Do Time of Day and Developer Experience Affect Commit Bugginess?

Jon Eyolfson
jeyolfso@uwaterloo.ca

Lin Tan
lintan@uwaterloo.ca

Patrick Lam
p.lam@ece.uwaterloo.ca

University of Waterloo
200 University Avenue West
Waterloo, Ontario, Canada N2L3G1

ABSTRACT

Modern software is often developed over many years with hundreds of thousands of commits. Commit metadata is a rich source of social characteristics, including the commit's time of day and the experience and commit frequency 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 social characteristics and its "bugginess"; such results can be very 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, and the experience and commit frequency of the commit authors. We survey two widely-used open source projects: the Linux kernel and PostgreSQL.

Our main findings include: (1) commits submitted between midnight and 4 AM (referred to as late-night commits) are significantly buggier and commits between 7 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; and (3) the bugginess of commits versus day-of-week varies for different software projects.

Categories and Subject Descriptors

D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement; D.2.9 [**Software Engineering**]: Management

General Terms

Human Factors, Management, Measurement, Reliability

Keywords

Bug Detection, Empirical Study, Source Control System

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
MSR '11, May 21-22, Waikiki, Honolulu, Hawaii, USA
Copyright 2011 ACM 978-1-4503-0574-7/11/05 ...$10.00.

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, 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 social characteristics from older, more homogeneously-developed projects; these characteristics can best be measured by going beyond the code itself and into the social characteristics of the code.

Software social 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 submitted 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 [3, 4, 6, 8, 12, 13, 15, 17, 23, 24], which predict buggy locations based on code complexity, code locations, the amount of in-house testing, historical data, socio-technical networks, etc. A second interesting question is whether more experienced developers are more or less likely to write buggy commits.

Contributions.

In this paper, we study the social characteristics of modern software development to understand the correlation between these social 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 and PostgreSQL, which have 222,332 and 31,098 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, and the experience and commit frequency of the commit authors. 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 [19]. 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 [19] 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, like Mozilla, it does not work for software whose commit messages rarely contain links to bug reports, like 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 86%-87% in identifying bug-fixing commits (Section 3).

Our major findings are summarized below (§ denotes the section where the finding and its implications are discussed):

- **Finding 1 (§3.1):** About a quarter (23.7–25.5%) of all the commits in the Linux kernel and PostgreSQL are labelled buggy—they require further developer activities to fix them.
- **Finding 2 (§3.2):** Commits that are checked into the software repository around midnight (between 0:00–4:00 AM) are more likely to be incorrect than average, while commits in the morning (7:00 AM–noon) are more likely to be correct. The result indicate that developers may want to double-check the code they write for these late-night commits (0:00–4:00 AM). It may also be beneficial for the version control system to warn the developers of late-night commits to improve software reliability.
- **Finding 3 (§3.3):** Developers who commit to the repository on a daily basis write less-buggy commits, while developers who appear to work on a project as part of their day-job are more likely to produce bugs, indicating that we may want to promote the practice of daily-committing developers reviewing other developers' commits.
- **Finding 4 (§3.5):** In contrast to a prior finding that Friday commits are buggier [19], our results on the Linux kernel and PostgreSQL show that the bugginess differences of commits that are checked in on different days of week are small. We found that the bugginess per day-of-week for commits varies for different software projects, implying that bugginess prediction based on day-of-week may need to be calibrated on a per-project basis.

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 that 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 Methodology

Following [19], 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 which were responsible for the previous (buggy) version of each of the changed lines.

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

Figure 1: An example bug-fixing commit

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

Figure 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);
+     }
```

Figure 3: Associated bug-introducing commit for the example

We describe each of these steps in more detail. First, to detect bug-fixing commits, c, we searched the commit messages for the keyword "fix" (as do [18]). 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, and our results show that the precision of this heuristic for identifying bug-fixing commits is 86%–87%. Next, we computed diffs for each file changed in c, omitting comments, and recorded the line numbers L_c which 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, which addresses some inaccuracies that exist in other version control systems such as CVS and Subversion, as discussed by Kim et al [11].

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) 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 com-

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

mit 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 which 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 which 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 which 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/email pair); adjusted local time (as described 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 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 which 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 to a particular project, and author experience at commit time for each patch, based on the elapsed time between that author's first commit to the project and the commit time.

The *author commit frequency* classification describes the author's most-common frequency between two consecutive commits by that author: 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. As a subclass of the daily committer classification, we also use a heuristic to identify committers who appear to work on the repository as part of their day job, namely those for which 85% of commits are between 8 AM and 4 PM Monday to Friday (see Section 2.3 for more discussion on this heuristic). This is to separate developers who must work on the code from those who are motivated by interest.

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

Time zone adjustments.

All PostgreSQL commits prior to September 18, 2010 (when the project switched to the Git version control system) 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 used the Python time zone utilities to convert 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. 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.

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 each project and manually verifying whether or not they are indeed fixes. False positives and false negatives contribute to precision and recall; Section 3 presents 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.

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 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 repository 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[2] 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.

We ignore Git merge commits, which record metadata about integration between different maintainers' trees. PostgreSQL does not use merge commits. Merge commit will never introduce or fix bugs; that will be attributed to one of the specific commits being merged.

Because the PostgreSQL repository was converted from CVS to Git, the accuracy of the conversion is another potential threat to

[2]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.

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 [5]. The existence of these customizations lead us to believe that the conversion was performed with care, mitigating threats to validity from repository corruption. Also, note that the conversion to Git obviates the need to mine transactions from CVS histories, as in [25].

We excluded commits that merge different branches from our study, because including these merging commits would double-count a commit, once for the original commit and once for the merging commit. Note that merging commits account for only 6.25% of all commits.

Threats due to imperfect recall.

Because our methodology only identifies bugs which have later been corrected, it will omit recently-introduced bugs, as well as longer-running bugs which have not yet been corrected. Furthermore, our recall of 71% for PostgreSQL and 73% for the Linux kernel means that our methodology does not manage to identify some of the bug fixes in the repository histories.

While these threats imply that our results will omit some bugs, we do not believe that this omission affects our validity. There is no reason to believe that there are important differences in the characteristics we measure for 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.

Threats to commit characteristics.

The next family of threats concerns commit characteristics, including the attributed time and author information for a commit. These threats do not apply to the Linux kernel repository, as the Linux kernel community uses the merge functionality of Git to preserve commit metadata from external contributors. The implications of Linux's 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.

Most PostgreSQL patches are committed by someone other than the author of the patch: the set of PostgreSQL committers is quite small, so external PostgreSQL commits will always be attributed to a member of the PostgreSQL team. 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. Third-party commits add uncertainty about when a patch was originally written, and confound our data about patch authors. We believe, however, that even for PostgreSQL, it is reasonable to ascribe responsibility for the patch to its committer, who would presumably review a patch before committing it.

Developers work on a patch over a possibly discontiguous time interval. However, 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.

Finally, we discuss threats to our author classification. Some Linux kernel 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 day-job classification for authors is based on the fixed hours of 8 AM–4 PM and Monday–Friday days of the week. Many software companies have flexible hours; however, we expect that developers who are directed to work on a project as their day job would still have the bulk of their commits between those hours. The day-job classification applies only to the Linux kernel committers; no PostgreSQL committers satisfy our day-job criterion. Note that this classification attempts to measure whether developers are working on the project of their own volition or not: some Linux committers are paid to work on Linux, but commit at all hours of the day. We believe that, in such cases, the committer has substantial autonomy in choosing what to work on, and is not being directed to work on specific parts of the project. Similarly, even though most of the core PostgreSQL team have PostgreSQL-related employment, we believe that they are not directed to work on PostgreSQL, which would account for the observation that most of their commits to PostgreSQL occur outside normal working hours.

Our author experience only counts the participation of an author to a particular project; a highly-experienced developer may be classified as an inexperienced developer for one of our projects due to a short history of contributions to that project. Therefore, the author experience in this paper should be interpreted as the author's experience with the target project. It would be possible to survey developers' participation across multiple projects by studying the public record, but such a study is beyond the scope of this paper.

3. RESULTS

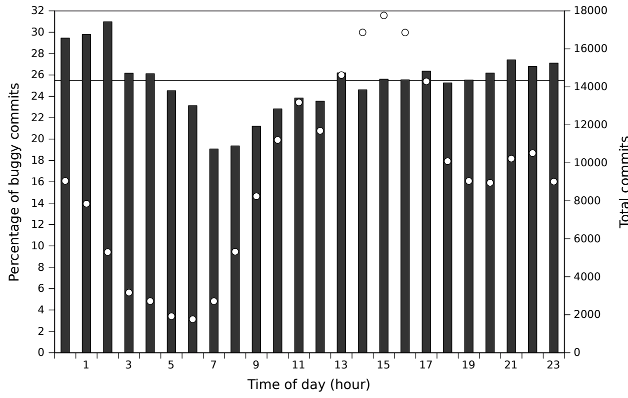
In this section, we present the results obtained from carrying out our methodology, and discuss some of the implications of our results. Most of our results investigate the effect of an independent variable (time-of-day and developer experience/frequency classifications) on the likelihood of a commit to be a bug-introducing commit, or *bugginess*. We also describe our findings with respect to the day of the week, which allows us to compare our results to those in [19]. We also discuss our finding that some bug-fixing commits only changed comments. Finally, we explain the precision and recall of our methodology and how we computed these figures.

3.1 Project Characteristics

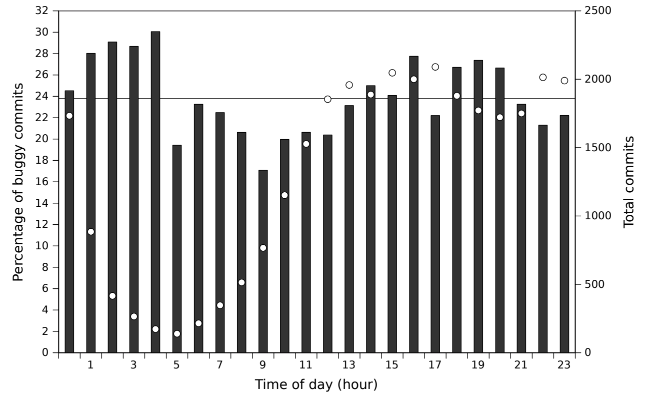
We chose two large open-source software repositories for our investigations: Linus Torvalds's mainline Linux kernel and PostgreSQL. Table 1 summarizes the characteristics of our repositories. The row "lines of code" refers to the current size of the code in the repository. The row "# bug-introducing" shows that 23.7–25.5% of the commits are buggy, which is slightly lower than the previously reported figure of nearly 40% for a commercial switching system [18]. Note that the PostgreSQL repository was carefully converted from CVS using `cvs2git` in September 2010. We discussed the quirks of the PostgreSQL repository in Section 2.

3.2 Time-of-day

Figure 4 presents our results correlating the time-of-day of a commit with its bugginess. 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; bars shorter than the line indicate that commits at that hour

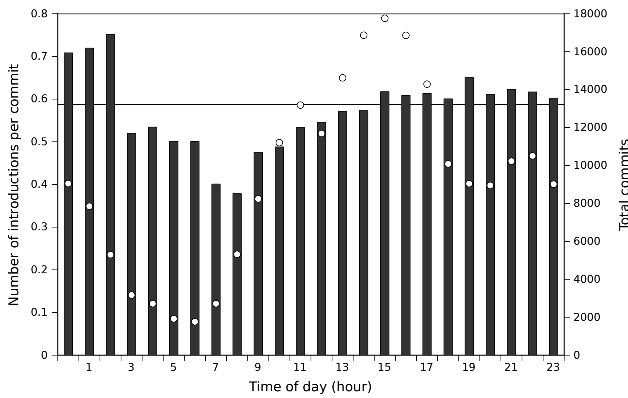


(a) Linux kernel

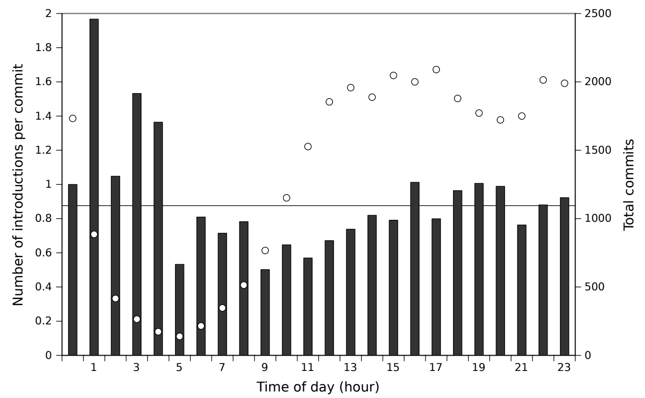


(b) PostgreSQL

Figure 4: Percentage of buggy commits (bars) and total number of commits (circles) versus time-of-day



(a) Linux kernel



(b) PostgreSQL

Figure 5: Subsequent bug fixes per commit (bars) and total commits (circles) versus time-of-day

	Linux kernel	**PostgreSQL**
First commit	April 16, 2005	July 9, 1996
Cloned	November 21, 2010	January 24, 2011
Lines of code	over 5 million	over 750,000
Number of authors	8,594	34
Number of commits	222,332	31,098
# bug-introducing	56,681 (25.5%)	7,366 (23.7%)
# bug-fixing	57,028	4,399

Table 1: Characteristics of the Linux kernel and PostgreSQL repositories.

were less likely to be buggy, while bars taller than the line indicate hours with more-buggy commits. The graphs also contain the raw number of commits at each hour, indicated by circles.

Figure 4, which summarizes bugginess percentages, shows a noticeable increase in the amount of commits which introduce a bug between 00:00 (midnight) and 04:00 (4 AM). After 04:00, commits tend to be less buggy than average, gradually increasing until noon. In the Linux kernel, commits between noon and midnight fluctuate around the average bugginess level, while the PostgreSQL commits are generally above the average bugginess level between 16:00 (4 PM) and 20:00 (8 PM), and then below the average bugginess level between 20:00 (8 PM) and 00:00 (midnight). Figure 5, which shows the number of subsequent bug-fixing commits for each bug-introducing commit (indicating how difficult it was to correct a bug), follows the trends from Figure 4. Note that even the smallest total number of commits, for any hour, is 139 for PostgreSQL (and an order of magnitude higher for the Linux kernel), so that all of the depicted bug introduction rates are meaningful.

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 circles in Figure 4. This suggests that the fact that a commit is bug-fixing is independent of its other characteristics.

Table 2 presents p-values evaluating the statistical significance of the per-hour commit bugginess for Linux and PostgreSQL. The null hypothesis is that each hour has the same probability as the overall bugginess for each project. Typically, a p-value less than 0.05 indicates that the null hypothesis is rejected, and the corresponding result is considered to be statistically significant. Therefore, our p-value results show that the differences in bugginess of different hours are statistically significant. Concretely, the p-values allow us to conclude that commits introduced between 00:00 (midnight) and 04:00 (4 AM) are buggier than average with statistical significance.

	P-value	
Hour	Linux kernel	PostgreSQL
0	9.62E-18	0.245
1	4.59E-18	0.00205
2	1.62E-19	0.00748
3	0.197	0.0382
4	0.232	0.0348
5	0.173	0.133
6	0.0116	0.464
7	1.56E-15	0.308
8	2.62E-26	0.0494
9	4.54E-20	3.80E-6
10	3.25E-11	0.00108
11	6.88E-6	0.00179
12	6.13E-7	2.63E-4
13	0.0255	0.258
14	0.00447	0.114
15	0.366	0.386
16	0.436	2.40E-5
17	0.00929	0.0456
18	0.301	0.00176
19	0.471	2.70E-4
20	0.0695	0.00314
21	4.91E-6	0.311
22	0.00115	0.00433
23	2.42E-4	0.0509

Table 2: Linux kernel and PostgreSQL bugginess p-values

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 3.4 imply that this correlation holds for both inexperienced and experienced developers. Our results do suggest that developers, being aware of such a correlation, may want to double-check code 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.

Our results also suggest that tired developers (midnight–4 AM) are more likely to miss corner cases in a pre-commit review (for PostgreSQL) or while finalizing their patch (for the Linux kernel). Furthermore, we can observe that commits before noon are least likely to be bug-introducing; perhaps committers are most careful in those hours.

3.3 Developer Characteristics

We next present our findings with respect to developers’ commit frequency and experience. Developers’ commit frequency summarizes the frequency of a developer’s contributions to a project, while developer experience 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. This information is only interesting for the Linux kernel, as almost all (28/34) of PostgreSQL’s committers are daily. We computed the bugginess rates for each of these classes of developers and plot author classification versus bug-introduction percentage in Figure 6. The graph also presents the number of commits by each class. Note that the Linux kernel has 49 day job authors, who provide quite a few of the total commits, 801 daily authors, who account for the overwhelming majority of commits, 238 weekly, 288 monthly, 3562 other (less than 20 commits and more than 1 commit), and 3664 single-commit authors.

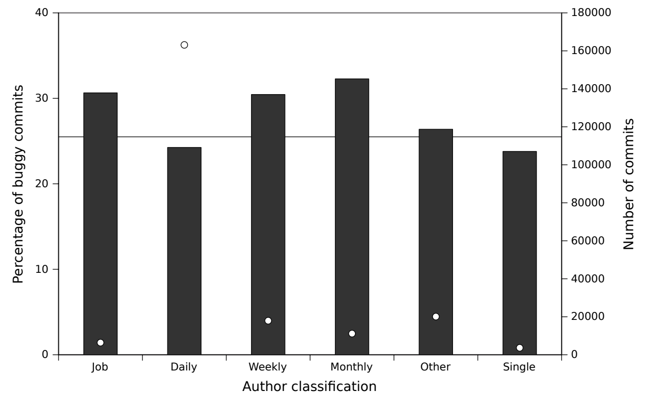


Figure 6: Linux percentage of buggy commits (bars) and number of commits (circles) versus author classification

Our results show that the Linux kernel developers who commit changes daily, but not as their day job, 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). The day job, weekly, and monthly committers all produce slightly more bug-introducing commits than average.

Discussion.

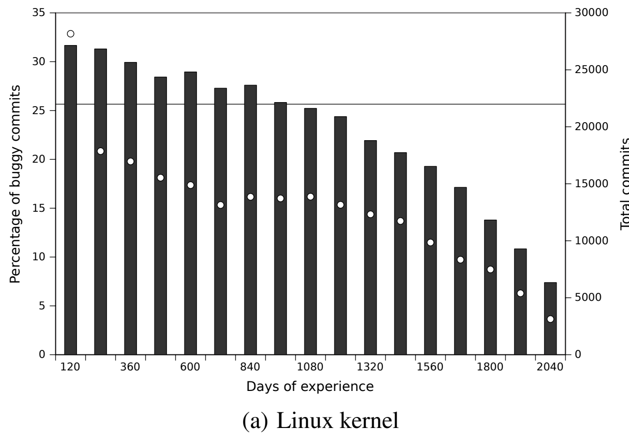
A possible cause for the difference between day-job and daily committers is that day-job developers might be required to make changes by their employers, while the daily developers are motivated purely by interest, and unlikely to be pressured to fix bugs on any particular schedule.

3.3.2 Developer Experience

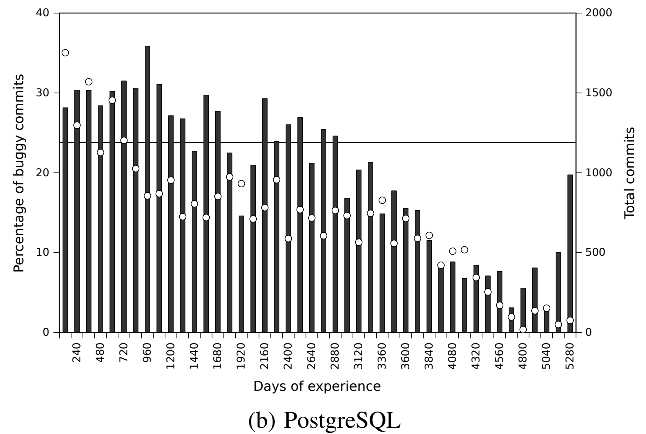
Figure 7 compares author experience at time of commit to the bugginess of the commit. It also presents the total number of commits by author experience. Note that a plurality of Linux commits are by authors with fewer than 120 days of experience. Both the Linux and PostgreSQL data show that bugginess decreases with increased author experience. For Linux, authors with at least 960 days of experience tend to produce commits that are less buggy than average, while the similar point for PostgreSQL occurs at 3000 days. The PostgreSQL data also shows a spike at the right, which implies surprisingly high bugginess in code recently committed by the original authors.

Discussion.

Our data shows that, in general, the more experienced the developers are, the less likely that their commits are buggy. Without further data, this correlation does not prove that the developer experience caused more experienced programmers’ commits to be less buggy. While we believe the above causation to hold, other interpretations are possible; perhaps more experienced developers wrote more complex code, whose bugs are harder to discover and less likely to be reported. Nonetheless, our results show that, given



(a) Linux kernel



(b) PostgreSQL

Figure 7: Percentage of buggy commits (bars) and total number of commits (circles) versus author experience

the fact that a commit is from a more experienced developer, one can be more confident about the correctness of the commit. Such a correlation could be exploited to help predict buggy code locations.

One can observe a decline in the total number of commits with experience. We believe that this is due to our sliding scale for author experience. 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 "inexperienced" developers, since all developers go through an inexperienced phase, while only a small number of developers reach the more experienced phase.

We do not understand the spike in percentage of buggy commits to the right of the PostgreSQL graph. Possible reasons include the shift to Git, which is a known historical event, or, more speculatively, perhaps PostgreSQL recently undertook major ongoing architectural revisions, carried out by the experienced developers.

3.4 Combined Time-of-day and Experience

Figure 8 combines data from Section 3.2 and 3.3.2 and correlates time-of-day with commit bugginess for inexperienced and experienced developers, plotted separately, for Linux. We used a cutoff of 2 years to separate inexperienced and experienced developers; this cutoff divides the number of commits into two approximately-equal groups. Horizontal lines in the figure represent overall bugginess.

We can see that inexperienced developers tend to do more commits between midnight and 2 AM than experienced 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 6 AM and noon) are less buggy.

Discussion.

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

3.5 Day-of-week

Our next experiment attempted to replicate the results in [19], and correlates the day of the week of a commit with its bugginess. Figure 9 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). Here, the solid horizontal line presents the overall bugginess of all commits to each project. Figure 10 presents the number of subsequent bug-fixing commits for each bug-introducing commit per day-of-week.

Our results, which use a disjoint set of repositories from those in [19], found about the same bugginess and number of introductions for each day in the Linux kernel repository, with the lowest bugginess on Sunday and highest on Monday; for the PostgreSQL repository, we observe 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. Note that, for Linux, 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). For PostgreSQL, commits fluctuate through the days of the week and decrease to about 70% of the weekday volume on the weekend.

Discussion.

We found that commits on different days of week have about the same bugginess, which does not agree with results from the prior study on two different open source projects [19]. We also found that the bugginess per day-of-week for commits varies for different software projects, implying that bugginess prediction based on day-of-week may need to be calibrated on a per-project basis.

3.6 Bug Lifetimes

Recall that the bug lifetime is the amount of time elapsed between the bug-introducing commit and its bug-fixing commit. Figure 11 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. Note that the distribution of bug lifetimes is similar for both projects; many bugs are fixed within a 120 day period and the overall lifetime appears to decrease exponentially.

Discussion.

PostgreSQL may have a longer average bug lifetime due to being a smaller, less complex project with a smaller user base. We found the sources of the long-time bugs include race conditions, incorrect

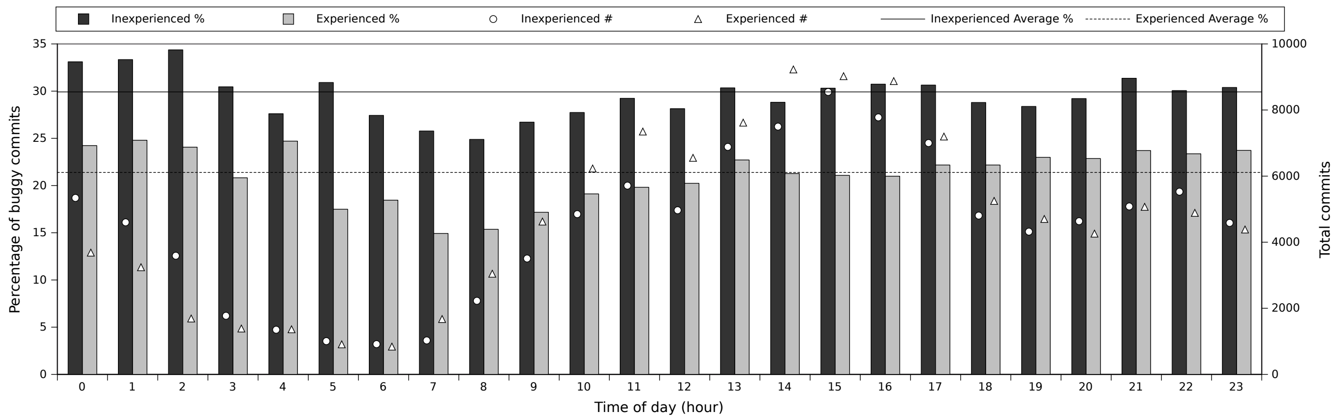
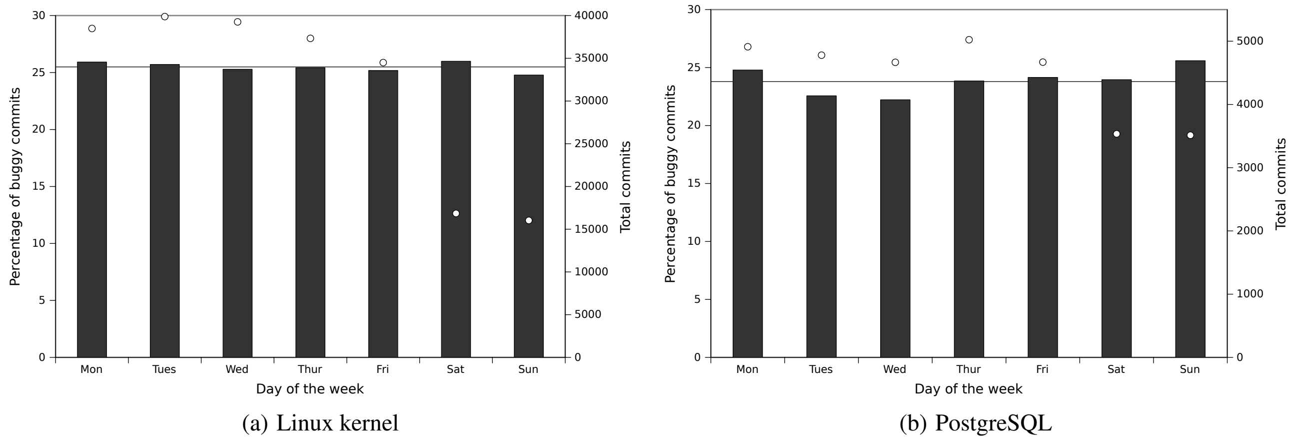


Figure 8: Percentage of buggy commits (bars) and total number of commits (circles/triangles) versus time-of-day for inexperienced and experienced Linux kernel developers



(a) Linux kernel (b) PostgreSQL

Figure 9: Percentage of buggy commits (bars) and total commits (circles) versus day-of-week

calculations and rare corner cases; such cases are intuitively more likely to be found with a larger user base.

3.7 Comment-only Commits

A surprisingly large number of bug-fixing commits reported 0 changed lines of code. We performed a random sample on 50 of these commits for each project and found that almost all of them only changed comments in the source code, which we did not count as changed lines of code. For the Linux kernel 2.15 $\pm$ 0.10%[3] of the bug-fixing commits (1,226 commits[4]) were on comments only. We followed the same procedure for PostgreSQL and found 2.97 $\pm$ 0.45% (about 131 commits) of bug-fixing commits were on comments only. These hundreds and thousands of comment-only commits show that developers spend a nontrivial amount of time purely maintaining the correctness of comments; these numbers do not even consider the amount of time that developers incidentally update comments along with the code.

3.8 Validation

To validate our results, we estimated the precision and recall of our technique for identifying bug-fixing commits on both projects.

[3]We report the margin of error with 95% confidence level.

[4]Estimated based on the percentage of comment-only commits and the total number of bug-fixing commits.

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 3(a) and 3(b) summarize our findings.

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

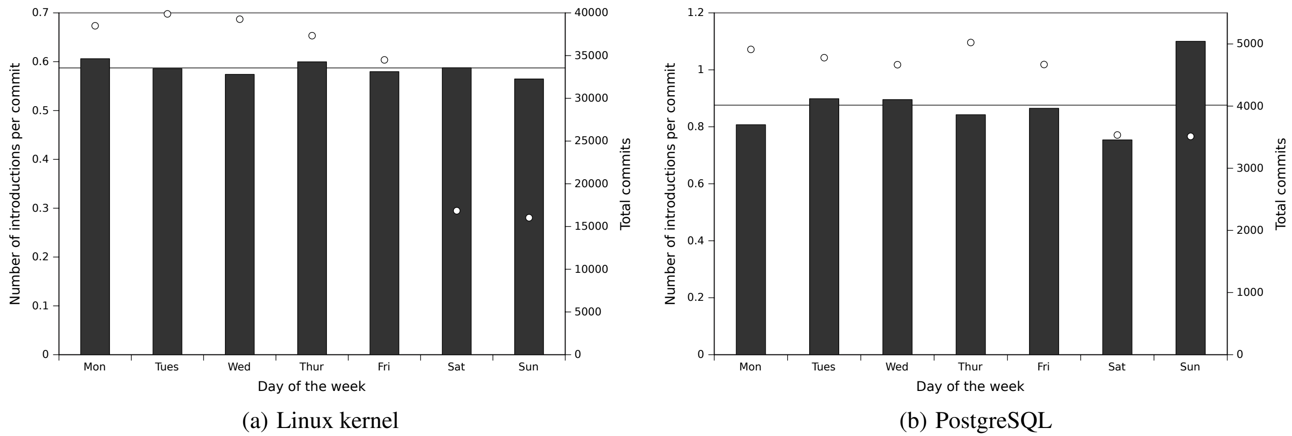
(a) Linux kernel

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

(b) PostgreSQL

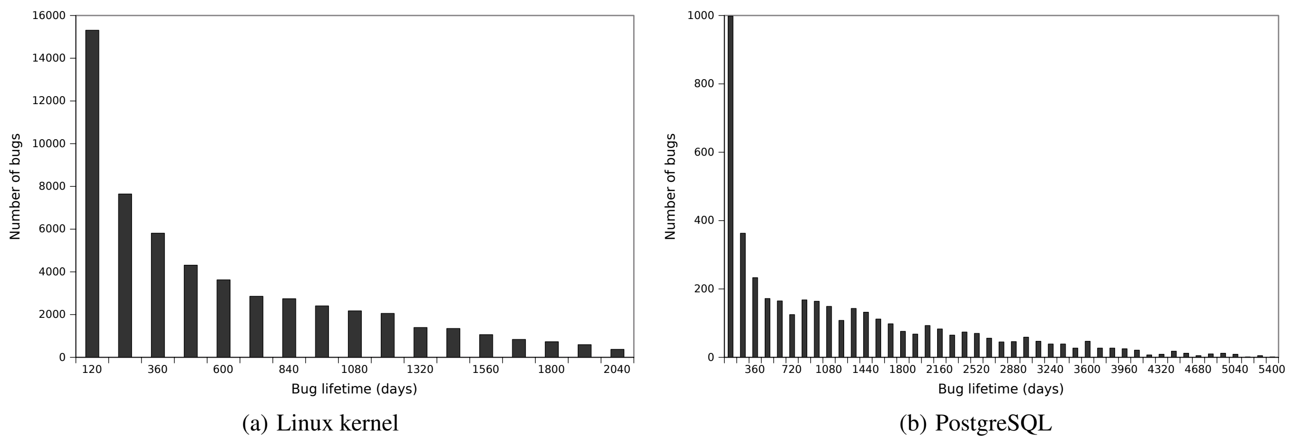
Table 3: Confusion matrices

We evaluated the precision—that is, the proportion of identified bug-fixing commits which do indeed fix bugs—and found that, for the Linux kernel, 48 of the 55 bugs that we automatically identified as bug-fixing commits did indeed fix bugs, while 7 did not; for PostgreSQL, 30 of 35 identified fixes were indeed fixes. Some 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-



(a) Linux kernel (b) PostgreSQL

Figure 10: Subsequent bug fixes per commit (bars) and total commits (circles) versus day-of-week



(a) Linux kernel (b) PostgreSQL

Figure 11: Histogram of bug lifetime counts

added in a later version; 4) poor uses of version control systems which included many different changes 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 Linux and 71% for PostgreSQL.

4. RELATED WORK

We survey work related to our study of factors affecting bugginess (namely, day-of-week); prior work on empirical studies of commits; studies of bugginess in distributed software development; and empirical studies on bug lifetime.

Day of Week of Commits.

The most closely related work to ours, Śliwerski et al [19], 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 and 86% for PostgreSQL.

Empirical Studies on Commits.

Many empirical studies have been conducted to understand and leverage different aspects of commits [7, 9, 14, 18, 21], which studied the distributions of commit sizes and how commit sizes correlate with other metrics such as different development activities, commit classifications, etc.

Hindle et al [9] classified commits into different categories, one of which is non-functional commits (e.g., modification of comments, documentation, etc.). Our study differs from [9] in that we specifically investigated comment-fixing commits. For example,

a refactoring commit would be considered to be a non-functional commit by the previous study, but not a comment-fixing commit in this paper. iComment [22] only showed that FreeBSD contains many comment-fixing commits; it is not a comprehensive study on comment-fixing commits. We estimate that about 1,200 and 130 commits (Section 3.7) in the Linux kernel and PostgreSQL, respectively, only fix comments, and do not modify the code, showing that developers spent time maintaining the correctness of comments.

Distributed Software Development and Code Quality.

Several previous studies sought to understand how distributed software development affects code quality [1, 16, 20] in open source and commercial software. While the two 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 social 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 [2] 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 [10] examined the bug lifetimes in PostgreSQL. Neither of the two previous studies investigated social characteristics such as commit time and author experience.

5. CONCLUSIONS AND FUTURE WORK

This paper analyzed 57,028 and 4,399 bug-fixing commits in two large and widely-used open-source software projects, the Linux kernel and PostgreSQL, to study the correlation between commit correctness with several commit social characteristics, such as the time-of-day of commits, the day-of-week of commits, developer experience, and developers' commit frequency. We presented several interesting findings, including: (1) late-night commits (between midnight and 4:00 AM) are buggier than average, while morning commits (7:00 AM–noon) are 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 for different software projects, implying that the bugginess prediction based on the day-of-week of commit metric may need to vary on a project-by-project basis; and (3) developers who commit to a project on a daily basis write fewer buggy commits for that project, while day-job developers are more likely to produce bugs, indicating that we may want to promote the practice of daily committing developers code-reviewing other developers' commits. 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. To extend our developer experience study, we can add developers' contributions to other open-source projects to better understand a developer's overall programming experience. 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.

6. REFERENCES

[1] C. Bird, N. Nagappan, P. T. Devanbu, H. Gall, and B. Murphy. Does distributed development affect software quality? In *ICSE*, pages 518–528, 2009.

[2] 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.

[3] T. Graves, A. Karr, J. Marron, and H. Siy. Predicting fault incidence using software change history. *IEEE TSE*, 2000.

[4] L. Guo, Y. Ma, B. Cukic, and H. Singh. Robust prediction of fault-proneness by random forests. In *ISSRE*, 2004.

[5] R. Haas. So, why isn't PostgreSQL using Git yet? http://rhaas.blogspot.com/2010_09_01_archive.html.

[6] A. E. Hassan. Predicting faults using the complexity of code changes. In *ICSE*, pages 78–88, 2009.

[7] L. Hattori and M. Lanza. On the nature of commits. In *ASE*, pages 63–71, 2008.

[8] 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.

[9] A. Hindle, D. M. German, and R. Holt. What do large commits tell us?: A taxonomical study of large commits. In *MSR*, pages 99–108, 2008.

[10] S. Kim and E. Whitehead Jr. How long did it take to fix bugs? In *MSR*, pages 173–174, 2006.

[11] S. Kim, T. Zimmermann, K. Pan, and E. Whitehead. Automatic identification of bug-introducing changes. In *ASE*, pages 81–90, 2006.

[12] 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.

[13] 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.

[14] A. Mockus and L. G. Votta. Identifying reasons for software changes using historic databases. In *ICSM*, 2000.

[15] A. Mockus, D. M. Weiss, and P. Zhang. Understanding and predicting effort in software projects. In *ICSE*, 2003.

[16] 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.

[17] T. Ostrand, E. Weyuker, and R. Bell. Predicting the location and number of faults in large software systems. *IEEE TSE*, 31(4), 2005.

[18] R. Purushothaman and D. E. Perry. Toward understanding the rhetoric of small source code changes. *IEEE TSE*, 2005.

[19] J. Śliwerski, T. Zimmermann, and A. Zeller. When do changes induce fixes? In *MSR*, pages 24–28, 2005.

[20] D. Spinellis. Global software development in the FreeBSD project. In *GSD*, pages 73–79, 2006.

[21] E. B. Swanson. The dimensions of maintenance. In *ICSE*, pages 492–497, 1976.

[22] L. Tan, D. Yuan, G. Krishna, and Y. Zhou. /* iComment: Bugs or bad comments? */. In *SOSP*, 2007.

[23] T. Zimmermann and N. Nagappan. Predicting defects using network analysis on dependency graphs. In *ICSE*, 2008.

[24] T. Zimmermann, R. Premraj, and A. Zeller. Predicting defects for eclipse. In *PROMISE*, 2007.

[25] T. Zimmermann and P. Weißgerber. Preprocessing CVS data for fine-grained analysis. In *MSR*, pages 2–6, 2004.