Empirical Software Engineering manuscript No.
(will be inserted by the editor)

Correlations between Bugginess and Time-Based Commit Characteristics

Jon Eyolfson · Lin Tan · Patrick Lam

the date of receipt and acceptance should be inserted later

Abstract Modern software is often developed over many years with hundreds of thousands of commits. Commit metadata is a rich source of time-based characteristics, including the commit's time of day and the commit frequency and seniority of its author. The "bugginess" of a commit is also a critical property of that commit. In this paper, we investigate the correlation between a commit's time-based characteristics and its "bugginess"; such results can be useful for software developers and software engineering researchers. For instance, developers or code reviewers might be well-advised to thoroughly verify commits that are more likely to be buggy.

In this paper, we study the correlation between a commit's bugginess and the time of day of the commit, the day of week of the commit, the commit frequency and seniority of the commit authors, and whether or not the developers have marked a commit as a "stable" commit. We survey three widely-used open source projects: the Linux kernel, PostgreSQL, and the Xorg server.

Our main findings include: (1) commits between midnight and 4 AM (referred to as late-night commits) are significantly buggier and commits between 8 AM and noon are less buggy, implying that developers may want to double-check their own late-night commits; (2) daily-committing developers produce less-buggy commits, indicating that we may want to promote the practice of daily-committing developers reviewing other developers' commits; (3) the bugginess of commits versus day-of-week varies for different software projects; and (4) stable commits are significantly less buggy than commits in general.

This submission extends a previous publication on the same subject. A paragraph at the end of Section 1 explains the relationship of this work with our previous work.
This work is supported by Canada's Natural Science and Engineering Research Council and an Ontario Graduate Scholarship. We would also like to acknowledge help from the Statistical Consulting Service at the University of Waterloo. We thank Tian Jiang for identifying the precision and recall of the heuristic for identifying bug-fixing commits in Xorg.

J. Eyolfson, L. Tan, and P. Lam
E-mail: {jeyolfso, lintan}@uwaterloo.ca, p.lam@ece.uwaterloo.ca
University of Waterloo

Keywords Bug Detection · Empirical Study · Source Control System

1 Introduction

Software users demand high software reliability. However, as software complexity increases, bug counts and rates inevitably rise, which undermine software reliability. The modern software development paradigm further complicates the situation: many modern software projects, including the Linux kernel, PostgreSQL, the Xorg server, Eclipse, and Apache, are developed by tens to thousands of developers, over decades, in a distributed manner. The software often receives tens of thousands or hundreds of thousands of commits (Section 3). Developers with different programming experience, time commitments, working hours, programming styles, and from diverse cultures across the world, work on the same software project at different times and in different time zones. They join and leave projects at their own pace over periods of decades. Code developed in the modern paradigm can therefore have different characteristics from older, more homogeneously-developed projects; these characteristics can best be measured by going beyond the code itself and into the developers' time-based characteristics related to the code.

Time-based software characteristics provide a rich and unique source of information for us to understand software and its bugs. As an example, it would be helpful to know if a commit's timestamp (including features such as time of day, day of week, etc.) affects the quality of that commit—are commits after midnight buggier than other commits? Such correlations may be useful for predicting what commits are more likely to be buggy so that we can budget more testing effort on these commits, following prior studies [9,10,12,14,13, 17,18,20,22,31,32]. These prior studies predict buggy locations based on code complexity, code locations, the amount of in-house testing, historical data, socio-technical networks, etc. A second question is: are developers who work on a project for a longer period of time (referred to as developers' *seniority*), or who commit more frequently, more or less likely to write buggy commits?

Contributions In this paper, we study time-based characteristics of modern software development to understand the correlation between these characteristics and the bugginess of commits to the software—the likelihood that a particular commit is later fixed, as determined by the fixing author. Specifically, we study the latest versions of the Linux kernel, PostgreSQL, and the Xorg server, which have 222,332, 31,098, and 10,835 commits, respectively. We study the correlation between a commit's bugginess and the time of day of the commit, the day of week of the commit, the commit frequency and seniority of the commit authors, and whether or not the developers have chosen to apply a commit to a maintenance-only "stable" branch. In addition, we study several other commit characteristics, such as comment-only fixes and bug lifetimes. To the best of our knowledge, we are the **first** to study the correlation between the commit time of day and the commit correctness.

To study the correlation between commit time and commit bugginess, we start from *bug-fixing commits*, commits that fix software bugs, and then mine the version control history to discover when the corresponding bugs were introduced [25]. Our methodology enables us to observe circumstances where bugs are more likely to be introduced. Note that we simply use "bug" to denote code that is later changed, even though such code may objectively be correct; we expand on this discussion later, in Section 2.

It is difficult to find bug-fixing commits in the sea of software commits. Prior work [25] defines a bug-fixing commit to be a commit whose commit message contains a bug ID that links to a bug report in a bug database. While this approach works for some projects, e.g., Mozilla, it does not work for software whose commit messages rarely contain links to bug reports, e.g., the Linux kernel. We have observed that only 2.3% of the bug-fixing commits in the Linux kernel are linked to a bug report. We address this problem by applying heuristics that scan commit messages; they do not rely on any links between bug commits and bug reports to extract bug-fixing commits. Our heuristics have a precision of 75%-87% in identifying bug-fixing commits (Section 3).

We summarize our major findings below (§ denotes the section where the finding and its *implications* are discussed):

- **Finding 1 (§3.1):** About a quarter (20.3–25.5%) of all the commits in the Linux kernel, PostgreSQL, and the Xorg server are "buggy": we observed further developer activities to fix them.
- **Finding 2 (§3.2):** Commits that are first checked into a developer's local repository around midnight (between 0:00–4:00 AM) are more likely to be incorrect than average, while commits in the morning (8:00 AM–noon) are more likely to be correct.
- **Finding 3 (§3.3):** For some projects, developers who commit to the repository on a daily basis write less-buggy commits than an "average" developer.
- **Finding 4 (§3.6):** In contrast to a prior finding that Friday commits are buggier [25], our results on the Linux kernel PostgreSQL, and the Xorg server show that the bugginess differences of commits that are checked in on different days of the week are small and inconsistent across projects. Our results imply that bugginess prediction based on day-of-week needs to be calibrated on a per-project basis.
- **Finding 5 (§3.7):** For the Linux kernel, commits that the developers label as "stable" commits are significantly less buggy than general commits to the Linux kernel repository. Stable commits are simpler and more well-tested than general commits. This finding verifies one's intuition that being selective about commits does, in fact, reduce the bugginess rate for those commits, and that the extra effort required to maintain a stable branch is not wasted.

Previous version of this paper This article extends our previous conference publication [8]. The novel contributions of this article include an examination of the bugginess of the Linux kernel commits that developers mark "stable"; a new benchmark, the Xorg server, which happens to share numerous committers with the Linux kernel; and a study of the commit bugginess characteristics of the authors who commit to both the Linux kernel and Xorg.

2 Experimental Methods

Our overall goal is to investigate the properties of "buggy", or bug-introducing, commits. We define a *bug-introducing* commit to be any commit for which there exists a later *bug-fixing* commit which purports to fix the bug. A single bug-fixing commit may fix bugs introduced in multiple bug-introducing commits. Despite our terminology, a bug-introducing commit is not necessarily bad code; it is possible that the later fix is adaptive or perfective, updating the code to work with changes in third-party code, or reflecting a change in requirements.

2.1 Core Methods

Following Śliwerski et al [25], our methodology has three steps: 1) enumerating bug-fixing commits; 2) identifying the lines changed in each bug-fixing commit; and 3) finding the commits that were responsible for the previous (buggy) version of each of the changed lines. We describe each of these steps in more detail. First, to detect a bug-fixing commit c, we searched the commit messages for the keyword "fix" (as do [23]). In our experience, most developers indicate that a change is a fix by including the keyword "fix" in the commit message. Section 3 explains how we verified this intuition; our results show that the precision of this heuristic for identifying bug-fixing commits is 75%–87%. Next, we computed diffs for each file changed in c, omitting comments, and recorded the line numbers L_c that c changed. Finally, we searched the repository metadata (as implemented by the "git blame" command) to identify the bug-introducing commits, c', which changed the lines L_c. In this step, we used git blame's `-w` option, which ignores whitespace in attributing responsibility for a code change, thus addressing some inaccuracies that exist in other version control systems such as CVS and Subversion, as discussed by Kim et al [16].

Example Figure 1 presents an example of a bug-fixing commit. Commits consist of a commit id, which is a hash of the commit's contents, represented as a hexadecimal number[1]; an author, identified by a name/email address pair; a commit message, which contains the keyword "fix"; and a diff, showing the lines the commit modified.

First, we find the commit in Figure 1 by searching the commit logs for the keyword "fix", which is indeed a substring of the commit message, "I fixed a bug!". Next, we observe that the commit modifies line 100 of an (unidentified)

[1] Following common practice, we drop trailing digits of the commit id: our commits have ids with unique first-8-digits.

S

```
Commit: 2cdc03fe...
Author: Alice <alice@project.com>
Message: I fixed a bug!
@@ -100,1 +100,1 @@
-     if (i <= 128) {
+     if (i < 128) {
```

Fig. 1 An example bug-fixing commit

```
f4ce718c...   100     if (i <= 128) {
```

Fig. 2 `git blame` output for the bug-fixing commit

```
Commit: f4ce718c...
Author: Bob <bob@project.com>
Message: I hope this works.
@@ -100,0 +100,5 @@
+     if (i <= 128) {
+         do_ascii(i);
+     else {
+         do_unicode(i);
+     }
```

Fig. 3 Associated bug-introducing commit for the example

file—in this case, the original author used less-than-or-equal (`<=`) instead of strictly-less-than (`<`), and the bug-fixing commit changes the comparison operator to the presumably-correct one. Finally, we perform a "git blame" on this commit, which shows the commit responsible for the previous version of line 100. Figure 2 presents plausible "git blame" output, which shows that the source of the bug fixed in Figure 1 is the commit whose id begins with f4ce718c, as shown in Figure 3. We flag this commit as a bug-introducing commit and store both the bug-introducing commit f4ce718c and its bug-fixing commit 2cdc03fe in our database, along with an association between these two commits. If a bug was introduced by multiple commits, then all of these bug-introducing commits are stored in our database.

Any change in c that removes or modifies an existing line of code is easy to attribute to a previous commit c', since the affected line of code existed in c'. However, a change in c that adds a new line of code has no corresponding change in any previous revision, since that line did not previously exist. In that case, we attribute responsibility to the commit that introduced the line just before the new line. This heuristic does not work for bug-introducing changes in newly-introduced files. Fortunately, our data show that such changes are extremely rare, so ignoring them should not affect the validity of our study.

2.2 Data Collected

Executing the above algorithm gives us data about the bug-fixing and bug-introducing commits in each repository, as well as about the authors of these commits. We record the following data for each commit: author (as a name/e-mail pair); the author's adjusted local commit time (we describe our adjustments below); number of lines changed; and number of times the commit introduced a bug later corrected (which is derived data; we record it to simplify later database queries). We also record a relation connecting bug-introducing commits and bug-fixing commits. For the Linux kernel repository, we mark some commits as stable commits, and maintain a relation between stable and mainline commits. For each author, we record the name, email(s) and commit frequency classification (defined below). We define a bug's *lifetime* to be the time from the earliest commit that introduced the bug to the bug-fixing commit.

We compute each author's commit frequency classification, based on the frequency of an author's commits, and author seniority at commit time for each patch, based on the elapsed time between that author's first observed commit and the commit time.

The *author commit frequency* classification describes an author's most-common interval between two consecutive commits: daily, weekly, monthly, other (less than 20 commits and more than 1 commit), and single (only 1 commit). For the author commit frequency classification, we count consecutive commits within 30 minutes of each other as one commit.

Author seniority reports the amount of time since an author's first commit to a project. To study the correlation between commit bugginess and author seniority, we calculate the author's seniority at the time of the commit. For example, author X's first commit to the Linux kernel was on March 28, 2009, so that commit is by an author with 0 days of seniority. Then, the second commit was on April 13, 2009, so it is by an author with 16 days of seniority. We bin the commits by author seniority and present the percentage of buggy commits for each author seniority bin.

Stable commits There are two types of the Linux kernel releases: mainline releases (numbered like 2.6.39 or 3.1), and stable revisions of mainline releases (numbered like 2.6.39.y or 3.1.y). Commits towards stable revisions, or stable commits, must: "be obviously correct and tested"; contain fewer than 100 lines of changes; only fix one problem; and exist in the main kernel repository. The documentation[2] also lists several other properties, which are unimportant for this paper.

The provenance of stable commits is as follows. Developers nominate commits for the stable branch, and the stable branch committee "cherry picks" acceptable commits from the main kernel repository. Git treats "cherry picked" commits as new commits; they do not contain any link to the original commit,

[2] `http://lxr.linux.no/#linux+v3.1.6/Documentation/stable_kernel_rules.txt`

and may contain modifications of that commit. We infer links by textually matching the commit message summary of a stable commit *s* with the closest match *c* in the main repository, and record the inferred link between *s* and *c* in our database. Note that a main branch commit *c* can be associated with more than one stable commit. We found, on average, that stable commits belong to 2 branches: a short-term branch and a long-term branch.

Time zone adjustments All PostgreSQL commits prior to September 18, 2010 (when the project switched to the Git version control system) and all Xorg commits prior to June 6, 2006 contain timestamps only in UTC (Coordinated Universal Time), meaning that no local time zone information was recorded. To enable us to reason about time-of-day effects for committers, who work in local time zones, we used publicly-available information (such as the PostgreSQL contributor-information page, which lists locations for frequent contributors, as well as time zones included in mailing list messages) to deduce time zones for all 34 PostgreSQL committers. We then converted the time for each commit (which was in UTC for CVS commits, and in local times for Git commits) into a local time for the committer, using the Python time zone utilities. We also deduced time zones for the 67 Xorg authors, using the same technique. We assume that each committer is indeed in the time zone we have estimated for that committer; we discuss this threat to validity in greater detail below.

Developer overlap between projects This experiment was prompted by our observation that our view of developer seniority only captures a very limited notion of experience; most programmers do not start by contributing to the Linux kernel! To broaden our notion of developer seniority, we decided to explore developers' contributions to multiple projects, and found that the Xorg server and the Linux kernel shared a nontrivial number of committers. PostgreSQL did not share any committers with either the Xorg server or the Linux kernel.

Understanding similarities and differences in the characteristics of developers' commits across projects has a number of implications. First, it can lead to a better understanding of the benefits and drawbacks of the different development processes employed by various open-source communities (community norms affect the behaviour of the same people in different contexts). In addition, it enables the comparison of characteristics between different projects with overlapping authors; fixing problems that occur across a number of projects can simultaneously improve software quality for all involved projects.

2.3 Threats to Validity

We discuss several threats to validity and how we address them, including general threats to construct validity and external validity, and specific threats to our particular methodology, including repository threats, recall and precision threats, and author identification threats.

General threats Construct validity requires that we correctly identify bug-fixing and bug-introducing commits. To assess threats to construct validity, we determine our confusion matrix by randomly sampling 200 commits from the Linux kernel, PostgreSQL, and Xorg, and manually verifying whether or not they are indeed fixes. False positives and false negatives affect precision and recall. We found that our precision (proportion of reported bug-fixing commits that do fix bugs) was 87% for the Linux kernel, 86% for PostgreSQL, and 75% for Xorg, while our recall (proportion of bug-fixing commits that we identify) was 73% for the Linux kernel, 71% for PostgreSQL, and 64% for Xorg. Section 3.8 elaborates on our evaluation of precision and recall in greater detail.

While we believe that the commits from the software that we examined well represent commits in open source software, we do not intend to claim external validity and draw any general conclusions about all software. Like any other characteristic study, our findings should be considered together with our evaluation methodology.

Given sufficiently many p-value computations, some values will pass the threshold for statistical significance solely by chance. False discovery poses a threat to validity when researchers mine through a large body of results to select (cherry-pick) interesting data points. For the most part, we do not select particular points to discuss; instead, we include all data points that we collect, along with relevant p-values. The sole exception occurs in our identification of trends among the time-of-day results in Section 3.2. Thus, for those results, we carry out a Benjamini-Hochberg correction [1], which limits the false discovery rate to 0.05.

Repository data threats We expect our methodology to properly account for developers in different time zones. Git records each developer's local time (and time zone) with a commit, thereby avoiding potential imprecisions in our time-of-day results. This works well for the Linux kernel repository, which is a native Git repository as of 2005. (Older Linux kernel repository information does not accurately record time-of-day information for commits.) In that repository, the local time might be inaccurate if a developer commits from a server in a different time zone; however, because Git was designed to work best when developers commit to local workstations, we expect that most of the 8000 Linux kernel contributors will commit locally on a machine with accurate local time. The accuracy of the commit times for the PostgreSQL and Xorg server repositories depends on the accuracy of our time adjustment algorithm for the part of the history that was originally a CVS history. We believe that committers do not often change time zones[3] and that they usually work from their home time zone, but the validity of our adjustment does depend on the validity our assumptions about home time zones. We present all of our results in the local time of the committer, when relevant.

[3] We were able to identify one move of a committer from Ontario, Canada to California, and incorporated that move into our adjustments, but did not find evidence of many such moves in our set of PostgreSQL contributors.

We ignore Git merge commits (which only account for 6.25% of all commits) when computing diffs and searching for bug-fixing commits. Merge commits record metadata about integration between different maintainers' trees. We chose to exclude merge commits for two reasons: 1) a merge commit does not introduce any novel changes to the code; and 2) a diff between a merge commit and its predecessor includes all changes that occurred since the initial branching point for that merge. This would double-count a commit: once for the original commit and once for the merge commit. Note that PostgreSQL does not use merge commits.

Because the PostgreSQL and Xorg server repositories were converted from CVS to Git, the accuracy of the conversion is another potential threat to validity. In particular, CVS does not have a notion of atomic multi-file commits, while Git does, and our methodology relies on the existence of such commits. The PostgreSQL conversion used the standard `cvs2git` tool, with customizations for their particular repository [11]. The existence of these customizations lead us to believe that the conversion was performed with care, mitigating threats to validity from repository corruption. In addition, note that the conversion to Git obviates the need to mine transactions from CVS histories, as in [33].

Threats due to imperfect recall Our methodology cannot identify all buggy or bug-fixing commits in the repository histories; we have estimated our recall at 73% for the Linux kernel, 71% for PostgreSQL, and 64% for Xorg. Because our methodology only identifies the sample of bugs that have later been corrected, it will omit recently-introduced bugs, as well as longer-running bugs that have not yet been corrected. We estimate the impact of that phenomenon by characterizing bug lifetimes in Section 3.9.

Although our analysis omits some bug-introducing and bug-fixing commits, we do not believe that this omission affects our validity. Recall that we compare various characteristics (including time-of-day, commit frequency, developer experience, weekday, and whether a commit is marked "stable") to commit bugginess. We believe that there should not be important differences in these characteristics with respect to fixed and unfixed bugs, nor between fixes labelled "fix" and fixes without that label. In other words, we believe that our sample is representative of bug-introducing and bug-fixing commits for these software projects.

Bird et al. [2] identified bias in bug-fixing commits that are linked with bug reports in bug databases. They found that commits linked with bug reports are more likely to be written by more experienced developers. Despite our belief that our sample is unbiased, we acknowledge that it is possible that some of our results may also be subject to similar bias. For instance, more-experienced developers might touch less of the codebase to fix a given bug than less-experienced developers. It would therefore be beneficial to study whether the false negatives have significantly different characteristics from the rest of the commits.

Threats to commit characteristics The next family of threats concerns commit characteristics, including the attributed time and author information for a commit. For Linux and Xorg, the attributed time is the time at which the initial author submits the change to his or her local repository. Commit characteristic-related threats therefore do not apply to the Linux kernel or Xorg repositories, as these communities use the merge functionality of Git to preserve commit metadata for each contribution, whether from a core member or from an external contributor. The implications of Git merges on our data are that 1) the commit time reflects the initial author's commit decision, and 2) the attributed author of a commit actually wrote the commit.

For PostgreSQL, we only have the time at which a committer applies a change to the central repository. We believe that commit characteristics are also meaningful for PostgreSQL, even though it does not use Git merges. The lifecycle of a PostgreSQL patch begins with its posting to the pgsql-hackers mailing list, along with optional submission to a CommitFest to ensure timely community review. The PostgreSQL community thoroughly reviews patches (including patches by committers) before committing them to the main repository; the project documentation suggests that contributors should plan for 3 iterations before acceptance[4]. Finally, once a PostgreSQL committer agrees that the patch is suitable, based on detailed community reviews, he pushes the patch to the main PostgreSQL repository.

Most PostgreSQL patches are committed by someone other than the author of the patch, as the set of PostgreSQL committers is much smaller than the set of PostgreSQL patch authors. Non-committer patches add uncertainty about when a patch was originally written, and confound our data about patch authors: although Git can record both the author and committer of a patch, the PostgreSQL community is currently requiring that the "author" field of a patch always equal its "committer" field. We would have to mine the pgsql-hackers mailing list to match patches to their original authors (and initial submission times). Nevertheless, we believe that the commit timestamp is still meaningful for PostgreSQL, since the committer is taking final responsibility for the patch by committing it[5].

For all projects, developers work on a patch over a possibly discontiguous time interval, yet the final commit only indicates the endpoint of that interval. The bug database may contain more information about the starting point of the interval (e.g. it records when a bug is assigned to a developer), but still does not capture any information about the work patterns of the developer within the interval. Some Git repositories, including some Linux kernel sub-repositories, do contain more information about intermediate local commits by developers, which can help understand the evolution of a commit. There may be correlations between a patch's evolution and its code quality, and we intend to investigate these issues in future work.

[4] `http://wiki.postgresql.org/wiki/Submitting_a_Patch`

[5] Stephen Frost, a PostgreSQL commiter, writes "We depend on the committers to do final review and commit, but they are a very finite resource." in a presentation about PostgreSQL patch reviewing, found at `http://www.pgcon.org/2011/schedule/events/368.en.html`.

Finally, we discuss threats to our author classification. Some Linux kernel and Xorg authors commit from several email addresses and with variations in their names. To clean our author data, we first merge authors with different names but the same email addresses. Then, we merge authors who share the same full name. This is not an issue for PostgreSQL: when they converted their repository to Git, they also normalized all author email addresses, so that each author has a single email address.

Our author seniority only counts the participation of an author to one or two projects; a highly-experienced developer may be classified as a junior developer for one of our projects due to a short history of observed contributions in these projects. Therefore, the author seniority in this paper should be interpreted as the author's seniority with the target project. We believe that it is appropriate to characterize authors by their participation in a single project when analyzing the quality of their contributions to that project.

3 Results

In this section, we present the results obtained from carrying out our methodology, and discuss some implications of our results. Most of our results investigate the effect of an independent variable (e.g., time-of-day, developer commit frequency classifications, developer seniority, and day-of-week) on the likelihood of a commit to be a bug-introducing commit, or *bugginess* (Section 3.2–Section 3.6). Section 3.6 describes the relationship we found between the day of the week and bugginess, which allows us to compare our results to those of Śliwerski et al [25].

We also examined whether developers could identify a subclass of commits as being lower risk than usual (potentially enabling them to leverage the results on bugginess rates at different times of day). Section 3.7 shows that the answer is yes: commits identified by Linux developers as "stable" commits are indeed less buggy than general Linux commits.

Finally, Section 3.8 explains how we validate our results, including an analysis of the precision and recall of our methodology, as well as an evaluation of the significance of the hour as a determining factor for commit bugginess. Section 3.9 addresses a particular facet of our recall results: because our repository snapshot cannot see bugs which have not yet been fixed, our bug lifetimes estimate how long it takes before a bug should show up in our counts.

3.1 Project Characteristics

We chose three large open-source software repositories for our investigations: Linus Torvalds's mainline Linux kernel, PostgreSQL and Xorg server. Table 1 summarizes the characteristics of our repositories. The row "lines of code" refers to the current size of the code in the repository. The row "# bug-introducing" shows that 20.3–25.6% of the commits are buggy, which is slightly lower than the previously reported figure of nearly 40% for a commercial switching system [23]. Note that the PostgreSQL repository was carefully converted from CVS using `cvs2git` in September 2010, and recall that we discussed the quirks of the PostgreSQL repository in Section 2. The Xorg project manually converted the repository for the Xorg server from CVS to git in July 2006.

	Linux kernel	**PostgreSQL**	**Xorg server**
First commit	April 16, 2005	July 9, 1996	November 19, 1999
Cloned	September 9, 2011	September 14, 2011	September 9, 2011
Lines of code	over 5 million	over 750,000	over 550,000
Number of authors	7,434	34	331
Number of commits	263,780	32,499	10,835
# bug-introducing	67,423 (25.6%)	7,911 (24.3%)	2,200 (20.3%)
# bug-fixing	71,219	6,945	2,489

Table 1 Characteristics of the Linux kernel, PostgreSQL, and Xorg server repositories.

3.2 Time-of-day

Figure 4 presents our results correlating the time-of-day of a commit with its bugginess[6]. The graphs compare the time-of-day of each commit, in the committer's local time on a 24-hour clock, to the percentage of bug-introducing commits.

The solid horizontal line indicates the overall percentage of buggy commits in each project; circles below the line indicate that commits at that hour were less likely to be buggy, while circles above the line indicate hours with more-buggy commits.

We computed p-values for the bugginess percentages, which we list in the Appendix and summarize in Figure 4. Our p-value calculations are with respect to the null hypothesis that a data point (bugginess per hour) has the same bugginess probability as the overall bugginess for the project. To compute our (one-sided) p-values, we modelled our observations of the bugginess using a binomial distribution; commits correspond to trials, bugginess of these commits to outcomes of the trials. For instance, we report a bugginess rate of 12.6% for Xorg commits at 8AM with a p-value of 0.0035. There are 24 bug-introducing commits out of 191 total commits to Xorg at 8AM, while the overall bugginess for Xorg is 20.3%. The p-value of 0.0035 represents the probability of there being at most 24 bug-introducing commits, assuming that each of the 191 commits had a 20.3% chance of being buggy.

To limit the false discovery rate, we applied a Benjamini-Hochberg correction with $\alpha = 0.05$. All but 5 of our p-values below 0.05 remain below the adjusted cutoff. Solid black circles indicate a corrected p-value allowing re-

[6] Table 6 in the Appendix presents a complete set of p-values evaluating the statistical significance of the per-hour commit bugginess for the Linux kernel, PostgreSQL, and Xorg.

jection of the null hypothesis; grey black circles indicate a raw p-value below 0.05; and hollow circles indicate higher p-values.

We have also included bolded horizontal line segments, which indicate the average bugginess over a number of hours; the number above the line segment is the p-value for that group of hours. Because we have cherry-picked these segments, the Benjamini-Hochberg correction is useful here; all line segments that we report also pass with $\alpha = 0.05$. We applied the correction individually to the set of 3-hour and 4-hour intervals for each benchmark, following the Benjamini-Hochberg procedure as described in [1].

Results For all three projects, we can observe two overall trends: taken as a group (as depicted by the **bolded** horizontal line segments), commits between midnight and 4AM are buggier than average (max p-value 4e-3 for a 4-hour period), while commits between 8AM and noon are less buggy than average (max p-value 7e-3). The bugginess over the four-hour period for the midnight to 4AM commits (in brackets, for all commits) is 29.37% (overall average 25.6%), 26.51% (24.3%), and 24.13% (20.3%) for Linux, PostgreSQL, and Xorg, respectively. The average bugginess between 8AM and noon is 22.56% (25.6%), 19.82% (24.3%), and 18.04% (20.3%). The Figure presents these average bugginess values using **bolded** horizontal line segments (max p-value 4e-3 for the midnight to 4AM commits). We have drawn these horizontal line segments at the weighted mean bugginess across the segment's time interval. For the 8AM to noon commits, we have also included bolded horizontal line segments for PostgreSQL and Xorg (p-values 1e-9 and 7e-3 respectively); for the Linux kernel, each hour is individually less buggy than average, with max p-value 6.13e-7.

Beyond the macro trends above, we can observe some project-specific tendencies. In the Linux kernel, the bugginess is above average betetween 9PM and midnight (p-value 2e-11); and below average for groups of hours between 5AM and 8AM, 7AM and 11AM, and 10AM and 1PM (max p-value 5e-12). For PostgreSQL, the bugginess is above average between 6PM and 10PM (p-value 1e-5). Finally, for the Xorg server, bugginess is above average between 10PM and 2AM (p-value 1e-3).

Figure 4 also shows the total number of commits per hour. The smallest value, for any hour, is 45 for Xorg (and an order of magnitude higher for the Linux kernel).

We also investigated correlations between the time-of-day and the number of bug-fixing commits, rather than the bug-introducing commits that we showed above. The proportion of total commits that are bug-fixing commits stayed almost constant, independent of the hour; the graphs (not shown) have exactly the same shape as that of the bars in Figure 4. This suggests that the fact that a commit is bug-fixing is independent of its commit time.

Discussion Code does not spontaneously improve if left to "mature" for 4 hours; our results do not indicate causation, but instead demonstrate a correlation between code committed early in the morning and increased bugginess.

We do not speculate about the cause of this correlation; however, the results in Section 3.5 imply that this correlation holds for both senior and junior developers. Our results do suggest that developers, being aware of such a correlation, may want to double-check code, or solicit reviews, before performing late-night commits (midnight–4:00 AM). It may also be beneficial for version control systems or IDEs to warn developers about the perils of late-night commits. Our p-values indicate that our observed bugginess differences between late-night/early-morning and all other commits are statistically significant.

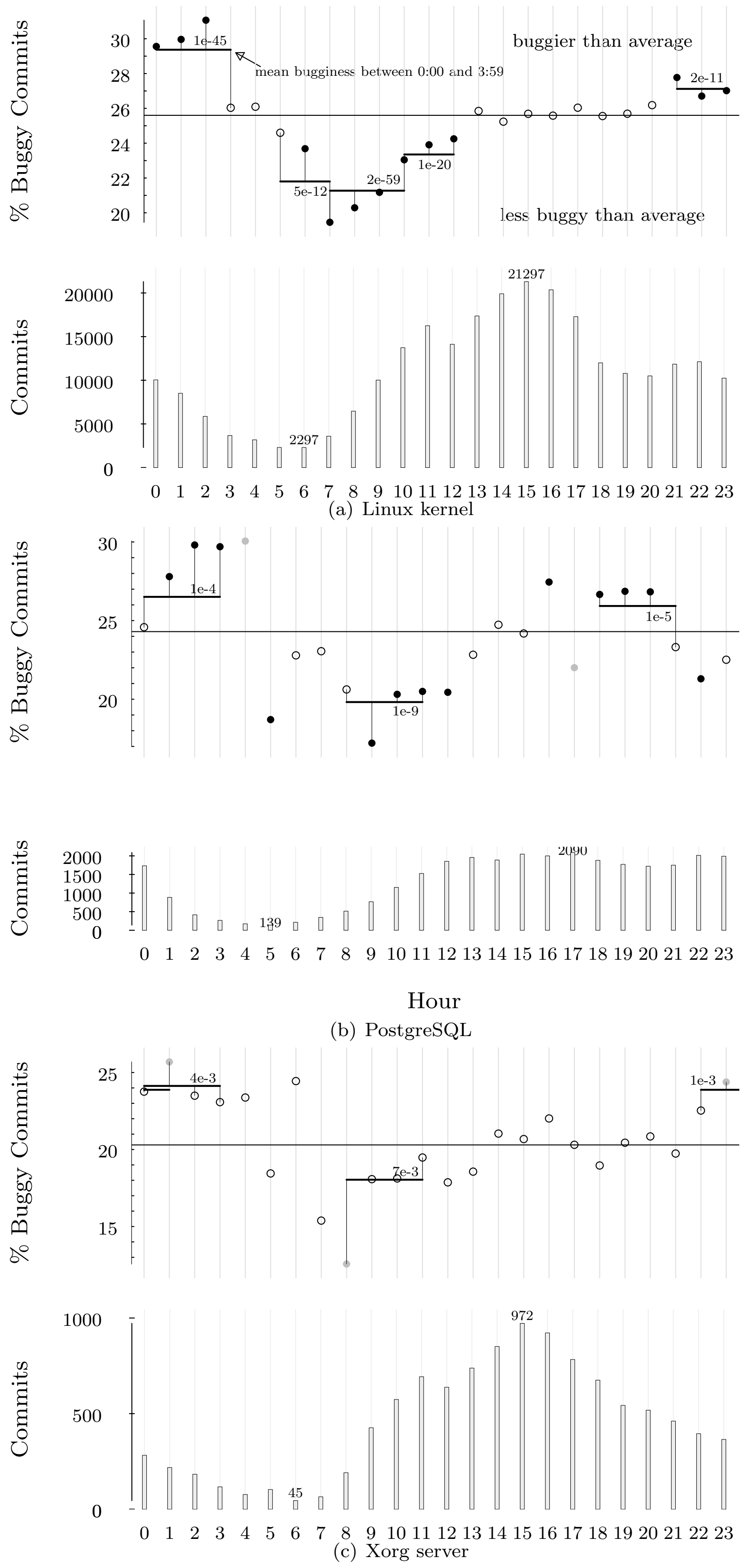


Fig. 4 Percentage of buggy commits (solid black circles indicate p-value < 0.05 after Benjamini-Hochberg correction; grey circles indicate raw p-values < 0.05) and total number of commits (bars) versus time-of-day. Solid horizontal lines indicate the percentage of buggy commits for a project. Horizontal line segments indicate the buggy percentage over several hours; the p-value for those hours is shown above the line segment.

3.3 Developer Characteristics

We next present our findings with respect to developers' commit frequency and seniority. Developers' commit frequency summarizes the frequency of a developer's contributions to a project, while developer seniority tracks how long a developer has contributed to a particular project.

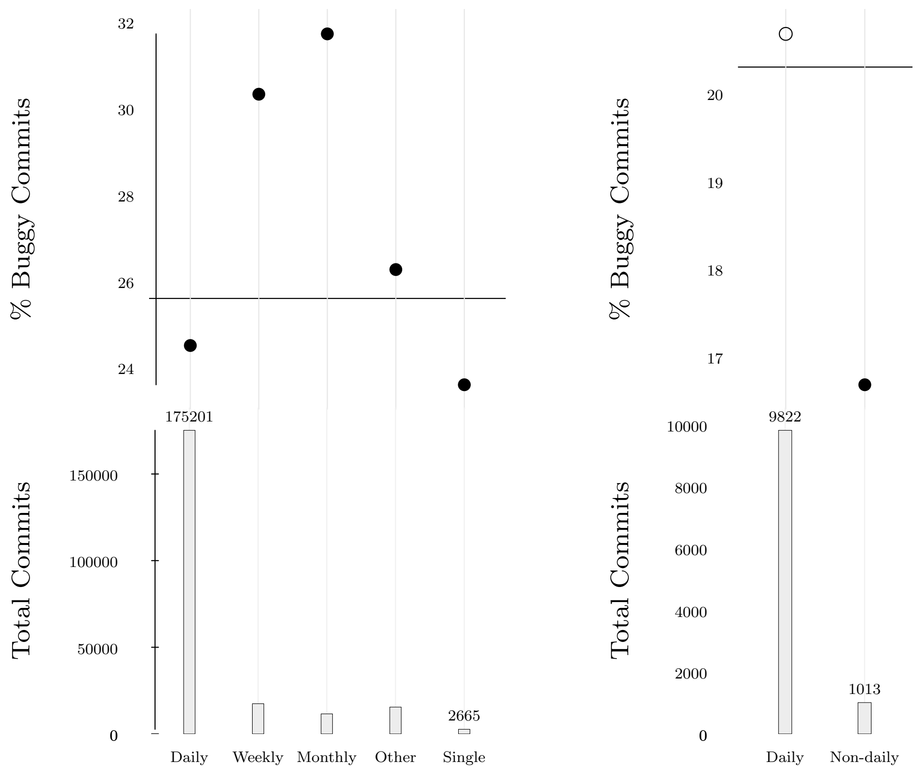
3.3.1 Commit Frequency Classification

As we described in Section 2.2, one of the ways that we classify developers is according to frequency, i.e. most-common interval between consecutive commits—daily, weekly, monthly, other, or single. For the Xorg server, the most important classification is daily versus non-daily, and we present results based on that classification. Almost all (28/34) of PostgreSQL's committers are daily, so we do not present PostgreSQL commit frequency results.

We computed the bugginess rates for each of these classes of developers and plot author classification versus bug-introduction percentage in Figure 5(a)[7]. The graph also presents the number of commits by each class. Note that the Linux kernel has 850 daily authors, who account for the overwhelming majority of commits; 238 weekly; 288 monthly; 3562 other (fewer than 20 commits and more than 1 commit); and 3664 single-commit authors. Figure 5(b) shows the results for the Xorg server. Because the vast majority of Xorg committers are daily committers, we only show the results for daily versus non-daily committers.

Our results show that the Linux kernel developers who commit changes daily produce the largest number of commits and the smallest number of bug-introducing commits, followed by the single-commit authors (whose patches would presumably be simple or closely-reviewed). Weekly and monthly committers produce slightly more bug-introducing commits than average. For the Xorg server, non-daily committers have a bugginess rate of 16.6%, which is below the mean bugginess rate (p-value 0.001).

[7] Table 8 in the Appendix presents p-values evaluating the statistical significance of the per-class commit bugginess for the Linux kernel and Xorg.



(a) Linux kernel (b) Xorg server

Fig. 5 Percentage of buggy commits (solid circles indicate p-value < 0.05) and number of commits versus author classification. The solid horizontal line across an entire subfigure indicates the percentage of buggy commits in that project.

3.3.2 Developer Seniority

Figure 6 compares author seniority at time of commit to the bugginess of the commit[8]. It also presents the total number of commits by author seniority. Our results show that for the Linux kernel, Postgres and Xorg, bugginess generally decreases with increased author seniority. For the Linux kernel, authors with at least 1,440 days of seniority tend to produce commits that are less buggy than average, while the similar point for PostgreSQL occurs at 3,360 days. The Xorg server data shows a spike at the right. Around 2,880 days of seniority, there is a large (statistically significant) spike in buggy commits; this spike is supported by 33 commits at that point.

We manually investigated the spike at day 2,880 of Figure 6(c). Keith Packard, a major contributor to the Xorg server, committed major enhancements to the XRandR functionality of Xorg in February and March of 2008.

[8] See Table 9 in the Appendix for p-values for the Linux kernel, PostgreSQL and Xorg.

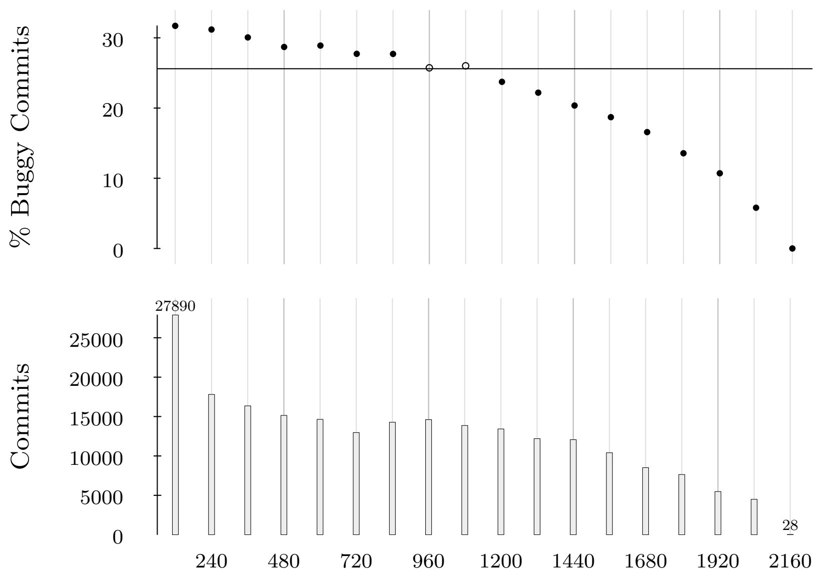
RandR stands for Resize and Rotate, which often involves the use of transformation matrices. A number of subsequent commits modify the February–March 2008 enhancements. Those commits 1) modify RandR to not use *fixed*-point arithmetic and 2) modify the file `xfixes.c`, among other changes. Even though their messages contain the word "fix", these subsequent commits are not actually bug-fixing. They are instead false positives. Together, these two sources of false positives account for 12 false positive reports out of 33 total commits. Manually excluding them would result in a bugginess rate of 8/33, or 24%, which is much closer to the average bugginess rate.

Discussion Our data shows that, in general, the more senior the developers are, the less likely that their commits are buggy. Without further data, this correlation does not prove that the developer seniority caused more senior programmers' commits to be less buggy. While we believe the above causation to hold, other interpretations are possible; perhaps more senior developers wrote more complex code, whose bugs are harder to discover and less likely to be reported. Nonetheless, our results show that, given the fact that a commit is from a more senior developer, one can be more confident that the commit is less likely to be buggy. Such a correlation could be exploited to help predict buggy code locations.

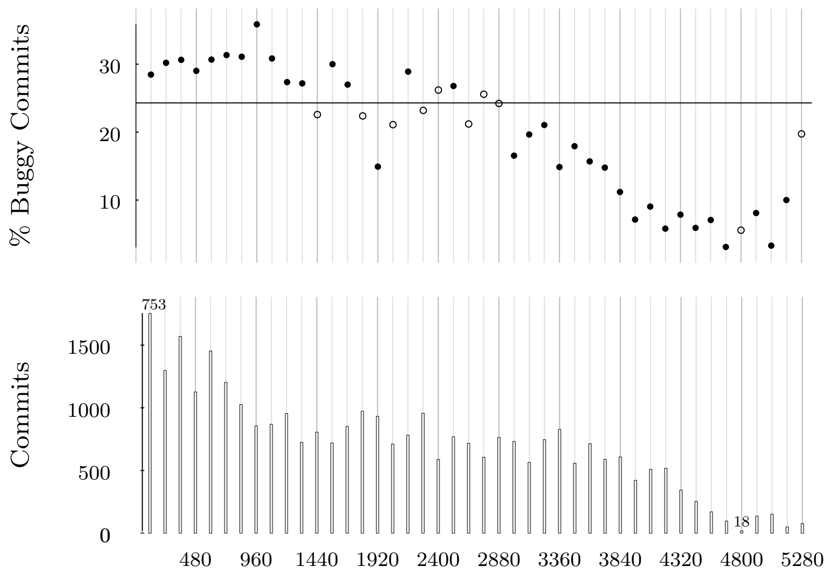
One can observe a decline in the total number of commits with seniority. We believe that this is due to our sliding scale for author seniority. Consider an author who has committed for 5 years. His or her commits do not show up in a single circle at the 1800-day mark; instead, they are distributed throughout the 5 years of the commits, so that a commit on the author's second birthday gets reported as a commit at day 700. One would therefore expect more commits from "junior" developers, since all developers go through a junior phase, while only a small number of developers reach the more senior phase.

We also calculated author seniority based on the first commit to any project for the Linux kernel and Xorg server. We found that this change did not affect the graphs in any significant way.

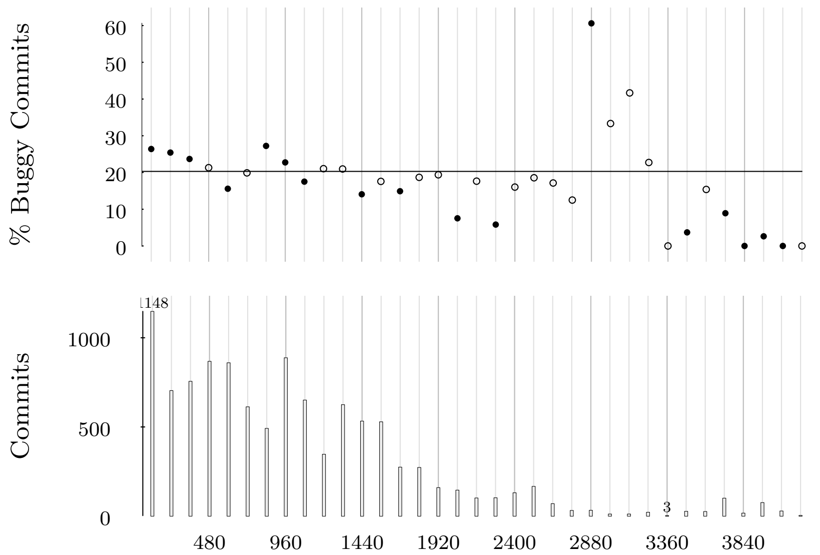
In addition, we studied *developer experience*: the number of lines committed by an author since her first commit to a project (figures not shown). We found no clear correlation between developer experience and commit bugginess for the Linux kernel or Xorg. Commits by more experienced developers in PostgreSQL are less likely to be buggy. Previous work reported no clear correlation between developer experience and commit bugginess for some projects [24], and inverse correlation for some other projects [19, 24]. Our study, on a disjoint set of projects, confirms the observation that the correlation between developer experience and commit bugginess is project-dependent (details in Section 4). Our results show that commits by senior developers are less likely to be buggy for the Linux kernel, Xorg, and PostgreSQL, which indicates that developer seniority can be a better feature for predicting buggy commits than developer experience (at least for our three studied projects).



(a) Linux kernel



(b) PostgreSQL



(c) Xorg server

3.4 Overlapping Developers

Table 2 summarizes our findings about bugginess rate from developers who commit to both the Linux kernel and Xorg server. The null hypothesis used for calculating the p-values is, the buggy rate for a developer type is the same as the overall buggy rate for the project. Commits to Linux from developers working on both projects exhibit no statistically significant difference in bugginess compared to developers working on the Linux kernel alone. For Xorg, we found a statistically significant difference: developers working on both projects commit to Xorg at a higher bugginess rate than developers working only on Xorg. There were no overlapping developers between PostgreSQL and the other projects.

	Developer Type	% buggy commits	Total commits	p-values
Linux	Overlapping	25.1%	19,462	0.0833
	Pure	25.6%	244,316	—
Xorg	Overlapping	21.2%	8,324	0.0179
	Pure	17.2%	2,511	4.61E-05

Table 2 Commit count and buggy commit percentage for joint Linux and Xorg developers

Commit frequency	Overlapping developers	Linux-only developers	Xorg-only developers
Daily	55	731	22
Weekly	4	229	2
Monthly	9	330	0
Other	73	3,023	79
Single	0	2,980	87

Table 3 Commit frequency distributions for the Linux kernel and Xorg developers.

Table 3 shows these authors divided according to their commit frequencies. These frequencies count commits across both projects, e.g. a daily committer commits daily to either the Linux kernel or Xorg. We found that 42.6% of the Xorg developers work on both projects, and these developers mainly fall into the daily or other category. (Recall that "other" committers have between 2 and 19 commits.)

Discussion We found that developers who committed to both the Linux kernel and Xorg committed bugs to the Linux kernel at about the same rate as pure Linux kernel committers. Perhaps kernel code is either complex or unique, and additional seniority does not add any benefit. For commits to the Xorg server, developers working purely on Xorg committed significantly fewer bugs than developers working on both. Because so many Xorg committers also commit to the Linux kernel, we can deduce that the developers working on both projects must therefore commit significantly more bugs than average to Xorg server. It is possible that developers working both repositories write lower-level code, e.g., drivers, which historically contain more bugs than other software components [6,27].

We attempted to investigate the overlap between the Linux kernel and Xorg server with the FreeBSD kernel. However, we found a lack of overlap between these projects. Only 4 authors commit to both the Linux kernel and the FreeBSD kernel, and only 3 authors commit to both Xorg and the FreeBSD kernel.

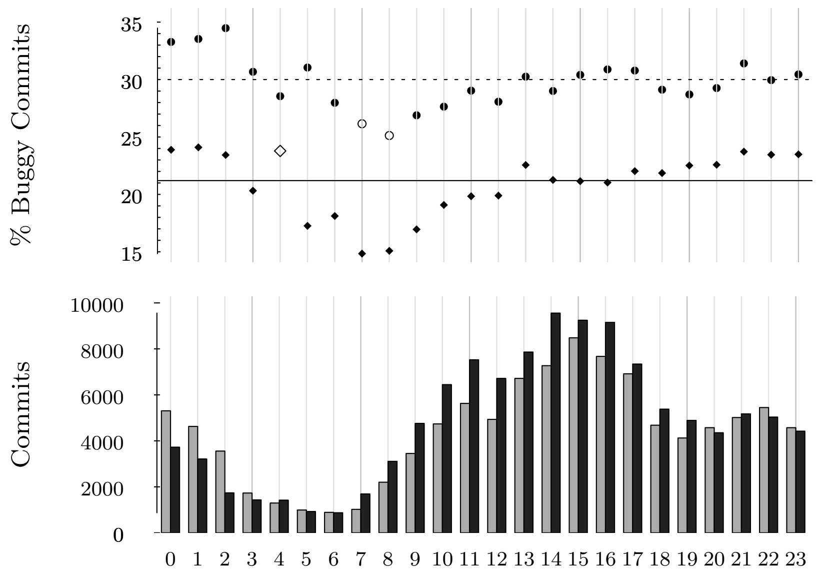


Fig. 7 Percentage of buggy commits (circles/diamonds) and total number of commits (light/dark bars) versus time-of-day for junior and senior Linux kernel developers, respectively. Circles and light bars denote junior developers, while diamonds and dark bars denote senior developers. Solid circles and diamonds indicate p-value < 0.05, while hollow circles and diamonds indicate p-value ≥ 0.05. Dashed and solid horizontal lines indicate the average bugginess percentage for junior and senior developers, respectively.

3.5 Combined Time-of-day and Seniority

Figure 7 combines data from Section 3.2 and 3.3.2 and correlates time-of-day with commit bugginess for senior and junior developers, plotted separately, for the Linux kernel[9]. We used a cutoff of 2 years to separate senior and junior developers; this cutoff divides the number of commits into two approximately-equal groups. Horizontal lines in the figure represent overall bugginess.

[9] Table 7 in the Appendix presents p-values evaluating the statistical significance of the combined seniority-and-hour commit bugginess for the Linux kernel and Xorg.

We can see that junior developers tend to do more commits between midnight and 2 AM than senior developers, who do more commits between 8 AM and 4 PM. However, there is a common trend for both: late night commits (especially between midnight and 2 AM) are more buggy and early morning commits (between 8 AM and noon) are less buggy.

Discussion This result suggests that the correlation between time-of-day versus bugginess is independent of seniority for the Linux kernel developers; it occurs for both senior and junior developers. It also shows that senior developers are much less likely to commit a bug; the average bugginess for senior Linux kernel developers is around 21%, versus 30% for junior developers.

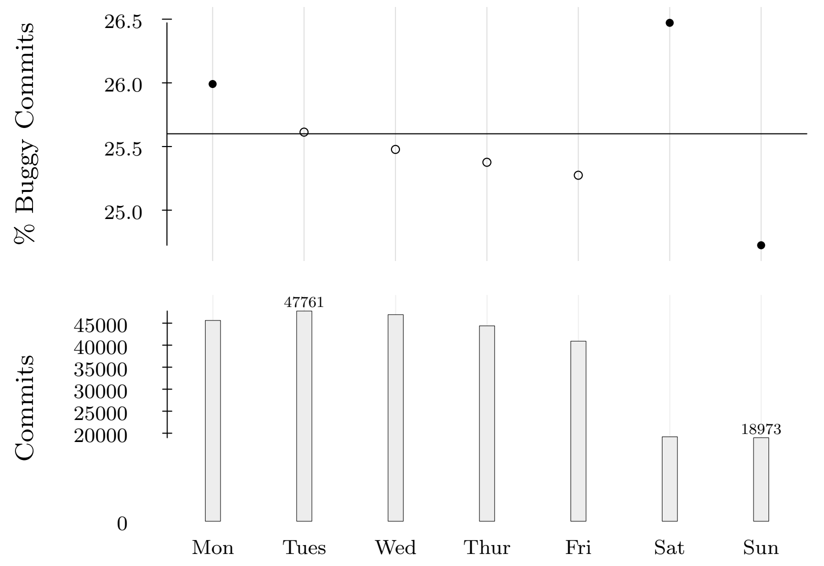
3.6 Day-of-week

Our next experiment attempted to replicate the results in [25], and correlates the day of the week of a commit with its bugginess. Figure 8 compares the day of the week with the bugginess of the commits on that day (bars), and also displays the total number of commits per day (circles)[10]. Here, the solid horizontal line presents the overall bugginess of all commits to each project.

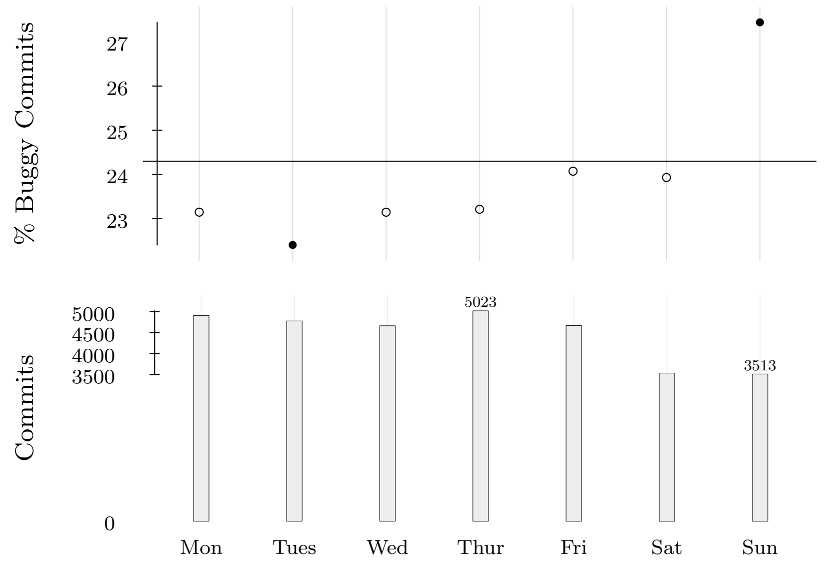
Our results, which use a disjoint set of repositories from those in [25], found about the same bugginess and number of introductions for each day of the week in the Linux kernel repository, with the lowest significant bugginess on Sunday and highest on Monday; for the PostgreSQL repository, we found a slight decrease in bugginess on Tuesday, and a noticeable increase on Sunday. These results are statistically significant with a p-value less than 0.05. For the Xorg server, no days are (statistically significantly) worse or better than average. Note that, for the Linux kernel, Saturday and Sunday each have about half as many commits as the other days of the week (commits peak on Tuesday and steadily decrease through Friday). Although Saturday commits to the Linux kernel are more likely to be buggy, the p-value for Saturday commits is 0.065, so that result is not statistically significant. For PostgreSQL and Xorg, commits fluctuate through the days of the week and decrease to about 70% of the weekday volume on the weekend.

Discussion We found that, across projects, no specific day of the week is particularly prone to bugginess, which does not agree with results from a prior study of bugs in Mozilla and Eclipse [25]. We did find that bugginess per day-of-week varied in a project-specific way, implying that bugginess prediction based on day-of-week needs to be calibrated on a per-project basis.

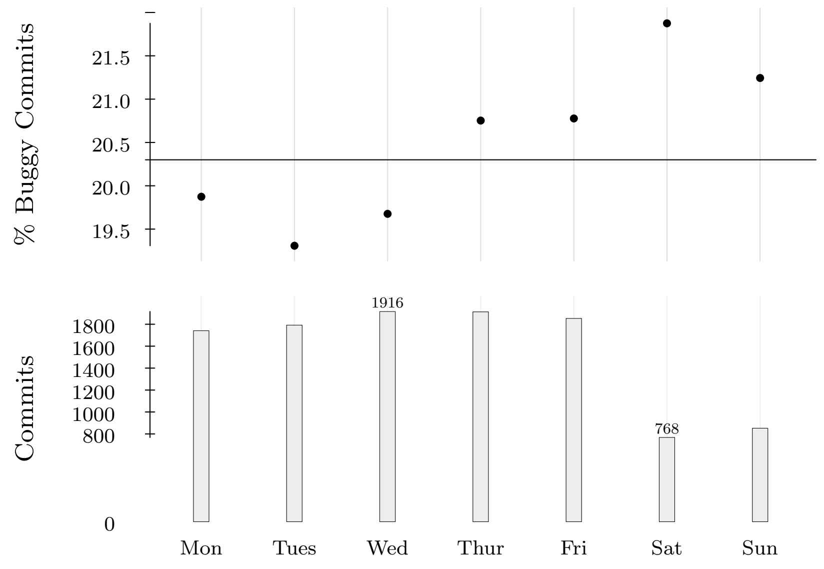
[10] Table 10 in the Appendix presents p-values evaluating the statistical significance of the per-day commit bugginess for the Linux kernel.



(a) Linux kernel



(b) PostgreSQL



(c) Xorg server

3.7 Stable Commits

Linux developers designate certain supposedly low-risk commits as stable commits. Our bugginess data enables us to empirically verify whether these commits are indeed lower-risk than arbitrary commits. Table 4 summarizes our findings on stable commits for the Linux kernel. We observe, with very high confidence, that stable commits are significantly less buggy than non-stable commits.

Discussion Stable commits are subject to a more rigid set of guidelines than the rest of the commits. We would expect these guidelines to lead to a lower bugginess rate, and our results show that the guidelines are indeed effective. Developers of error-prone portions of software could leverage this finding by aiming to mimic the properties of stable commits (when appropriate): they might benefit from committing smaller patches and reviewing them carefully. Alternatively, developers could choose to work on lower-risk commits at high-bugginess times of day.

Type	**Percentage of buggy commits**	**Total commits**	**p-values**
Stable commits	20.1%	10,518	1.43E-40
Non-stable commits	25.8%	253,262	0.00421

Table 4 Linux kernel stable commits.

3.8 Validation

We estimated the precision and recall of our technique for identifying bug-fixing commits on both projects. As our algorithm for identifying the associated bug-introducing commits was a straightforward application of git blame, we did not systematically verify its performance. (A brief manual inspection of bug-introducing commits did not reveal any anomalies.) For both projects, we randomly sampled 200 commits and manually verified the results. Table 5(a) and 5(b) summarize our findings.

We evaluated the precision—the proportion of identified bug-fixing commits which do indeed fix bugs—and found that, the precision of this heuristic for identifying bug-fixing commits is 87% for the Linux kernel, 86% for PostgreSQL, and 75% for Xorg. Misclassifications included: 1) a commit message which fixed a merge commit was classified as a fix; 2) apparently garbled commit messages which included the keyword “fix” for no good reason; 3) changes which were reverted (in the alleged “fix”) but then re-added in a later version; 4) poor uses of version control systems which included many different changes

		Predicted Fix	Predicted ¬Fix
Actual	Fix	48	18
	¬Fix	7	127

(a) Linux kernel

		Predicted Fix	Predicted ¬Fix
Actual	Fix	30	12
	¬Fix	5	153

(b) PostgreSQL

		Predicted Fix	Predicted ¬Fix
Actual	Fix	30	17
	¬Fix	10	143

(c) Xorg

Table 5 Confusion matrices.

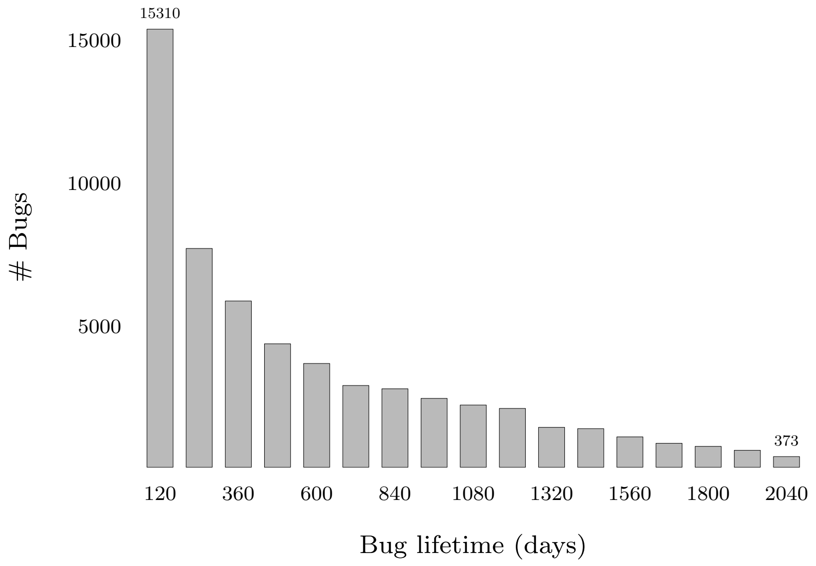
in a single commit, including a fix as a small part of the commit; and 5) refactoring changes, which moved or renamed functions; these could arguably be considered to be fixes to a buggy initial design.

Our recall—the proportion of bug-fixing commits in the entire sample that our technique identifies—is 73% for the Linux kernel, 71% for PostgreSQL, and 64% for Xorg. We could improve the precision and recall by using more advanced techniques for identifying bug-fixing commits, such as those by Tian et al. and by Wu et al [28,30].

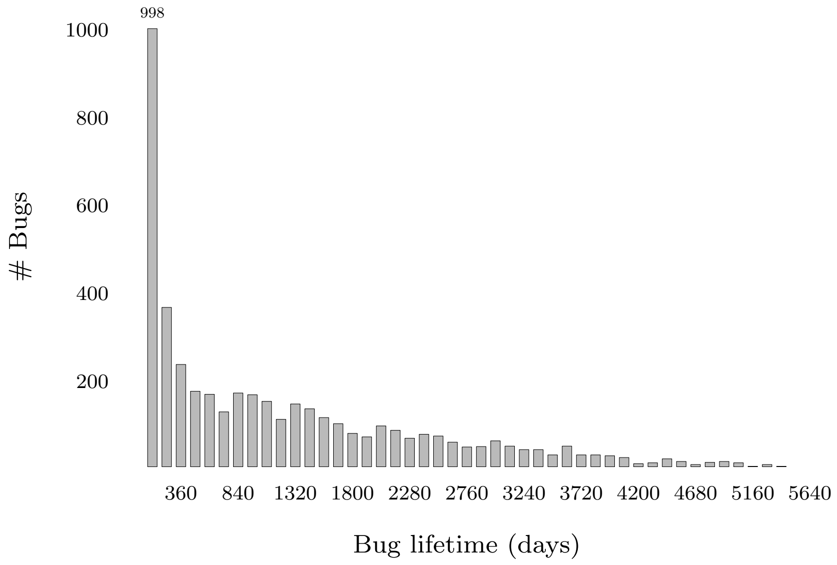
3.9 Bug Lifetimes

The recall numbers that we report in Section 3.8 only include bugs that have been fixed by a later commit. Some bugs in the repository will not have been fixed yet simply because they are too new. We define bug lifetime as the amount of time elapsed between the bug-introducing commit and its bug-fixing commit. Estimating bug lifetimes thus addresses a threat to the computed recall: it gives us confidence that any sufficiently-old bug will have likely been fixed.

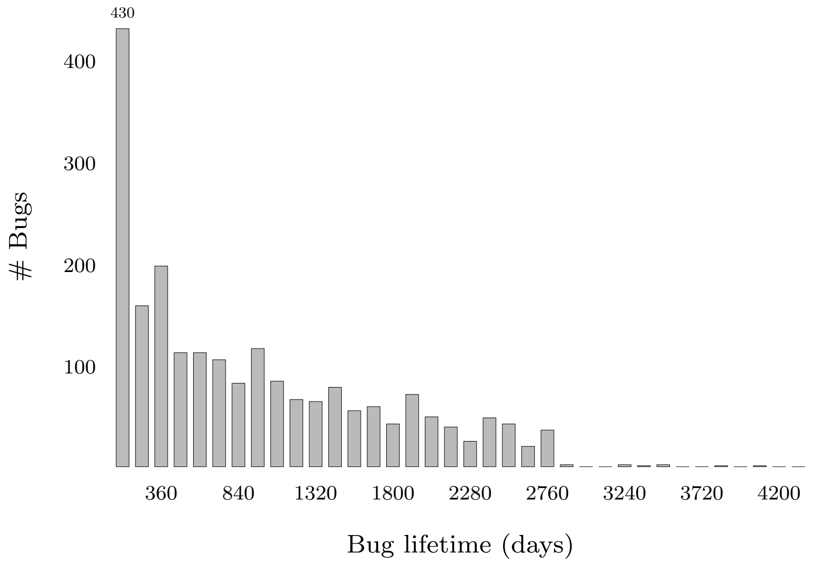
Figure 9 shows bug lifetimes for the Linux kernel and PostgreSQL, grouped in 120 day intervals. We found the average bug lifetime for the Linux kernel is 1.38 years with a standard deviation of 1.35 years. The average bug lifetime for PostgreSQL is 3.07 years with a standard deviation of 3.19 years. Finally, the average bug lifetime for Xorg server is 2.36 years with a standard deviation of 2.15 years. Note that the distribution of bug lifetimes is similar for all projects; many bugs are fixed within a 120 day period and the overall lifetime appears to decrease exponentially. We found that the sources of PostgreSQL's high-lifetime bugs included race conditions, incorrect calculations and rare corner cases.



(a) Linux kernel



(b) PostgreSQL



(c) Xorg server

4 Related Work

We survey related work that studied the following commit characteristics in the context of commit bugginess: day of week; developer experience and distributed software development; and bug lifetime.

Day of Week of Commits The most closely related work to ours, Śliwerski et al [25], studied the day of the week of commits for two totally different projects, Eclipse and Mozilla, and found that the commits on Fridays are buggiest. This paper differs from the work of Śliwerski et al in the following three key aspects. Firstly, we investigated how the commits' time of day correlates with the bugginess of commits, which has not been studied before, to the best of our knowledge. Secondly, we studied developer characteristics, including correlations between commit bugginess and developers' commit frequency, as well as developers' experience, which the previous work did not consider. Finally, we used different data collection techniques. Specifically, we did not rely on the link between a commit and a bug report to extract bug-fixing commits, which enabled us to study software for which such links are not maintained or not well maintained by the developers. For example, we found that only 2.3% of the Linux kernel's bug-fixing commits are linked to a bug report, by manually examining a random sample of our bug-fixing commits. While using links between bug reports and bug commits may increase the precision of extracting bug-fixing commits, our results demonstrate that high precision can be obtained without using such links: the precision of our bug-fixing commit extraction techniques are 87% for the Linux kernel, 86% for PostgreSQL, and 75% for Xorg.

Developer Experience and Distributed Software Development Mockus and Weiss studied the correlation between developer experience and code bugginess at the maintenance request (MR) level and found that changes by more experienced developers are less likely to be buggy [19]. Rahman and Devanbu [24] studied the correlation at a finer granularity: groups of continuous lines in a commit (referred to as hunks). They found that greater experience was correlated with more-buggy code for Gimp and Nautilus, but that there was no clear correlation between developer experience and code bugginess for Apache and Evolution. We examined code at a different granularity level—code commits, and found no clear correlation between developer experience and commit bugginess for the Linux kernel or Xorg, while commits by more experienced developers in PostgreSQL are less likely to be buggy. Our study, on a disjoint set of projects, confirms this observation that the correlation between developer experience and commit bugginess is project dependent. While one MR can involve changes to multiple files by multiple authors, a single commit has one author; so is a hunk. Other previous work studied the correlation between the number of authors and their weight of authorship with code quality [5, 29]. Our present work goes beyond previous studies, as we investigate other time-based characteristics such as commit time and author commit classification, as well as how they correlate with developer experience.

Several previous studies sought to understand how distributed software development affects code quality [3,4,21,26] in open source and commercial software. While the three projects we studied are open-source and developed in a distributed fashion, the goal of this paper differs from those studies—we aim to understand the correlation between code bugginess and time-based characteristics of commits, e.g., commits' time of day, commits' day of week, developers' experience, and developers' commit frequencies, etc.

Bug Lifetime Engler et al [7] examined the bug lifetime of the Linux kernel in 2001. Our study on the bug lifetime complements theirs by analyzing recent commits to the Linux kernel from 2005-2010. Kim and Whitehead [15] examined the bug lifetimes in PostgreSQL. Neither of the two previous studies investigated other time-based characteristics such as commit time and author experience.

5 Conclusions and Future Work

We have analyzed over 80,000 bug-fixing commits in the Linux kernel, PostgreSQL, and the Xorg server (three large and widely-used open-source software projects) to study the correlation between commit correctness with several time-based characteristics, such as the time-of-day of commits, the day-of-week of commits, developer seniority, developers' commit frequency, and whether a commit was marked "stable". We presented several interesting findings, including: (1) late-night commits (between midnight and 4:00 AM) are collectively buggier than average, while morning commits (8:00 AM–noon) are collectively less buggy, suggesting that developers may want to double-check late-night commits before committing, and that it may be beneficial for the version control system to warn the developers of late-night commits to improve software reliability; (2) the bugginess of commits per day-of-week varies across different software projects, implying that the bugginess prediction based on the day-of-week of commit metric needs to vary on a project-by-project basis; (3) developers who commit to a project on a daily basis write fewer buggy commits for that project, indicating that we may want to promote the practice of daily committing developers code-reviewing other developers' commits; and (4) commits marked "stable" are indeed less buggy than average. We believe such results are valuable to the software engineering community and software developers.

In the future, we would like to study commit times with respect to individual developers to understand, for example, whether a developer's commits outside of his/her normal committing hours are buggier than average for that developer. In addition, we plan to study more software projects written in different programming languages to further understand how social characteristics affect commit correctness. As there may be interesting correlations between a commit's evolution and its code quality, we intend to study such correlations in the future.

References

1. Y. Benjamini and D. Yekutieli. The control of the false discovery rate in multiple testing under dependency. *The Annals of Statistics*, 29(4):1165–1188, Aug 2001.
2. C. Bird, A. Bachmann, E. Aune, J. Duffy, A. Bernstein, V. Filkov, and P. Devanbu. Fair and balanced?: bias in bug-fix datasets. In *ESEC/FSE '09: Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 121–130, 2009.
3. C. Bird and N. Nagappan. Who? where? what? examining distributed development in two large open source projects. In *Proceedings of the international working conference on Mining software repositories*, 2012.
4. C. Bird, N. Nagappan, P. T. Devanbu, H. Gall, and B. Murphy. Does distributed development affect software quality? In *ICSE*, pages 518–528, 2009.
5. C. Bird, N. Nagappan, B. Murphy, H. Gall, and P. Devanbu. Don't touch my code!: examining the effects of ownership on software quality. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, ESEC/FSE '11, pages 4–14, 2011.
6. A. Chou, J. Yang, B. Chelf, S. Hallem, and D. R. Engler. An empirical study of operating system errors. In *Symposium on Operating Systems Principles*, pages 73–88, 2001.
7. D. Engler, D. Y. Chen, S. Hallem, A. Chou, and B. Chelf. Bugs as deviant behavior: A general approach to inferring errors in systems code. *SIGOPS OSR*, 35(5):57–72, 2001.
8. J. Eyolfson, L. Tan, and P. Lam. Do time of day and developer experience affect commit bugginess? In *MSR*, 2011.
9. T. Graves, A. Karr, J. Marron, and H. Siy. Predicting fault incidence using software change history. *IEEE TSE*, 2000.
10. L. Guo, Y. Ma, B. Cukic, and H. Singh. Robust prediction of fault-proneness by random forests. In *ISSRE*, 2004.
11. R. Haas. So, why isn't PostgreSQL using Git yet? http://rhaas.blogspot.com/2010_09_01_archive.html.
12. A. E. Hassan. Predicting faults using the complexity of code changes. In *ICSE*, pages 78–88, 2009.
13. I. Herraiz, J. M. González-Barahona, G. Robles, and D. M. Germán. On the prediction of the evolution of libre software projects. In *ICSM*, pages 405–414, 2007.
14. S. Kim, E. J. Whitehead, Jr., and Y. Zhang. Classifying software changes: Clean or buggy? *IEEE Trans. Softw. Eng.*, 34(2):181–196, Mar. 2008.
15. S. Kim and E. Whitehead Jr. How long did it take to fix bugs? In *MSR*, pages 173–174, 2006.
16. S. Kim, T. Zimmermann, K. Pan, and E. Whitehead. Automatic identification of bug-introducing changes. In *ASE*, pages 81–90, 2006.
17. A. Meneely, L. Williams, W. Snipes, and J. Osborne. Predicting failures with developer networks and social network analysis. In *SIGSOFT/FSE*, pages 13–23, 2008.
18. T. Menzies, Z. Milton, B. Turhan, B. Cukic, Y. Jiang, and A. B. Bener. Defect prediction from static code features: current results, limitations, new approaches. *ASE*, 2010.
19. A. Mockus and D. M. Weiss. Predicting risk of software changes. *Bell Labs Technical Journal*, 5(2):169–180, 2000.
20. A. Mockus, D. M. Weiss, and P. Zhang. Understanding and predicting effort in software projects. In *ICSE*, 2003.
21. N. Nagappan, B. Murphy, and V. R. Basili. The influence of organizational structure on software quality: An empirical case study. In *ICSE*, pages 521–530, 2008.
22. T. Ostrand, E. Weyuker, and R. Bell. Predicting the location and number of faults in large software systems. *IEEE TSE*, 31(4), 2005.
23. R. Purushothaman and D. E. Perry. Toward understanding the rhetoric of small source code changes. *IEEE TSE*, 2005.

24. F. Rahman and P. Devanbu. Ownership, experience and defects: a fine-grained study of authorship. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 491–500, 2011.
25. J. Śliwerski, T. Zimmermann, and A. Zeller. When do changes induce fixes? In *MSR*, pages 24–28, 2005.
26. D. Spinellis. Global software development in the FreeBSD project. In *GSD*, pages 73–79, 2006.
27. M. M. Swift, B. N. Bershad, and H. M. Levy. Improving the reliability of commodity operating systems. In *SOSP*, pages 207–222, 2003.
28. Y. Tian, J. Lawall, and D. Lo. Identifying linux bug fixing patches. ICSE'12.
29. E. J. Weyuker, T. J. Ostrand, and R. M. Bell. Do too many cooks spoil the broth? using the number of developers to enhance defect prediction models. *Empirical Software Engineering*, 13(5):539–559, 2008.
30. R. Wu, H. Zhang, S. Kim, and S.-C. Cheung. Relink: recovering links between bugs and changes. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, ESEC/FSE '11, pages 15–25, 2011.
31. T. Zimmermann and N. Nagappan. Predicting defects using network analysis on dependency graphs. In *ICSE*, 2008.
32. T. Zimmermann, R. Premraj, and A. Zeller. Predicting defects for eclipse. In *PROMISE*, 2007.
33. T. Zimmermann and P. Weißgerber. Preprocessing CVS data for fine-grained analysis. In *MSR*, pages 2–6, 2004.

Appendix

Hour	**Linux kernel**	**PostgreSQL**	**Xorg server**
0	8.67E-20	0.218	0.0875
1	3.12E-20	0.00306	0.0319
2	1.70E-21	0.00272	0.163
3	0.261	0.0153	0.260
4	0.251	0.0340	0.291
5	0.150	0.0944	0.373
6	0.0201	0.405	0.297
7	3.40E-18	0.408	0.205
8	1.32E-23	0.0511	0.00355
9	5.42E-25	6.39E-6	0.139
10	4.41E-12	0.00294	0.104
11	5.90E-7	0.00132	0.314
12	1.75E-4	3.64E-4	0.0677
13	0.192	0.174	0.129
14	0.150	0.166	0.311
15	0.336	0.334	0.398
16	0.475	7.37E-5	0.106
17	0.0765	0.0312	0.514
18	0.503	0.00187	0.207
19	0.379	0.00133	0.485
20	0.0728	0.00169	0.396
21	2.34E-8	0.343	0.408
22	0.00206	0.00479	0.150
23	4.00E-4	0.100	0.0327

Table 6 Bugginess p-values, per hour, for the Linux kernel, PostgreSQL, and Xorg.

Hour	Junior Linux kernel	Senior Linux kernel	Junior Xorg	Senior Xorg
0	1.66E-37	1.59E-02	0.44	0.045
1	5.68E-35	4.35E-02	0.083	0.13
2	2.33E-33	2.93E-02	0.17	0.37
3	4.93E-07	3.31E-06	0.33	0.36
4	5.77E-03	8.21E-02	0.33	0.44
5	3.89E-05	1.79E-09	0.39	0.57
6	4.45E-02	1.80E-07	0.30	0.10
7	3.08E-01	2.75E-26	0.35	0.029
8	3.85E-01	1.24E-44	0.11	0.0096
9	2.55E-02	8.32E-45	0.36	0.15
10	2.73E-04	1.04E-33	0.38	0.10
11	4.74E-10	1.76E-30	0.15	0.0511
12	1.27E-05	7.39E-27	0.48	0.019
13	2.66E-19	1.95E-09	0.28	0.18
14	2.52E-12	9.38E-22	0.19	0.49
15	2.57E-25	3.75E-22	0.10	0.25
16	3.20E-27	4.18E-23	0.13	0.27
17	6.58E-24	6.37E-12	0.059	0.042
18	5.78E-09	5.83E-10	0.016	7.36E-05
19	9.70E-07	1.23E-06	0.12	0.15
20	2.43E-09	7.13E-06	0.13	0.25
21	1.11E-21	2.49E-03	0.37	0.24
22	2.62E-14	6.53E-04	0.26	0.23
23	1.14E-14	1.56E-03	0.03	0.26

Table 7 Bugginess p-values by hour and seniority for the Linux kernel and Xorg.

Classification	**Linux kernel**	**Xorg**
Daily	4.78E-20	0.182
Weekly	7.34E-48	
Monthly	7.69E-52	
Other	0.0104	
Single	0.0144	
Non-daily		0.00195

Table 8 Bugginess p-values, by classification, for the Linux kernel and Xorg.

Seniority in days	**Linux kernel**	**PostgreSQL**	**Xorg**
120	1.82E-122	3.18E-6	4.03E-7
240	6.18E-67	6.47E-8	5.81E-4
360	5.43E-41	2.89E-10	0.0130
480	4.08-20	2.85E-5	0.241
600	1.04E-21	1.03E-9	2.31E-4
720	1.40E-9	1.28E-9	0.425
840	2.89E-10	5.12E-8	1.36E-4
960	0.208	1.32E-15	0.0399
1080	0.0522	1.13E-6	0.0406
1200	3.06E-6	0.00552	0.388
1320	4.55E-17	0.0182	0.357
1440	2.32E-39	0.230	1.23E-4
1560	8.50E-60	7.33E-5	0.0645
1680	5.23E-86	0.0155	0.0135
1800	5.54E-143	0.166	0.280
1920	1.29E-164	1.69E-11	0.431
2040	1.90E-264	0.0509	1.88E-5
2160		5.37E-4	0.299
2280		0.358	3.57E-5
2400		0.0915	0.133
2520		0.0278	0.327
2640		0.0574	0.313
2760		0.157	0.193
2880		0.397	5.03E-7
3000		1.13E-6	0.214
3120		0.0111	0.0767
3240		0.0434	0.472
3360		1.62E-10	0.506
3480		5.17E-4	0.0172
3600		7.61E-8	0.369
3720		4.71E-8	0.00163
3840		2.41E-15	0.0211
3960		1.51E-19	6.63E-6
4080		5.84E-18	0.00134
4200		3.28E-28	0.403
4320		8.60E-15	
4440		2.24E-14	
4560		8.44E-9	
4680		1.85E-8	
4800		0.0501	
4920		1.71E-6	
5040		2.58E-12	
5160		0.0118	
5280		0.250	

Table 9 Bugginess p-values by seniority for the Linux kernel, PostgreSQL, and Xorg.

Day	Linux kernel	PostgreSQL	Xorg
Mon	0.018	0.16	0.34
Tues	0.40	0.014	0.15
Wed	0.34	0.17	0.26
Thurs	0.19	0.19	0.32
Fri	0.093	0.31	0.32
Sat	0.020	0.41	0.15
Sun	0.0042	2.5E-07	0.26

Table 10 Bugginess p-values by day for the Linux kernel, PostgreSQL, and Xorg.