Verlag, Inc., 562–573.

M. E. Fagan. 1976. Design and code inspections to reduce errors in program development. *IBM Syst. J.* 15, 3, 182–211.

R. T. Fielding and G. Kaiser. 1997. The Apache HTTP server project. *IEEE Internet Comput.* 1, 4, 88–90.

K. Fogel. 2005. *Producing Open Source Software*. O'Reilly.

Freshmeat. 2009. About Freshmeat: A catalog of software projects. `http://freshmeat.net/about` (Last accessed September 2009).

D. German. 2007. Using software distributions to understand the relationship among free and open source software projects. In *Proceedings of the 4th International Workshop on Mining Software Repositories (MSR)*. IEEE, 8.

C. Gutwin, R. Penner, and K. Schneider. 2004. Group awareness in distributed software development. In *Proceedings of the ACM Conference on Computer Supported Cooperative Work (CSCW)*. 72–81.

S. Hambridge. 1995. Netiquette guidelines. `http://edtech2.boisestate.edu/frankm/573/netiquette.html`.

R. Hartill. 1998. Email. `http://mail-archives.apache.org/mod_mbox/httpd-dev/199801.mbox%3CPine.NEB.3.96.980108232055.19805A-100000@localhost%3E`.

L. Hatton. 2008. Testing the value of checklists in code inspections. *IEEE Softw.* 25, 4, 82–88.

H. Heinzl and M. Mittlböck. 2003. Pseudo r-squared measures for poisson regression models with over- or underdispersion. *Computat. Stat. Data Analy.* 44, 1–2, 253–271.

A. Hindle, M. W. Godfrey, and R. C. Holt. 2008. Reading Beside the Lines: Indentation as a proxy for complexity metric. In *Proceedings of the 16th IEEE International Conference on Program Comprehension (ICPC)* . IEEE Computer Society, 133–142.

J. Howison, M. Conklin, and K. Crowston. 2006. FLOSSmole: A collaborative repository for FLOSS research data and analyses. *Int. J. Inf. Techno. Web Eng.* 1, 17–26.

G. Jeong, S. Kim, T. Zimmermann, and K. Yi. 2009. Improving code review by predicting reviewers and acceptance of patches. ROSAEC Memo 2009-006, Research on Software Analysis for Error-Free Computing.

P. M. Johnson. 1998. Reengineering inspection. *Commun. ACM* 41, 2, 49–52.

P. M. Johnson and D. Tjahjono. 1998. Does every inspection really need a meeting? *Empir. Softw. Eng.* 3, 1, 9–35.

P. Kampstra. 2008. Beanplot: A boxplot alternative for visual comparison of distributions. *J. Stat. Softw. Code Snippets* 1 28, 1–9.

P. Kennedy. 1981. Estimation with correctly interpreted dummy variables in semilogarithmic equations. *Am. Econ. Rev.* 71, 4, 801.

J. C. Knight and E. A. Myers. 1993. An improved inspection technique. *Commun. ACM* 36, 11, 51–61.

S. Kollanus and J. Koskinen. 2009. Survey of software inspection research. *Open Softw. Eng. J.* 3, 15–34.

O. Laitenberger and J. DeBaud. 2000. An encompassing life cycle centric survey of software inspection. *J. Syst. Softw.* 50, 1, 5–31.

C. Larman and V. Basili. 2003. Iterative and incremental developments: A brief history. *Computer* 36, 6, 47–56.

G. Lee and R. Cole. 2003. From a firm-based to a community-based model of knowledge creation: The case of the Linux kernel development. *Org. Sci.* 14, 6, 633–649.

S. Lussier. 2004. New tricks: How open source changed the way my team works. *IEEE Softw.* 21, 1, 68–72.

F. Macdonald and J. Miller. 1999. A comparison of computer support systems for software inspection. *Autom. Softw. Eng.* 6, 3, 291–313.

J. Maindonald and J. Braun. 2003. *Data Analysis and Graphics Using R: An Example-Based Approach*. Cambridge University Press.

J. Martin and W. T. Tsai. 1990. N-Fold inspection: A requirements analysis technique. *ACM Commun.* 33, 2, 225–232.

V. Mashayekhi, C. Feulner, and J. Riedl. 1994. CAIS: Collaborative asynchronous inspection of software. In *Proceedings of the 2nd ACM SIGSOFT Symposium on Foundations of Software Engineering*. ACM Press, 21–34.

P. McCullagh and J. Nelder. 1989. *Generalized Linear Models* (Monographs on Statistics and Applied Probability Book 37). Chapman & Hall/CRC.

A. Mockus, R. Fielding, and J. Herbsleb. 2000. A case study of open source software development: The Apache server. In *Proceedings of the 22nd International Conference on Software Engineering (ICSE)*. 262–273.

A. Mockus, R. T. Fielding, and J. Herbsleb. 2002. Two case studies of open source software development: Apache and Mozilla. *ACM Trans. Softw. Eng. Methodo.* 11, 3, 1–38.

A. Mockus and J. D. Herbsleb. 2002. Expertise browser: A quantitative approach to identifying expertise. In *Proceedings of the 24th International Conference on Software Engineering (ICSE'05)*. ACM Press, 503–512.

N. Nagappan and T. Ball. 2005. Use of relative code churn measures to predict system defect density. In *Proceedings of the 27th International Conference on Software Engineering (ICSE)*. IEEE, 284–292.

Netcraft. 2010. Web Server Survey. http://news.netcraft.com/archives/2010/06/16/june-2010-web-server-survey.html.

M. Nurolahzade, S. M. Nasehi, S. H. Khandkar, and Shreya Rawal. 2009. The role of patch review in software evolution: An analysis of the Mozilla Firefox. In *Proceedings of the International Workshop on Principles of Software Evolution*. 9–18.

D. L. Parnas and D. M. Weiss. 1985. Active design reviews: Principles and practices. In *Proceedings of the 8th International Conference on Software Engineering (ICSE)*. IEEE Computer Society Press, 132–136.

D. Perry, A. Porter, M. Wade, L. Votta, and J. Perpich. 2002. Reducing inspection interval in large-scale software development. *IEEE Trans. Softw. Eng.* 28, 7, 695–705.

A. Porter, H. Siy, A. Mockus, and L. Votta. 1998. Understanding the sources of variation in software inspections. *ACM Trans. Softw. Eng. Methodol.* 7, 1, 41–79.

A. Porter and L. Votta. 1994. An experiment to assess different defect detection methods for software requirements inspections. In *Proceedings of the 16th International Conference on Software Engineering*. IEEE Computer Society Press, 103–112.

A. A. Porter, H. P. Siy, and G. V. J. Lawrence. 1997. Understanding the effects of developer activities on inspection interval. In *Proceedings of the 19th International Conference on Software Engineering (ICSE)*. 128–138.

R Development Core Team. 2011. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria. ISBN 3-900051-07-0.

E. S. Raymond. 1999. *The Cathedral and the Bazaar*. O'Reilly and Associates.

P. Resnick. 2001. RFC 2822: Internet message format. http:www.ietf.org/rfc/rfc2822.txt.

P. Rigby, B. Cleary, F. Painchaud, M. Storey, and D. German. 2012. Contemporary peer review in action: Lessons from open source development. *IEEE Softw.* 29, 6, 56–61.

P. C. Rigby. 2011. Understanding open source software peer review: Review processes, parameters and statistical models, and underlying behaviours and mechanisms. Ph.D. Dissertation, University of Victoria, Canada. http://hdl.handle.net/1828/3258.

P. C. Rigby and D. M. German. 2006. A preliminary examination of code review processes in open source projects. Tech. rep. DCS-305-IR. University of Victoria.

P. C. Rigby, D. M. German, and M.-A. Storey. 2008. Open source software peer review practices: A case study of the Apache server. In *Proceedings of the 30th International Conference on Software Engineering (ICSE)*. 541–550.

P. C. Rigby and M.-A. Storey. 2011. Understanding broadcast based peer review on open source software projects. In *Proceeding of the 33rd International Conference on Software Engineering (ICSE'11)*. ACM, New York, NY, 541–550.

C. Sauer, D. R. Jeffery, L. Land, and P. Yetton. 2000. The effectiveness of software development technical reviews: A behaviorally motivated program of research. *IEEE Trans. Softw. Eng.* 26, 1, 1–14.

E. Shihab, Z. Jiang, and A. Hassan. 2009. On the use of Internet relay chat (IRC) meetings by developers of the GNOME GTK+ project. In *Proceedings of the 6th IEEE International Working Conference on Mining Software Repositories (MSR)*. IEEE Computer Society, 107–110.

SourceForge. 2010. SourceForge: Find and develop open source software. http://sourceforge.net/about (Last accessed December 2010).

L. G. Votta. 1993. Does every inspection need a meeting? *SIGSOFT Softw. Eng. Notes* 18, 5, 107–114.

P. Weissgerber, D. Neu, and S. Diehl. 2008. Small patches get in!. In *Proceedings of the 5th IEEE International Working Conference on Mining Software Repositories (MSR)*. ACM, 67–76.

K. E. Wiegers. 2001. *Peer Reviews in Software: A Practical Guide*. Addison-Wesley Information Technology Series. Addison-Wesley.

R. K. Yin. 2003. *Applications of Case Study Research* (2nd Ed.). Applied Social Research Methods Series, Vol. 34. Sage Publications Inc.

Received August 2011; revised December 2013; March 2014; accepted March 2014