View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by Crossref

Souza *et al. Journal of Software Engineering Research and Development* (2015) 3:9
DOI 10.1186/s40411-015-0024-z

Journal of Software Engineering Research and Development
a SpringerOpen Journal

RESEARCH Open Access

Patch rejection in Firefox: negative reviews, backouts, and issue reopening

Rodrigo RG Souza[1*], Christina FG Chavez[1] and Roberto A Bittencourt[2]

*Correspondence:
rodrigo@dcc.ufba.br
[1]Department of Computer Science, Federal University of Bahia, Salvador, Brazil
Full list of author information is available at the end of the article

Abstract

Background: Writing patches to fix bugs or implement new features is an important software development task, as it contributes to raise the quality of a software system. Not all patches are accepted in the first attempt, though. Patches can be rejected because of problems found during code review, automated testing, or manual testing. A high rejection rate, specially later in the lifecycle, may indicate problems with the software development process.
Our objective is to better understand the relationship among different forms of patch rejection and to characterize their frequency within a project. This paper describes one step towards this objective, by presenting an analysis of a large open source project, Firefox.

Method: In order to characterize patch rejection, we relied on issues and source code commits from over four years of the project's history. We computed monthly metrics on the occurrence of three indicators of patch rejection—negative code reviews, commit backouts, and bug reopening—and measured the time it takes both to submit a patch and to reject inappropriate patches.

Results: In Firefox, 20 % of the issues contain rejected patches. Negative reviews, backouts, and issue reopening are relatively independent events; in particular, about 70 % of issue reopenings are premature; 75 % of all inappropriate changes are rejected within four days.

Conclusions: Patch rejection is a frequent event, occurring multiple times a day. Given the relative independence of rejection types, existing studies that focus on one single rejection type fail to detect many rejections. Although inappropriate changes cause rework, they have little effect on the quality of released versions of Firefox.

Keywords: Release engineering; Mining software repositories; Empirical software engineering; Patch rejection

Introduction

According to Lehman et al. (1997), many software systems need to be constantly changed to remain useful, and the quality of such systems will be perceived as declining unless they are rigorously maintained. Therefore, for a high quality product, that satisfies users' needs, it is important to keep track of issues with the product, the patches that resolve those issues, and all verification steps during the lifecycle of a software release.

A patch goes through multiple stages before it is integrated into a release, depending on the specific development process employed in a team. In Mozilla Firefox, for example,

Springer

© 2015 Souza et al. This is an Open Access article distributed under the terms of the Creative Commons Attribution License (http://creativecommons.org/licenses/by/4.0), which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is properly credited.

after a patch is written, it goes through code review, then it is committed to a source code repository and compiled to multiple platforms, undergoes comprehensive automated testing, and finally manual testing.

At any of these verification steps, a patch that is deemed incomplete or inappropriate is rejected. Rejecting a patch for an issue raises the confidence that, when a new version of Firefox is released, relevant issues found in the last version are indeed resolved. On the other hand, a high rejection rate raises the concern that significant time is spent with rework, i.e., writing, reviewing, and testing new patches for a previously handled issue.

In this paper, we use the following definitions: *issues* are bugs, feature requests, or other types of maintenance requests (e.g., refactoring, creation of test cases) that are reported in an issue tracking system for a software system; issue reports are closed when the corresponding issues are resolved. *Patches* are source code changes that resolve issues; patches are usually first submitted for code review and then committed into a source code repository. *Verification steps* are tasks that evaluate whether a patch is appropriate, and include code review, automated testing, and manual testing. A *patch rejection* is the finding, during a verification step, that a patch is inappropriate; therefore, a patch may be rejected during code review, automated testing, manual testing, or even after a release.

In this paper, we aim to characterize the problem of patch rejection in terms of its occurrence during code review, automated and manual testing, and by measuring its impacts on the time needed to definitely resolve issues. We compare those three forms os patch rejection, measuring how often they occur in the same issue report, and also compare inappropriate and appropriate patches. Finally, we expand on previous studies that compared patch rejection in Firefox under two distinct periods. To this end, we analyze a four-year period of data from Firefox, a large open source project.

In order to allow other researchers to replicate our findings and to perform derived research, we have made all the source code of our analysis scripts available on https://github.com/rodrigorgs/withdrawal-firefox.

The remainder of this paper is organized as follows. Section 'Literature review' presents related work on patch rejection. Section 'Background' presents the process and tools adopted by Firefox. Section 'Methods' presents the data used in this study, together with data transformation procedures and statistical methods employed. Section 'Results and discussion' presents the quantitative results of the analysis, as well as some explanations on the results and a discussion of threats to validity. Section 'Conclusions' summarizes the conclusions of this study and presents perspectives on future work.

Literature review

This section presents previous work about patch rejection and related concepts. It includes previous studies about patch rejection in Firefox, from which this work partially derives.

Patch rejection can happen in different verification steps and be tracked in multiple ways. It can be the rejection of a patch during code review by a peer, which may be tracked in an issue tracking system, in a mailing list, or in a specialized review tool (Rigby and Storey 2011). It can also happen after a patch is committed to a central code repository, in which case a supplementary patch should be committed (Park et al. 2012). Many projects close issues that are thought to be resolved, and then reopen the issue report to signal that the patch was deemed inappropriate (Shihab et al. 2010).

Although issues are, by definition, a superset of bugs, many software projects use the term *bug* to refer to non-corrective maintenance (Chapin et al. 2001). A study on five open source projects showed that about 33 % of all bug reports refer, instead, to feature requests, requests for performance improvements, and other maintenance tasks (Herzig et al. 2013). In the next subsections, we keep the terms *bug* and *bug fix*, used by the original papers, although, more often than not, they can interchangeably be used with *issue* and *patch*, respectively.

Bug reopening

A bug report may be closed even if it is not fixed when, for instance, it is considered a duplicate or an invalid bug report. Therefore, bug reopening is not always related to patch rejection; sometimes, it is the result of new information that helps triage bugs.

Shihab et al. (2010; 2012) developed a decision tree model, based on features about the bug report, the bug fix, and human factors, to predict which bugs would be reopened. They found that among the top predictors of bug reopening are the component in which the bug was found and the time needed to submit the first fix to the bug.

In a partial replication of Shihab's work, Zimmermann et al. (2012) found that bugs reported by users are more likely to be reopened than those discovered through code review or static analysis, supposedly because those reported by users are harder to reproduce and tend to be more complex. Other factors that favor reopening, according to the authors, include bug severity and geographical distribution of developers participating in the bug.

The authors also asked Microsoft engineers about the common causes for bug reopening. Responses included the difficulty to reproduce a bug, the misunderstanding of root causes, the lack of information in the initial report, the increase of the bug priority, incomplete fixes, and code integration problems.

Focusing on the reopening of bugs fixed by a source code patch, Almossawi (2012) analyzed 32 open source systems in the GNOME project and, using a logistic regression model, concluded that bugs located in code with high cyclomatic complexity are more likely to be reopened. Jongyindee et al. (2011) found that bugs fixed by more experienced developers are less likely to be reopened.

Regarding the impact of reopened bugs, the literature reports reopening rates as low as 1.4 % (Almossawi 2012) and as high as 11.7 % (Jongyindee et al. 2011), obtained from open source projects. Also, reopened bugs are said to have a life cycle from 2 to 5 times higher (Jongyindee et al. 2011; Shihab et al. 2010) than regular bugs, and to involve the participation of 40 % more developers (Jongyindee et al. 2011).

Supplementary bug fixes

Park et al. (2012) studied bug fixes with programming errors that led to the creation of supplementary bug fixes, i.e., new, improved fixes for the same bug. By analyzing projects from both Eclipse Foundation and Mozilla Foundation, they found that 22 % to 33 % of resolved bugs involved more than one source code commit. The numbers overestimate the proportion of rejected bug fixes because, while multiple commits often represent multiple attempts to fix a bug (i.e., a commit fixes problems with a previous commit), sometimes they are the result of a developer splitting a bug fix into multiple small commits to facilitate peer review.

An et al. (2014) analyzed the relationship between supplementary bug fixes and bug reopening on WebKit and projects from Eclipse Foundation and Mozilla Foundation, partially replicating the studies by Park et al. (2012) and by Shihab et al. (2010). They found that between 21 % and 34 % of all bugs with supplementary fixes were eventually reopened, and that only a little more than 50 % of all reopened bugs are associated with supplementary bug fixes. They conclude that bug reopening is not always related to incorrect bug fixes.

Code review

Bird et al. (2007) developed an algorithm to detect patches in email messages and determine when such patches were applied to a code base. In their analysis of three open source projects—Apache, Python, and PostgreSQL—they found that between 25 % and 49 % of all submitted patches were applied without modifications to the code base. The remaining patches were either rejected or applied with modifications—their algorithm cannot tell the two cases apart.

Rigby et al. (2011; 2014) studied the peer review process of several open source projects. They identified two styles of review processes: review-then-commit, and commit-then-review. The first style is the most common, and is adopted by Mozilla. They also characterized reviews according to their frequency, the participation of reviewers, among other dimensions.

Jeong et al. (2009) studied the acceptance of bug fixes submitted to peer review in Firefox and Mozilla Core. They found that about 7 % of the bug fixes are rejected, and that an average review takes about 1.5 days. About 50 % of review requests are in an "open" state (i.e., neither accepted nor rejected), which may indicate a "gentle rejection", a lack of interest from reviewers, or that the initial bug fix was superseded by an improved version even before the reviewer had the opportunity to look at the first version.

Nurolahzade et al. (2009) inspected a random sample of 112 bug reports from Firefox. They identified recurrent patterns in the behavior of contributors and reviewers. For example, some contributors submit work-in-progress patches, and some reviewers avoid explictly rejecting patches (what Jeong et al. called "gentle rejection"). They also make a distinction between reviews conducted by module owners (i.e., developers responsible for specific modules in the software), who are concerned with long-term maintainability, and those conducted by other peers, who are usually more interested in functionality and usability.

Patch rejection and rapid releases

In 2011, Firefox changed its release process from a traditional release model, delivering major versions after more than one year of development, to a rapid release model, in which versions containing new features are released every 6 weeks. The change gave rise to numerous studies analyzing its impact on Firefox's quality (Khomh et al. 2012), testing efforts (Mantyla et al. 2013), and reputation among users (Plewnia et al. 2014).

In previous studies, we evaluated how rapid releases impacted the issue reopening rate (Souza et al. 2014) and the backout rate (Souza et al. 2015), two forms of patch rejection. In this study, we add another form of patch rejection—negative code reviews—and focus on characterizing and comparing, using data from Firefox, the three forms of patch rejection.

Summary on patch rejection

The papers in the scientific literature approach phenomena related to patch rejection under a multitude of perspectives. Some papers analyze status changes in issue reports to detect reopening events; such events may be related to patch rejection, but can also be caused by insufficient information and communication problems in a team. Other papers look at supplementary fixes, i.e., sets of commits aimed at the same issue, as an indicator of inappropriate patches. To our knowledge, only one study (An et al. 2014) compares bug reopening and supplementary fixes, and it concludes that, although both are used as indicators of patch rejection, they agree at only 50 % of the issues. Finally, there are papers that study strategies for code review, rejection rates and review time.

Background

The objective of this section is to explain the development process adopted by Mozilla, as well as to describe how specific parts of the process are tracked using software development tools. The description presented in this section is based on semi-structured interviews with a former Mozilla engineer, and on information found in Mozilla's wiki (Mozilla 2014) and in Mozilla's draft on process documentation for software release mechanics (Mozilla 2011).

Tools

At Mozilla, two tools are central to coordinate the development of features and bug fixes: Bugzilla[1], an issue tracking system developed by Mozilla, and Mercurial[2], a distributed version control system.

Bugzilla

In Bugzilla, people can report issues and then update issue reports with information regarding the issue and the process of resolving the issue, either by uploading files (screenshots, trace logs, patches...) or by commenting and updating issue report fields. Each issue report has a *status* field, which can take values such as NEW, RESOLVED, VERIFIED, and REOPENED.

For RESOLVED issues, a resolution must be chosen among FIXED, INVALID, WONTFIX, DUPLICATE, WORKFORME, or INCOMPLETE. Although the actual interpretation of each resolution is project-dependent, Bugzilla's documentation (Bugzilla 2015) proposes the following interpretations, which are followed by Mozilla's projects:

- FIXED: a fix for the issue was commited to a source code repository and tested;
- INVALID: the report is not an issue;
- WONTFIX: the report is an issue, but developers have no intention of resolving it;
- DUPLICATE: the report is a duplicate of another report;
- WORKSFORME: developers were unable to reproduce the bug;
- INCOMPLETE: the report provides insufficient information.

Figure 1 shows typical issue status and transitions between them. An issue starts with status UNCONFIRMED, if reported by a regular user, or NEW, if reported by a trusted user. After that, the issue may be ASSIGNED to a developer, and then RESOLVED (with resolution FIXED, if a patch was committed to the source code repository). After successful manual testing, the status is changed to VERIFIED. Any issue marked RESOLVED or

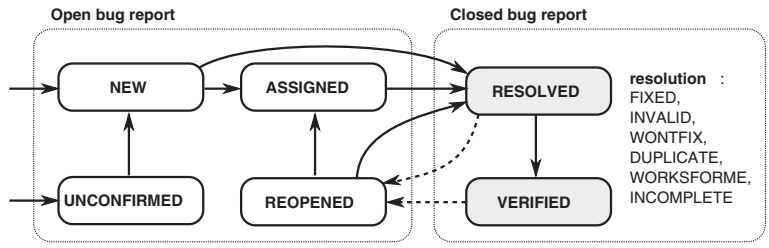

Fig. 1 Issue status and their transitions on Bugzilla

VERIFIED is considered closed and may be REOPENED if the initial solution was deemed inappropriate, so it can be RESOLVED correctly.

The code review process for Firefox also happens on Bugzilla, with the help of issue report flags. First of all, a developer attaches a patch to an issue report and adds the flag "review?" to the attachment, meaning that he wants someone to review the patch. The reviewer then responds by replacing the flag "review?" with either "review+" or "review-", representing patch acceptance or rejection, respectively. When a patch is rejected, the developer should submit a revised patch and flag the old one as "obsolete".

Mercurial

Source code is kept in Mercurial repositories, the most important being called Mozilla-central. Patches are represented by *commits*, identified by a hash. Besides the hash, commits also contain a diff of the changes, a message that describes the change, and metadata such as author and date. By convention, the commit message often refers to the number of the issue resolved by the commit.

Whenever a commit is deemed inappropriate, it is *backed out*, a process that creates a new commit reverting the inappropriate commit. By convention, backout commit messages contain the expression "backout"—or one the variations "back out", "backing out", and "backed out"—, followed by the hash of the commit being backed out.

Since 2011, the conventions used in commit messages are enforced by a Mercurial hook that prevents pushing to Mozilla repositories any commits whose message does not conform to conventions[3]. In addition to patches to issues and backout commits, there are conventions for merge commits and a few other special cases.

Firefox's change process

The process of resolving an issue starts with an issue being chosen by a developer. The developer checks out a copy of Mozilla-central or, more likely, updates his local copy, and then modifies it in order to resolve an issue. The changes are packed in a patch, which the developer attaches to the issue report and asks a specific colleague to review, usually the owner of the module being modified. If the reviewer approves the patch, the developer commits it to the Mozilla-central Mercurial repository. Otherwise, if the patch is *rejected* during peer review, it should be marked as obsolete and the developer should improve his patch to start the process again.

After the patch is committed, the Mozilla integration server checks out the code from Mozilla-central, builds it for multiple platforms and runs automated tests. The whole process takes hours to complete, and developers who commit code to Mozilla-central are

expected to wait and see if the build was successful and all tests pass. If this is the case, the developer should change the issue resolution to FIXED. Otherwise, the developer is expected to backout his commit. A backout that occurs before a successful build is called an *early backout*.

Backing out a problematic commit as soon as possible is important because other developers rely on code that is in Mozilla-central to write their patches. If that code cannot be built or fail tests, a developer cannot test his own patches on that code.

For that reason, developers can try their patches on the so-called Try server before committing them to Mozilla-central. A developer submits a patch to the server and choose to which platforms the code should be built and which test suites should be run. A successful Try build, even on a limited set of plataforms and test suites, raises the confidence that the patch will not break the build on Mozilla-central.

After a successful build, a tester downloads a nightly build and manually tests it to check if all issues marked as RESOLVED were actually resolved. If this is the case, the issue report is marked as VERIFIED; otherwise, it is marked as REOPENED, and the corresponding commit is backed out. A backout that occurs after a successful build is called a *late backout*.

Wrapping up, there are three types of patch rejection in the process: the *negative review* of a patch submitted to code review, the *backout* of a commit, and the *reopening* of an issue report. Backouts can be further split into *early backouts*, for commits that were not integrated into a successful build, and *late backouts*, for commits that passed compilation and automated tests but were later discovered to be inappropriate.

Integration repositories

In June 8, 2011, integration repositories, such as Mozilla-inbound, were introduced in the process. With this change, developers stopped committing directly to Mozilla-central, and started to commit to integration repositories instead. The code on integration repositories is built and run against automated tests before being merged into central.

The merging to Mozilla-central is not performed by the developers who commit the patches; instead, it is performed by designated developers called *build sheriffs*. Sheriffs watch the build and are responsible for backing out commits that break the build or cause test failures before merging commits into central.

With integration repositories, only code that passes build and automated tests are merged into Mozilla-central, keeping it more stable. Furthermore, because of sheriffs, developers do not need to watch builds and backout commits anymore.

Types of rejections and their interpretations

There are three types of events that are indicators of a patch rejection:

- *negative review* of patch under code review;
- *backout* of a commit (further classified in *early* and *late*);
- *reopening* of an issue report;

The three types of events have different meanings, and sometimes the difference is subtle. *Negative review* is the very first form of rejection that can occur in the lifecycle of patch. A developer can review negatively a patch for varied reasons (Tao et al. 2014), e.g., because it could induce failures or poor performance in particular cases, because it

does not conform to coding conventions, because it introduces unintuitive user interface elements, and so on. Therefore, a rejection in peer review does not necessarily mean that the patch is functionally wrong, it only means that the change is somehow inappropriate or less than ideal.

A *backout* indicates that a patch that was eventually approved by a reviewer and committed to a code repository was then deemed inappropriate. Usually, in this case, it means that there is a problem with the solution: a performance issue, incorrect outputs, uncovered corner cases, and so on. A backout is classified as early when it occurs before the corresponding issue report is closed (i.e., its status is changed to RESOLVED with resolution FIXED), which happens when the patch has been integrated into a successful build. Therefore, early backouts are likely an indicator that the corresponding patch failed compilation or automated testing. A backout is classified as late when it occurs after the issue report is closed, meaning that the problem was likely discovered during manual testing.

A *reopening* occurs when someone explictly changes the status of an issue report to REOPENED, which can only happen when the issue report is closed. It may appear that reopenings and late backouts should always occur together, but this is not the case. Sometimes a closed issue report is *prematurely reopened* because of misunderstandings. For instance, after a developer reopens an issue report, a second developer may argue that the problem found by the first is in fact a different issue, and then change the status back to RESOLVED. Therefore, reopening may imply a valid rejection, when it is followed by a late backout, or may just signal communication problems.

Methods

In order to characterize patch rejection, we analyzed all issue reports and commit messages created in a period of more than four years, as shown in Fig. 2. This sample is not intended to support conclusions about the earlier years of Firefox, since the development process may have changed significantly along the years. The period includes about two years of traditional releases, about two years of rapid releases, and also a five-month transitional period, during which two changes took place: the rapid release model was partially implemented and integration repositories were created.

Issue reports were provided as a SQL database dump by a Mozilla engineer. The dump contains everything that is stored by Bugzilla, except for some user data, for security reasons, and attachment contents, which would increase significantly the data set size.

The database dump used to be publicly available, but was withdrawn in August, 2014. As of November, 2014, Mozilla engineers are working on improving the data sanitization process in order to release database dumps again. For more information see issue reports 1013953 and 1054795 at https://bugzilla.mozilla.org/.

Commit messages were extracted using Mercurial and further analyzed using regular expressions. The Mercurial repository is publicly available at https://hg.mozilla.org/mozilla-central/.

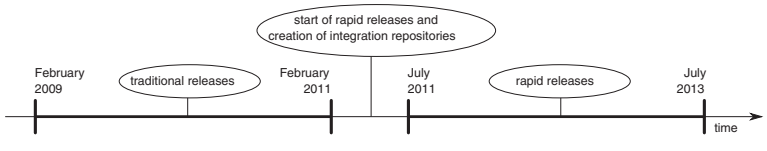

Fig. 2 Period analyzed in this study

Research design

To enable the characterization of patch rejection, we detect, for each issue in our data set—including those in traditional releases, in rapid releases, and in the transitional period,—three types of rejection events:

- the *negative review* of a patch that was submitted to peer review;
- the *backout* of a patch that was committed (further split into *early* and *late backout*);
- the *reopening* of an issue report that was closed.

It should be noted that an issue report may, over its history, receive multiple patches, which may be rejected multiple times. For the sake of simplicity, we study only the first rejection, the first early backout, the first late backout, and the first reopening of an issue report. Therefore, we do not measure directly the number of patches that were rejected; instead, we measure the number of issue reports that had at least one patch rejected.

For each type of rejection event, we compute metrics related to three perspectives: patch effectiveness, patch efficiency, and issue latency.

The *patch effectiveness* perspective is concerned with the proportion of issues that contain only effective patches, i.e., patches that are not eventually rejected. For instance, the metric *backout rate* computes the proportion of issues with at least one commit that was considered inappropriate and was thus backed out. Other metrics include *negative review rate* and *issue reopening rate*. A high proportion of issues with inappropriate patches suggests that the source code is hard to understand or that it is hard to change without breaking it.

The *patch efficiency* perspective is concerned with how long it takes to propose a patch. For instance, the metric *time to commit* measures the time between the creation of an issue report and the first commit of a patch to that report. Other metrics include *time to submission* (time to submit to code review) and *time to closing* (the issue). We separately compute the time it takes to propose an *inappropriate patch* (i.e., a patch that will be eventually rejected), and the time it takes to propose a *supplementary patch* (i.e., a patch that improves a previously rejected patch). This perspective helps weigh the problem of rejections in terms of the time they take during development.

The *issue latency* perspective is concerned with how long it takes to reject an inappropriate patch. For instance, the metric *time to backout* measures that time it takes to backout for the first time an inappropriate commit. Other metrics include *time to negative review* and *time to reopening*. If it takes too long to reject inappropriate patches, it is more likely that developers write changes that rely on source code with problems.

Figure 3 shows important events in an issue lifecycle, and how they are used as reference to compute metrics related to patch efficiency and issue latency. For the peer review process, for instance, we measure the time from issue report creation to the submission of the first patch to Bugzilla (*time to submission*), the time between the first submission and the first negative review (*time to negative review*), and the time from the first rejection to the next patch submission (*time to supplementary submission*). The same logic is applied to backouts and reopenings.

We also partially replicate our previous studies about the impact of rapid releases on patch rejection (Souza et al. 2014, 2015), however adding negative reviews, which were not previously considered. In order to study how those metrics changed after Firefox's adoption of rapid releases, we compute them for two periods, one during which Firefox

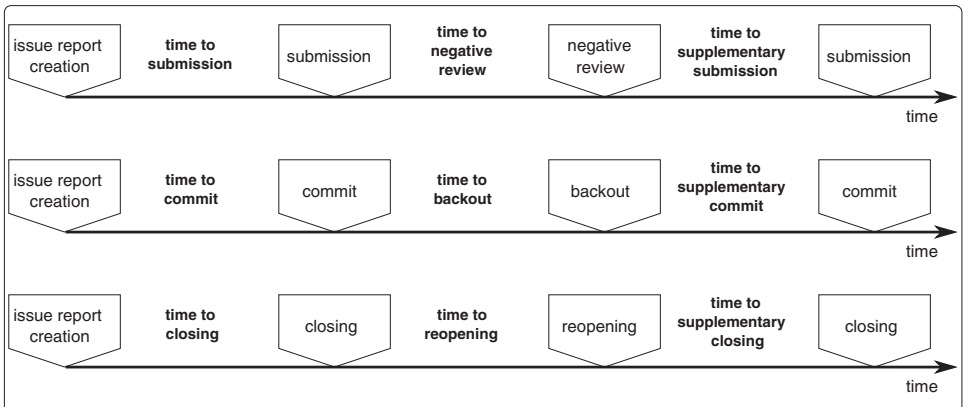

Fig. 3 Events and time-related metrics (efficiency and latency)

was developed with traditional releases, and another one during in which it was developed with rapid releases and integration repositories. In this analysis, the transitional period is ignored, since it is not representative of either traditional or rapid releases.

Data analysis

Figure 4 outlines the data analysis process. Boxes represent processing steps, and ovals represent data, with arrows connecting a processing step to input and output data. Numbers near ovals represent number of data points (e.g., number of issues, number of commits).

First, all issue reports for Firefox that ended up with resolution FIXED are loaded, together with the history of their changes (step 1 of Fig. 4), resulting in 82920 issues created between 1997 and 2013. Relevant attributes are extracted, such as issue number, creation date, changes to issue status, resolution and flags, together with dates for all changes.

In parallel, all commits from 2007 to 2014 in Mozilla-central are loaded, resulting in 215296 commits (step 2). Note that not all those commits are for Firefox, since other Mozilla products use this repository. For each commit, we extract the identifier, or commit hash, the date, and the first line of the message.

Detection of events from issue reports

After that, all relevant events are detected from issue reports (steps 3 and 4). Events from issue reports are stored as structured data.

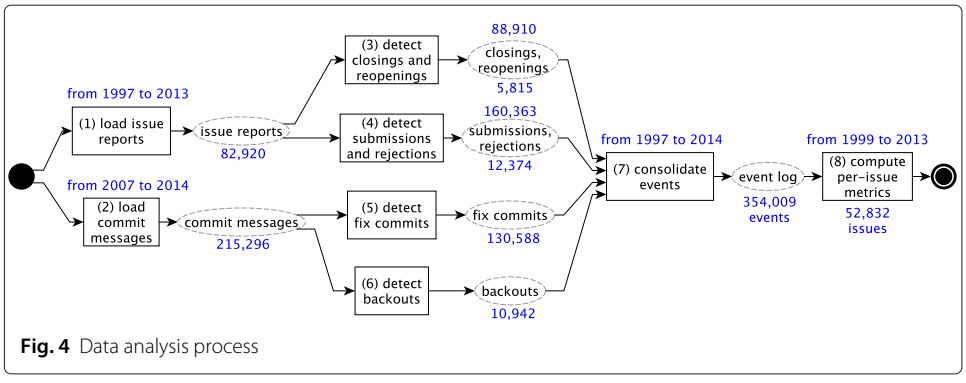

Fig. 4 Data analysis process

In step 3, we detect closings and reopenings:

- the event *issue report creation* is extracted from the "reported date" field;
- the event *closing* is detected whenever the issue resolution is changed to FIXED (we ignore events that close an issue as INVALID, WONTFIX, DUPLICATE, WORKFORME, or INCOMPLETE), resulting in 88910 closing events;
- the event *reopening* is detected whenever issue status is changed to REOPENED when the issue resolution is set to FIXED (we ignore reopening of an issue when it is tagged as duplicate, invalid, etc.), resulting in 5815 reopening events;

In step 4, we detect submissions and negative reviews:

- the event *submission* corresponds to adding the "review?" flag to an attachment (the question mark stands for a developer "asking for a review"); we detected 160363 submissions;
- the event *negative review* corresponds to adding the "review-" flag to an attachment; we detected 12374 such events.

Detection of events from commits

We detect two events from commits: "commit" (step 5), i.e., committing an issue-resolving patch to a central repository, and "backout" (step 6), using the same heuristics we used in previous work (Souza et al. 2014, 2015). Detecting issue-resolving and backout commits is more difficult, since involves extracting information from messages written in natural language. Also, besides detecting the type of commit, it is also necessary to map commits to issues. Threats to validity are discussed later in this paper.

A commit is detected as issue-resolving if its message starts with the word "bug", followed by a 5- or 6-digit number (step 5). Firefox uses the term "bug" to refer to all issues, even feature requests. We found 130588 such commits. The number is the issue identifier, which allows mapping the commit to the issue report it resolves. The following is a typical message for an issue-resolving patch (emphasis added):

Bug 939080 - Allow support-files in manifests to exist in parent paths.

For backout commits, it is necessary to determine which patches they backout (step 6). A typical backout commit references either the commits it reverts, the issues whose patches are being reverted, or both, e.g.:

Back out 7273dbeaeb88 (bug 157846) for mochitest and reftest bustage.

All commits whose message matched the regular expression "back.{0,5}out", i.e., the words "back" and "out"separated by up to five characters, were interpreted as backout commits. The purpose of the five characters separation is to match common variations, such as "backed out" and "backing out". In a backout commit, all decimal numbers with 5 to 6 digits were interpreted as issue report identifiers, and all hexadecimal numbers with 7 to 12 digits were interpreted as commit hashes.

Not all issue reports referenced in backout commits are issue whose patch is being reverted, though. For instance, consider the following commit message:

Back out parts of bug 698986 to *resolve* bug 716945

The commit message references two issue reports. The first one is the issue being reopened; the second one, though, is the the issue being resolved by the commit.

Therefore, in the example, issue 716945 was not backed out (at least not in this particular commit).

After reading a sample of backout commits, we decided that issue identifiers after the following expressions should not be interpreted as issues being backed out:

- resolve, fix – e.g., "Backout bug 555133 to fix bug 555950." (i.e., bug 555133 was backed out, while bug 555950 was not);
- causing, cause, because – e.g., "Backed out changeset 705ef05105e8 for causing bug 503718 on OS X";
- due to – e.g., "Backed out changeset 58fd8a926bf5 (bug 366203) due to it causing bug 524293.";
- suspicion – e.g., "Backout revisions (...) on suspicion of causing (...) bug 536382."

Finally, whenever a back commit references other commits, we identify the issue they tried to resolve. For example:

commit b6d4...: Bug 475968. Pad out the glyph extents of Windows text (...)
commit 980e...: Backed out changeset b6d4... for causing (...)

In this example, one can deduce that issue 475968 was reopened, because commit `b6d4...`, that resolved it, was later backed out by commit `980e...`.

Consolidation of events and analysis

After all relevant events were detected, they were consolidated in a single data set with records in the form ⟨issue number, event, time when event occurred⟩ (step 7 of Fig. 4). In this step we only considered issues with a complete lifecycle, i.e., closed issues associated with at least one issue-resolving commit, resulting in 354009 events.

After that, events were aggregated by issue, so all previously described per-issue metrics could be computed, such as *reopening rate* and *time to backout* (step 8). This resulted in 52832 issues created between 1999 and 2013, which received commits between 2007 and 2014 (it may seem counterintuitive that an issue reported created in 1999 would receive a commit in 2007, but that actually happens in our data set).

For the characterization of patch rejection, metrics were computed for all issues that were created in the period under analysis (Fig. 2). Quartiles were computed to understand the variation of time-related metrics across issues. Venn diagrams were created to help understand how often different types of rejection occur together.

For the assessment of the impacts of Firefox's adoption of rapid releases, issues were split into two groups, those in traditional releases and those in rapid releases, based on their timestamps, and metrics were computed for each group. Issues in the transitional period were not considered in this analysis.

The specific timestamp used to determine the group of an issue depends on the metric. For instance, when computing *time to commit*, we are interested in the time it takes to commit a patch for an issue that was created in a certain period; therefore, for this specific metric, issues are classified according to their creation time. When computing *time to backout*, on the other hand, the question is "how long it takes to backout a patch that was commited under rapid/traditional releases"; therefore, the timestamp of the first issue-resolving commit is used. For some metrics, issues were groupped by month and a time series was plotted to understand their variation over time.

Because we use distinct timestamps for distinct analyses, our sample also changes for each analysis. For instance, when analyzing issues created in the period under analysis, the sample contains 39770 issues; when analyzing issues which received an issue-resolving commit in the period, the sample contains 40419 issues.

To determine whether observed differences were statistically significant, we performed statistical tests such as Mann-Whitney's and Fisher's. The Mann-Whitney test was performed for efficiency and latency metrics, because these metrics are continuous and do not follow a normal distribution. Fisher's exact test was performed for effectiveness metrics, because those metrics are categorical. During the analysis, we talked to Mozilla engineers in the firefox-dev mailing list in order to better understand their process and to validate our early findings.

Results and discussion

All resolved issues that were created in the months of February, 2009, up to July, 2013, were analyzed, totaling 39770 issues. On average, on each day, 24.2 issues were resolved, 2.5 received at least one negative review, 2 had at least one commit backed out, and 1.4 were eventually reopened. The numbers show that patch rejection is a recurrent task that occurs on a daily basis.

Relationship between negative review, backout, and reopening

Figure 5 presents a Venn diagram displaying the proportion of resolved issues that underwent any combination of negative reviews, backouts, and reopenings. The number outside the circles, 80.4 %, is the proportion of issues in our data set that were not rejected. The sum of the proportions equals 100 % (except for rounding errors).

The diagram shows that most resolved issues are associated with a single rejection type, i.e., there are comparatively fewer issues that combine two or three rejection types. This result supports our observation that negative reviews, backouts, and reopenings are three

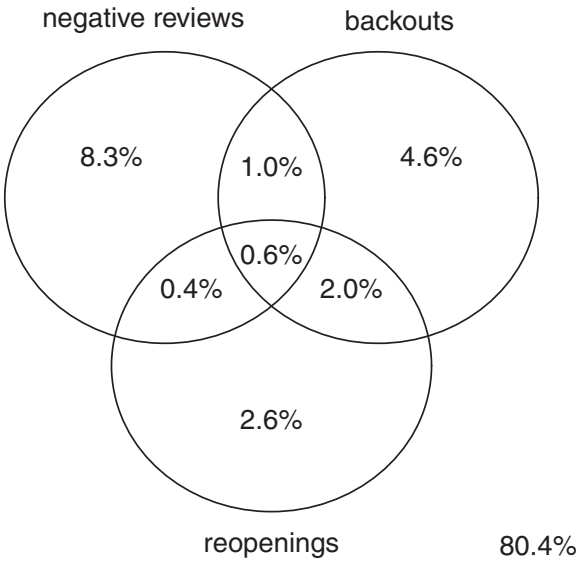

Fig. 5 Venn diagram of rejection types

essentially different phenomena (see the Background section) and that it is worthwhile to study all three to better understand patch rejection.

Still, the intersections on the diagram reveal useful information about the relationship between rejection types. Rejections in code review occur in 10.3 % (i.e., 8.3 % + 0.4 % + 1.0 % + 0.6 %) of all issues, and also, in 1.6 % of all cases, an issue is both rejected in code review and backed out. By dividing the latter by the former, we conclude that, although those reviews were rigorous (after all, they led to the rejection of a patch), 15.6 % of the time they failed to discover a problem that was later detected by automated or manual testing.

The diagram also shows that issue reopening has a significant intersection with backouts. This intersection is detailed in Fig. 6, which shows a Venn diagram of issue reopenings, early backouts, and late backouts (for simplicity, negative reviews are not explicit in the diagram). Since an issue can only be reopened after it is closed, which occurs when a patch is tested and commited, one would expect that reopened issues are often associated with a late backout. However, the diagram shows that, 68.1 % of the time, an issue reopening is premature: it is followed by some discussion that does not lead to the rejection (late backout) of the corresponding patch. The percentage was computed by dividing the proportion of issues that were reopened and not backed out late (3.3 % + 0.6 %) by the proportion of reopened issues (3.3 % + 0.6 % + 0.2 % + 1.6 %).

Patch effectiveness

Table 1 shows specific rejection rates in the period, i.e., what proportion of all resolved issues had at least one negative review, backout, or reopening. This is a measure of patch effectiveness, since it is related to the proportion of patches that were successful.

The table can be directly obtained by summing numbers obtained from Fig. 5 and from Fig. 6. For instance, the negative review rate is the sum of the four percentages inside the *negative review* circle in Fig. 5. Small discrepancies in the summation are due to the fact

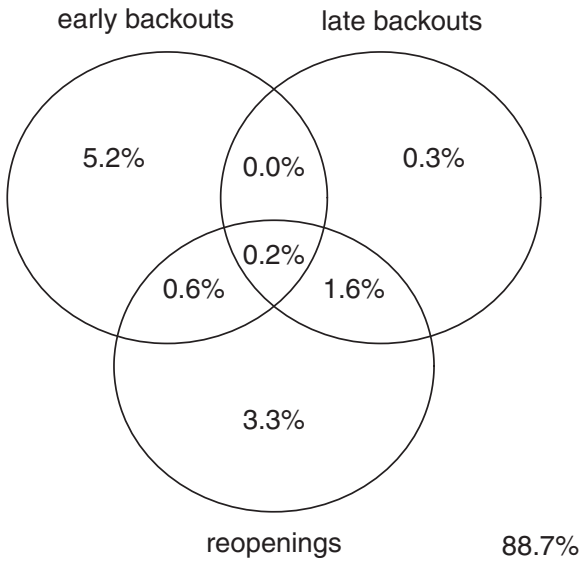

Fig. 6 Venn diagram for reopening and backout

Table 1 Metrics related to patch effectiveness

Metric	Value
Negative review rate	10.3 %
Backout rate	8.3 %
Early backout rate	6.1 %
Late backout rate	2.2 %
Reopening rate	5.7 %
Overall rejection rate	19.6 %

that all values are rounded to one decimal place. Notice that the overall rejection rate is smaller than the sum of negative review, backout and reopening, since it is possible for a single issue to be associated with more than one kind of rejection (e.g., backout and reopening).

The numbers show that inappropriate patches (i.e., patches that are eventually rejected) introduce a significant overhead in the process. About 19.6 % of all patches are rejected either by negative review, backout, or reopening (overall rejection rate), inducing overhead on developers, who have to write supplementary patches. Also, 10.3 % are rejected during peer review, which also induces overhead on reviewers, who have to review a supplementary patch, besides having to explain why the first one was rejected. Furthermore, 8.3 % of all resolved issues are backed out, a process that is performed nowadays by sheriffs (and, before June, 2011, by developers themselves). Although usually a simple operation, backouts can be hard when multiple commits are pushed together to the central repository; in this case, sheriffs must sort out which specific commit broke the build. Finally, about 5.7 % of all resolved issues are reopened, which induces a discussion overhead, since developers should discuss whether it was appropriate to reopen the issue.

Patch efficiency

As explained in the previous section, the *patch efficiency* perspective is concerned with how long it takes to propose a patch for an issue. Table 2 shows metrics on the time it takes to propose a patch in multiple contexts: the time to submit a patch to code review, the time to commit a patch to a source code repository, and the time to close an issue report. The metrics are also computed for two subsets of the data: inappropriate patches and supplementary patches. For each metric, the table shows the first quartile (25 %), the second quartile or median (50 %), and the third quartile (75 %).

Table 2 Metrics related to patch efficiency

	25 %	50 %	75 %
Time to submission (days)	0.3	1.1	12.9
(inappropriate patch)	0.4	4.6	35.7
(supplementary patch)	0.1	0.8	4.9
Time to commit (days)	1.4	6.8	30.8
(inappropriate patch)	2.8	13.0	51.9
(supplementary patch)	0.5	1.9	9.3
Time to closing (days)	3.0	10.0	39.7
(inappropriate patch)	4.8	16.4	58.7
(supplementary patch)	0.5	2.4	13.2

The first observation is that the distribution of the times is heterogeneous, heavily skewed to the right. That means that, while half of all patches is concentrated in a range of little times to submission (taking up to 1.1 day to be submitted), the other half is spread over a wider range of longer durations.

As expected, taking the time of issue creation as reference, commiting a patch takes longer than submitting a patch for peer review, and closing an issue report takes even longer. A patch is usually submitted shortly after the creation of an issue report; in half of the cases, it takes less than 1.1 day, and 75 % take at most 12.9 days. Half of all issue reports are closed within 10.0 days of its creation, while 75 % of issue reports are closed within 39.7 days.

The table also shows metrics for inappropriate and supplementary patches. Those metrics are discussed in the next subsections.

Inappropriate patches

By comparing the times of inappropriate patches with the times of patches in general (in bold in Table 2), we can see that, compared to patches in general, inappropriate patches take longer to submit, to commit, and the corresponding issue report takes longer to be closed. For instance, while 75 % of the issues take less than 12.9 days to have a patch submitted, if we consider only inappropriate patches, the time raises to 35.7 (Table 2), a 2.8x increase. The same trend shows up if we analyze the time needed to commit a patch or to close an issue report.

The implication of inappropriate patches taking longer to be submitted is that the overhead they cause is higher than what the rejection rate (Table 1) suggests. While 10.3 % of issue reports' first patches are rejected by peer review, they account for more than 17.4 % of the time needed to submit patches. Likewise, 8.3 % of all issues are associated with issue-resolving commits that were backed out, but those commits account for 11.5 % of the time that takes for patches to be committed after the creation of an issue report.

It should be noted that, with the available data, it is not possible to measure the time during which the developer was actually writing patches. Therefore, the time to submission and the time to commit of an issue includes the time in which the developer was resting or working on other issues.

Supplementary patches

The numbers on Table 2 also confirm the intuition that correcting an inappropriate patch takes less time than submitting the first patch for an issue. While in 75 % of the cases it takes up to 12.9 days to submit the first patch for an issue, it takes less than 4.9 days to submit a supplementary patch. Subsequent commits and closings also take significantly less time when compared to the first commit and the first closing of an issue report.

Intuition suggests that the greater the time between the submission of an inappropriate first patch and its rejection (i.e., latent time), the longer it would take to submit a supplementary patch. After all, a long latent time means that the developer's memory about the issue context is not as fresh as when he wrote the first patch. However, we could not find a significant correlation between latent time and the time to submit a supplementary patch. We attribute this lack of correlation to the fact that we cannot measure how much time was spent actually working on each patch, which is a subset of the time taken to submit it.

Also, we could not find a correlation between the time to submit an inappropriate patch and the time to submit a supplementary patch after the rejection of the first patch, tested

using Spearman's nonparametric correlation coefficient. The lack of correlation, in this case, suggests that the first and second patches are different in nature, and the effort needed to write the former is unrelated to that needed to write the latter.

Latent time and product quality

Table 3 shows the proportion of rejected issues that were latent for at most 12 hours, 24 hours, 1 week, and 6 weeks, before they were rejected by either a negative review, a backout, or a reopening. Latent times represent either the time between a patch submission and a negative review, the time between a commit and its backout, or the time between closing and reopening an issue report.

Latent times are usually low: about half of issues that are eventually rejected remain latent for 12 hours or less. More than 80 % of rejected issues are rejected within a week, and less than 7 % take more than 6 weeks to be rejected. Low latent times contribute to fast feedback cycles, which are valued in agile methodologies (Cockburn and Williams 2003).

Because inappropriate patches are rejected quickly, they are unlikely to be shipped with the code for a release. Therefore, based on the analysis of the data set, there are initial evidences that the quality of the product delivered is not affected by inappropriate patches.

Impact of rapid releases and integration repositories

In previous studies (Souza et al. 2014, 2015), we explored the evolution of patch rejection over time by analyzing issue reopening and backout metrics under two periods: traditional releases, from February 2009 to February 2011, and rapid releases, from July 2011 to July 2013. In this study, we replicate the analyses adding another form of patch rejection, negative code reviews.

Although the two periods have roughly the same number of days, the number of resolved issues under rapid releases was 1.8× higher (24536 vs 13434). This increase does not represent an increase in the productivity, since the number of active developers increased in the same proportion.

Patch effectiveness. Table 4 shows the variation of rejection rates under traditional and rapid releases. All differences are statistically significant ($p < 0.05$ using Fisher's exact test for count data).

As shown in previous studies, reopening rate and late backout rate decreased in the period, while early backout increased, supposedly because of integration repositories and better automated testing. In this study, we show that the negative code review rate also decreased.

Figure 7 presents the evolution of the negative review rate over time. It shows a decreasing trend of the metric in the rapid release period.

Patch efficiency. Table 5 shows median values for time-related metrics under traditional and rapid releases. The number of asterisks correspond to statistical significance ($* \Rightarrow p < 0.05, ** \Rightarrow p < 0.01, *** \Rightarrow p < 0.001$).

Table 3 Metrics related to bug latency

	12 h	24 h	1 week	6 weeks
% of negative reviews that occur within...	49.5 %	61.9 %	88.7 %	98.3 %
% of backouts that occur within...	61.1 %	67.6 %	84.0 %	94.6 %
% of reopenings that occur within...	50.8 %	59.2 %	82.4 %	93.9 %

Table 4 Metrics related to patch effectiveness under both traditional and rapid releases

	Traditional	Rapid
Negative review rate	12.2 %	10.1 %
Backout rate	6.1 %	9.2 %
Early backout rate	2.8 %	7.7 %
Late backout rate	3.3 %	1.6 %
Reopening rate	7.8 %	4.7 %

Under rapid releases, the median times to submit or commit a patch, or to close an issue report, were all reduced. Figure 8 shows that the metrics started to decrease after the adoption of rapid releases and integration repositories, which suggests that they played a role in this reduction.

Regarding code reviews, supplementary patches took approximately the same amount of time to submit under rapid releases. The numbers suggest that submitting a supplementary patch was already a fast task, and was not affected by either rapid releases or integration repositories.

While the time to commit and the time to closing became about twice as fast under rapid releases, the reduction in time to submission was much smaller. This result suggests that the changes that Firefox underwent in 2011 affected mainly later stages of the change process.

Threats to validity

As in any empirical study, the validity of the results presented in this paper is subject to threats. The most relevant threats to construct, internal, and external validity are described below.

Construct validity (to which extent the study measures what it intends to measure). The detection of backouts and patches was based on heuristics, including pattern matching

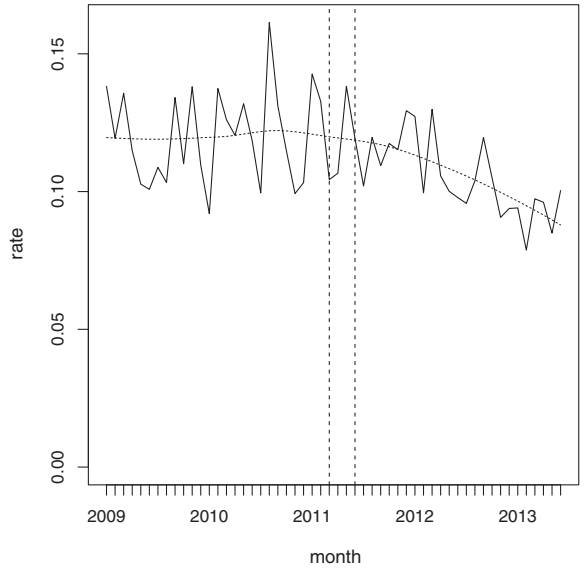

Fig. 7 Evolution of negative review rate

Table 5 Metrics related to patch efficiency under both traditional and rapid releases

	Traditional	Rapid	Significance
Time to submission (days)	1.3	0.9	***
(inappropriate patch)	4.9	4.5	**
(supplementary patch)	0.8	0.8	
Time to commit (days)	11.1	5.4	***
(inappropriate patch)	22.2	10.1	***
(supplementary patch)	3.4	1.6	***
Time to closing (days)	15.9	7.7	***
(inappropriate patch)	20.7	14.3	***
(supplementary patch)	2.8	2.1	*

on commit messages. Although numerous messages were read in order to determine recurrent patterns to help extract issue identifiers, not all commits conform to the most common patterns. It is possible that the actual backout rate is different from what was measured, although we believe that the deviance would not be large.

Also, in our analyses, we considered only the first negative review, the first backout, and the first reopening of each issue report. We believe this simplification did not change significantly the results, since few issue reports contain more than one ocurrence of each of these events.

There are limitations on what can be measured with the available data. In particular, we cannot measure the time each developer spent writing patches. As an imperfect substitute, we measure the time between the creation of an issue report and the submission of a corresponding patch. Also, it is possible that a developer discovered an issue and started writing a patch even before reporting it on Bugzilla.

Internal validity (to which extent conclusions can be made from what was measured). The observed differences under traditional and rapid releases could be attributed to factors other than release length, since the development process may have changed over time

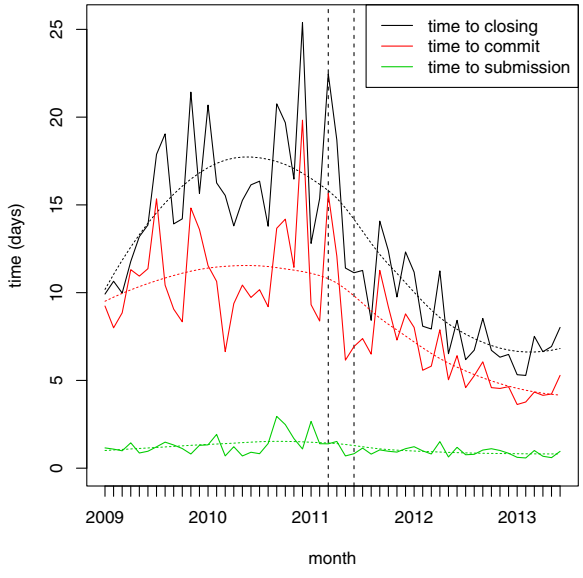

Fig. 8 Submitted patches over time

in many different ways. In this paper, we already identified factors such as improvements in testing infrastructure, the introduction of sheriff-managed integration repositories. Other factors may also play a role. For example, a Mozilla engineer pointed out that the criteria used to determine when an issue should be reopened and when a new issue report should be filled can change over time and influence the analysis of issue reopening. To mitigate this threat, we have plotted the variation of metrics over time to visualize whether there were significant variations within a release type.

External validity (to which extent results can be generalized). All the conclusions in this study were based on data from one single open source project. The results are not expected to be generalizable to other projects; instead, the purpose of this study is to provide insights on the questions beings studied. Furthermore, it is not trivial to extend the study to other projects, since interpreting the results require a high level of understanding about the process, in a detailed perspective, as well as about process changes over time.

Conclusions

In this paper we described an empirical study to characterize the rejection of inappropriate patches, an indicator of process quality, in Mozilla Firefox. We have looked at three sources for studying patch rejections: code review, backouts—further divided into early and late backouts—and issue reopening. The importance of this strategy revealed itself when trends diverged. For instance, when assessing the impact of rapid releases and integration repositories on rejection rate, some metrics increased, while others decreased.

Also, we discussed early results with Mozilla engineers in the firefox-dev mailing list, some of which were surprising for them. Had we analyzed only one source of information about rejections or relied only on quantitative data, we could have reached incomplete and biased conclusions.

The main findings of this study are summarized below:

- patch rejection happens daily, on average;
- rejection by negative review, backout, and issue reopening are three events that, although related, are relatively independent;
- almost 70 % of issue reopenings are premature, because they are not accompanied by the backout of the corresponding patch;
- almost 20 % of all resolved issue reports contain at least one patch that is eventually rejected, introducing overhead in the process;
- inappropriate patches take longer to be submitted when compared to patches in general;
- supplementary patches are submitted faster than initial patches;
- the time it takes to submit a supplementary patch is not correlated with the time it took to submit the first, inappropriate patch, nor with the time it took for the inappropriate patch to be rejected;
- in 75 % of the cases, it takes less than 4 days to reject an inappropriate patch;
- for the reason above, inappropriate patches are unlikely to become part of a final release;
- the late backout rate, the reopening rate, and the negative review rate decreases under rapid releases;
- the time to submit a patch decreases under rapid releases.

We believe that this study contributes both to scientific literature and to software development practice in the following ways: (i) it compares negative review, backout, and issue reopening, three events related to patch rejection that are usually studied separately; (ii) it helps understand the overhead caused by inappropriate patches that are rejected; (iii) it provides a detailed account on how patch rejection metrics evolved in a project that adopted a rapid release model; (iv) it unveils facts previously unknown to Firefox engineers that reinforce their perception that changes introduced in 2011 were benefical from a technical point of view; (v) it reveals practices adopted by Firefox that were successful in improving process quality while speeding up the release of new versions.

In a future work, we intend to study code reviews in greater detail, especially investigating to what extent they anticipate problems that would otherwise be detected during automated testing. Such study would cast light on the relationship between code review and automated testing, two important methods for early problem detection.

Endnotes

[1]http://www.bugzilla.org/.

[2]http://mercurial.selenic.com/.

[3]The Mercurial hook was proposed at https://bugzilla.mozilla.org/show_bug.cgi?id=506949.

Competing interests

The authors declare that they have no competing interests.

Authors' contributions

RRGS worked on the study design and implementation, interpreted the results, and drafted the manuscript. CFGC and RAB supported the study design and revised the manuscript. All authors read and approved the final manuscript.

Authors' information

Rodrigo Souza holds a MSc from the the Department of Systems and Computing (DSC) of the Federal University of Campina Grande (UFCG), Brazil. He is also a PhD candidate at the Department of Computer Science (DCC) of the Federal University of Bahia (UFBA), Brazil. His research interests include empirical software engineering, release engineering, software evolution, and mining software repositories.

Christina Chavez is an Associate Professor at the Department of Computer Science (DCC) of the Federal University of Bahia (UFBA), Brazil. She obtained her PhD in Computer Science at the Pontifical Catholic University of Rio de Janeiro (PUC-Rio), Brazil, in 2004. Her research interests include software architecture, software evolution, software engineering education, and also topics on software mining and archaeology, documentation, software architecture recovery and assessment, design principles and patterns, and agile methods and practices.

Roberto A. Bittencourt is an Assistant Professor at the Computer Engineering Program of the State University of Feira de Santana (UEFS), Brazil. He obtained his PhD in Computer Science at the Federal University of Campina Grande (UFCG), in cooperation with the University of British Columbia (UBC), Canada, in 2012. His research interests include collaboration systems, recommendation systems, social network analysis, computing education, software design and architecture, and software evolution and maintenance.

Acknowledgements

The authors would like to thank all Firefox engineers who provided feedback on early results of this research.

Author details

[1] Department of Computer Science, Federal University of Bahia, Salvador, Brazil. [2] Department of Exact Sciences, State University of Feira de Santana, Feira de Santana, Brazil.

Received: 8 December 2014 Accepted: 9 June 2015
Published online: 18 June 2015

References

Almossawi A (2012) Investigating the Architectural Drivers of Defects in Open-source Software Systems: an Empirical Study of Defects and Reopened Defects in GNOME. Master thesis, Massachusetts Institute of Technology, 2012

An L, Khomh F, Adams B (2014) Supplementary bug fixes vs. re-opened bugs. In: Source Code Analysis and Manipulation (SCAM), IEEE 14th International Working Conference On. IEEE, Washington, DC, USA. pp 205–214

Bird C, Gourley A, Devanbu P (2007) Detecting patch submission and acceptance in oss projects. In: Proceedings of the Fourth International Workshop on Mining Software Repositories. MSR 2007. IEEE Computer Society, Washington, DC, USA. p 26. doi:10.1109/MSR.2007.6

Bugzilla (2015) Bug Fields. https://bugzilla.mozilla.org/page.cgi?id=fields.html Accessed 13/06/2015

Chapin N, Hale JE, Kham KM, Ramil JF, Tan WG (2001) Types of software evolution and software maintenance. J Softw Maint 13(1):3–30
Cockburn A, Williams L (2003) Agile software development: It's about feedback and change. Computer 36(6):0039–43
Herzig K, Just S, Zeller A (2013) It's not a bug, it's a feature: How misclassification impacts bug prediction. In: Proceedings of the 2013 International Conference on Software Engineering. ICSE '13. IEEE Press, Piscataway, NJ, USA. pp 392–401
Jeong G, Kim S, Zimmermann T, Yi K (2009) Improving code review by predicting reviewers and acceptance of patches. Technical Report ROSAEC MEMO 2009-006
Jongyindee A, Ohira M, Ihara A, Matsumoto KI (2011) Good or bad committers? a case study of committers' cautiousness and the consequences on the bug fixing process in the Eclipse project. In: Proc. of the 2011 Joint Conf. of the 21st International Workshop on Softw. Measurement and the 6th International Conf. on Softw. Process and Product Measurement. IEEE Computer Society, Washington, DC, USA. pp 116–125. doi:10.1109/IWSM-MENSURA.2011.24
Khomh F, Dhaliwal T, Zou Y, Adams B (2012) Do faster releases improve software quality? an empirical case study of mozilla firefox. In: Lanza M, Pent MD, Xi T (eds). 9th IEEE Working Conference on Mining Software Repositories, MSR 2012. IEEE, Washington, DC, USA. pp 179–188
Lehman MM, Ramil JF, Wernick PD, Perry DE, Turski WM (1997) Metrics and laws of software evolution-the nineties view. In: Software Metrics Symposium, 1997. Proceedings., Fourth International. IEEE, Washington, DC, USA. pp 20–32
Mantyla MV, Khomh F, Adams B, Engstrom E, Petersen K (2013) On rapid releases and software testing. In: Software Maintenance (ICSM), 29th IEEE International Conference On. IEEE Computer Society, Washington, DC, USA. pp 20–29. doi:10.1109/ICSM.2013.13
Mozilla (2011) Mozilla Release Processes. http://mozilla.github.io/process-releases/ Accessed 13/06/2015
Mozilla (2014) Mozilla wiki: Releases. https://wiki.mozilla.org/Releases Accessed 13/06/2015
Nurolahzade M, Nasehi SM, Khandkar SH, Rawal S (2009) The role of patch review in software evolution: An analysis of the mozilla firefox. In: Proceedings of the Joint International and Annual ERCIM Workshops on Principles of Software Evolution (IWPSE) and Software Evolution (Evol) Workshops. IWPSE-Evol '09. ACM, New York, NY, USA. pp 9–18. doi:10.1145/1595808.1595813
Park J, Kim M, Ray B, Bae DH (2012) An empirical study of supplementary bug fixes. In: 9th IEEE Working Conference on Mining Software Repositories. IEEE Computer Society, Washington, DC, USA. pp 40–49. doi:10.1109/MSR.2012.6224298
Plewnia C, Dyck A, Lichter H (2014) On the influence of release engineering on software reputation. In: 2nd International Workshop on Release Engineering. http://releng.polymtl.ca/RELENG2014/html/proceedings/releng2014_submission__3.pdf
Rigby PC, Storey MA (2011) Understanding broadcast based peer review on open source software projects. In: Proceedings of the 33rd International Conference on Software Engineering. ACM, New York, NY, USA. pp 541–550
Rigby PC, German DM, Cowen L, Storey MA (2014) Peer review on open source software projects: Parameters, statistical models, and theory. ACM Trans Softw Eng Methodol 34
Shihab E, Ihara A, Kamei Y, Ibrahim WM, Ohira M, Adams B, Hassan AE, Matsumoto KI (2010) Predicting re-opened bugs: A case study on the eclipse project. In: Proceedings of the 2010 17th Working Conference on Reverse Engineering. WCRE '10. IEEE Computer Society, Washington, DC, USA. pp 249–258. doi:10.1109/WCRE.2010.36
Shihab E (2012) An exploration of challenges limiting pragmatic software defect prediction. PhD thesis, School of Computing, Queen's University, Kingston, Ontario, Canada
Souza R, Chavez C, Bittencourt RA (2014) Do rapid releases affect bug reopening? A case study of Firefox. In: Software Engineering (SBES), 2014 Brazilian Symposium On. IEEE Computer Society, Washington, DC, USA. pp 31–40. doi:10.1109/SBES.2014.10
Souza R, Chavez C, Bittencourt R (2015) Rapid releases and patch backouts: A software analytics approach. Softw IEEE 32(2):89–96. doi:10.1109/MS.2015.30
Tao Y, Han D, Kim S (2014) Writing acceptable patches: An empirical study of open source project patches. In: 30th IEEE International Conference on Software Maintenance and Evolution, Victoria, BC, Canada, September 29 - October 3, 2014. IEEE Computer Society, Washington, DC, USA. pp 271–280. doi:10.1109/ICSME.2014.49
Zimmermann T, Nagappan N, Guo PJ, Murphy B (2012) Characterizing and predicting which bugs get reopened. In: Proceedings of the 2012 International Conference on Software Engineering. ICSE 2012. IEEE Press, Piscataway, NJ, USA. pp 1074–1083

Submit your manuscript to a SpringerOpen journal and benefit from:

- ► Convenient online submission
- ► Rigorous peer review
- ► Immediate publication on acceptance
- ► Open access: articles freely available online
- ► High visibility within the field
- ► Retaining the copyright to your article

Submit your next manuscript at ► springeropen.com