# Is It Safe to Uplift This Patch?
## An Empirical Study on Mozilla Firefox

Marco Castelluccio
Mozilla Corporation, United Kingdom
DIETI, Università Federico II, Italy
mcastelluccio@mozilla.com

Le An and Foutse Khomh
SWAT Lab
Polytechnique Montréal, QC, Canada
{le.an, foutse.khomh}@polymtl.ca

***Abstract*—In rapid release development processes, patches that fix critical issues, or implement high-value features are often promoted directly from the development channel to a stabilization channel, potentially skipping one or more stabilization channels. This practice is called *patch uplift*. Patch uplift is risky, because patches that are rushed through the stabilization phase can end up introducing regressions in the code. This paper examines patch uplift operations at Mozilla, with the aim to identify the characteristics of uplifted patches that introduce regressions. Through statistical and manual analyses, we quantitatively and qualitatively investigate the reasons behind patch uplift decisions and the characteristics of uplifted patches that introduced regressions. Additionally, we interviewed three Mozilla release managers to understand organizational factors that affect patch uplift decisions and outcomes. Results show that most patches are uplifted because of a wrong functionality or a crash. Uplifted patches that lead to faults tend to have larger patch size, and most of the faults are due to semantic or memory errors in the patches. Also, release managers are more inclined to accept patch uplift requests that concern certain specific components, and–or that are submitted by certain specific developers.**

***Index Terms*—Patch uplift, Urgent update, Mining software repositories, Release engineering**

## I. Introduction

The advent of continuous delivery and rapid release practices have significantly reduced the amount of stabilization time available for new features, forcing companies to resort to innovative techniques to ensure that important features are released to the public, in a timely manner and with a good quality. To cope with short release cycles, Mozilla has reorganized its release process around four channels: a development channel named *Nightly*, two stabilization channels (*Aurora* and *Beta*), and a main *Release* channel. Features corresponding to a new release are developed on the Nightly channel over a period of six weeks. After that, the code is transferred to Aurora, where it is tested by Mozilla developers and contributors, for a period of six weeks, and then to Beta where it is tested by a selected group of external users. Finally, mature Beta features are imported into the main Release channel and delivered to end users. This pipelined process allows Mozilla to avoid mixing the development of new features with the stabilization process, which is particularly important given that integration operations are unpredictable [1], and can significantly delay a release process, if not enough time is allowed for stabilization. However, this well organized release process is frequently subverted by urgent patches, implementing high-value features or critical fixes, that cannot wait for the next release train. These features and fixes are directly promoted from the development channel to stable channels (*i.e.*, Aurora, Beta, and main Release), a practice called *patch uplift*. Patch uplift is risky because the time allowed for the stabilization of uplifted patches is reduced by six weeks for each skipped channel. Therefore, it is important to carefully pick the patches that are uplifted and ensure that developers scrutinize them properly, to reduce the risk of regressions. There are a set of rules in place at Mozilla to govern this uplift process. One of these rules is that patches uplifted to the Beta channel should be *(1) ideally reproducible by the QA team, so that they can be verified; (2) should have been verified on Aurora/Nightly first; and (3) should not contain string changes (i.e., changes in the text which is visible to users)*. However, despite these rules, multiple uplifted patches still introduce regressions in the code. Hence, it is unclear if–and–how the rules are enforced at Mozilla and why certain uplifted patches introduce post-release bugs.

In this paper, we conduct a series of quantitative and qualitative analyses to understand the decision making process of patch uplift at Mozilla and the characteristics of uplifted patches that introduce regressions. Overall, we analyze 33,664 issue reports (corresponding to 7,267 uplift requests) in 17 versions of Firefox over a period of two years and answer the following research questions:

*RQ1: What are the characteristics of patches that are uplifted?*

We observe that most patches are uplifted to resolve wrong functionalities or crashes. Rejected uplift requests required longer decision time than accepted requests. We attribute this difference to the high complexity of these rejected patches (since complex patches require longer time for risk assessment). Last but not least, release managers tend to trust patches that concern certain specific components, and–or that are submitted by certain specific developers.

*RQ2: What are the characteristics of uplifted patches that introduced faults in the system?*

From our analysis, we observe that uplifted patches that lead to faults tend to have larger patch size; suggesting that developers and release managers need to carefully review patch candidates for uplift with a large amount of changes, before allowing for their uplift. Most faulty uplifts are due to semantic or memory-related errors. We also observed that patches related to certain components and–or submitted by certain developers are more likely to cause faults.

**The remainder of this paper is organized as follows.** Section II provides background information about patch uplift. Section III describes the design of our case study. Section IV presents the results of the case study, and Section V elaborates on the implications of these results. Section VI discusses threats to the validity of this study. Section VII summarizes related works, and Section VIII concludes the paper.

## II. Mozilla Patch Uplift Process

This section describes the Mozilla patch uplift process and the rules governing this process.

Firefox follows a pipelined release process [2], with four release channels (*Nightly*, *Aurora*, *Beta*, and *Release*). New feature work is done on the *Nightly* channel, while *Aurora* and *Beta* serve as stabilization channels, and the *Release* channel is used to deliver the software to end users. Every six weeks, there is a *merge day*, when the code from a less stable channel flows into a more stable one (*e.g.*, the Nightly code is moved in the Aurora repository). Most of the development work is performed in the *Nightly* channel, where patches can be committed after a normal review process. For the stabilization channels, a different process for committing patches has been put in place (*i.e.*, patch uplift), to keep the channels as stable as possible (as code committed to Aurora and Beta is closer to be released to users). Patches with important features or severe fault fixes that cannot wait for the entire process are promoted directly from the development channel to one of the stable channels, skipping the stabilization phase on one or more channels.

The lifecycle of an uplifted patch can be summarized as follows: developers write a patch, which gets reviewed by one or more reviewers. After a successful review, the patch is committed to the Nightly channel. If developers (or other stakeholders) believe that the patch is particularly important (*e.g.*, it fixes a frequent crash, or a performance issue), they can ask for approval to uplift the patch to one (or more) of the stable channels, *i.e.*, Aurora, Beta, or Release.

Release managers (who are independent and different from reviewers) are responsible for deciding which patches can be uplifted. They can either *accept* or *reject* the patch uplift request, after a careful consideration of the risks involved.

The more a channel is stable, the higher is the bar for approval of uplift requests. Below we present an excerpt of the rules in place at Mozilla on the different channels.
*Aurora*: Uplifts to the Aurora channel are less critical, as they still have considerable time for stabilization. The rules are not strict in this case: no new features are accepted; no disruptive refactorings; no massive code changes; no string changes, unless the localization team is aware and has approved; they must be accompanied, if possible, by automated tests.
*Beta*: Uplifts to the Beta channel are more critical, as they have less time for stabilization. In addition to the rules outlined for Aurora, the changes uplifted to the Beta channel should be (1) ideally reproducible by QA, so that they can be verified; (2) they should have been verified on Aurora/Nightly first; and should not contain (3) changes to the user-visible strings in the application (as those require a very high effort and time to be localized, since Mozilla relies on volunteer contributors). The uplifted changes can be proven performance improvements, fixes to important crashes, fixes for recent regressions. The closer to the release date, the stricter the release managers should be in enforcing the rules.
*Release*: Uplifts to the Release channel are generally discouraged, as they require a new version to be built and released to users. Possible uplifts are fixes for major top crashes, security issues, functional regressions with a very broad impact.

Once a patch is accepted for uplift, Tree Sheriffs [3] (*i.e.*, engineers responsible for supporting developers in committing patches and ensuring that the automated tests are not broken after commits, monitoring intermittent failures and backing out patches in case of test failures) or the developers themselves can commit it to the stabilization channel(s) for which the patch was approved.

## III. Case Study Design

In this section, we describe the data collection and analysis approaches that we use to answer our two research questions.

### A. Data Collection

We collect, from the Mozilla issue tracking system (Bugzilla), all issues marked as *resolved* or *verified* in the Firefox and Core products between July 2014 (release date of Firefox 31.0) and August 2016 (release date of Firefox 48.0). In total, there are 35,826 issue reports in our dataset.

Mozilla developers use customized Bugzilla flags to request for patch uplifts. These flags have the form `approval-mozilla-CHANNEL`, where `CHANNEL` can be Aurora, Beta, or Release. The postfix of the flag is set to a question mark (`?`) when a developer asks for an uplift, to a minus sign (`-`) if the release manager rejects the uplift, and to a plus sign (`+`) if the release manager approves the uplift. We rely on these flags to identify uplifted patches. At Mozilla, release managers usually inspect all patches in an issue report before deciding whether they can be uplifted together. Thus, in this work, we consider uplift characteristics at the issue level. If an issue contains multiple patches, we bundle the patches together. To study the patch uplift process, we need to consider a period of time during which the practice was well established at Mozilla. To decide on this period, we computed the amount of patches that were uplifted each month, over our initial period of July 2014 to August 2016. Figure 1 shows the distribution of the number of uplifts in three Firefox's release channels during this period. We do not consider uplifts that concern the "Pocket" component, as the inclusion of Pocket (which is a third-party add-on) in Firefox, a one-time event, might introduce noise in our data. In Figure 1, each time point

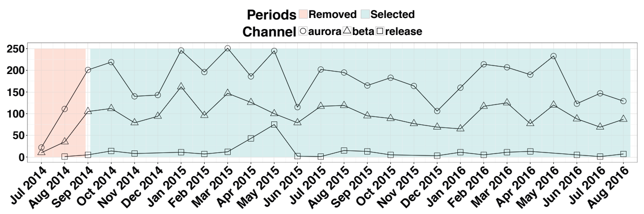

Figure 1: Number of uplifts during each month from July 2014 to August 2016. Periods with low number of uplifts or not covering all the three channels are removed.

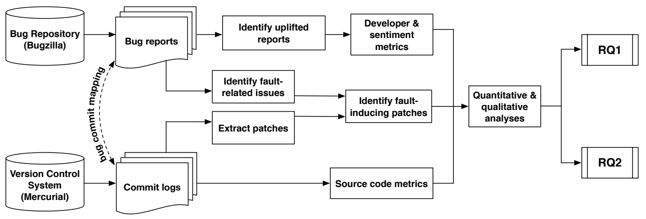
Bug Repository (Bugzilla)
Bug reports
bug commit mapping
Version Control System (Mercurial)
Commit logs
Identify uplifted reports
Identify fault-related issues
Extract patches
Developer & sentiment metrics
Identify fault-inducing patches
Source code metrics
Quantitative & qualitative analyses
RQ1
RQ2

Figure 2: Overview of our data processing approach.

represents a period of one month (we can see that the Release channel did not receive any uplift in May and November 2015). Figure 1 shows that the number of uplifted patches increased from July 2014 to August 2014 and then became stable from September 2014 to August 2016. Based on this distribution, we selected the period between September 2014 and August 2016, for our study. In other words, we limited our dataset to only issue reports and commits that occurred within this period. Between September 2014 and August 2016, we study in total 33,664 issue reports, in which there are 7,267 uplift requests: 285 to Release, 2,614 to Beta, and 4,368 to Aurora.

## B. Data processing

Figure 2 shows a general overview of our approach. We describe each step of the approach below. The corresponding data and scripts are available online at: https://github.com/swatlab/uplift-analysis.

*1) Identification of Fault-related Issues:* Mozilla uses Bugzilla to manage and track its issues. All types of issues, whether they are faults or new features, are managed in this system. Unlike JIRA [4], which offers the possibility to distinguish between issues using a tag, Bugzilla does not provide issue type information. Therefore, our first processing task is to differentiate issues that are related to faults, from new feature requests or improvements. To automatically identify fault-related issues, we use a keyword-based heuristic to search information in the title, description, flags, and user comments of each issue report. Our list of keywords includes: crash, regression, failure, leak, steps to reproduce (STR), and hang. The full list is available at: https://github.com/swatlab/uplift-analysis.

To ensure the accuracy of our detection on fault-related issues, we manually validated a sample of our results. From a total of 33,664 issue reports, we randomly selected a sample of 380 issue reports, which corresponds to a confidence level of 95% and a confidence interval of 5%. The first and the second authors read each of the 380 issue reports independently and classified them into *fault-related* and *other* categories. We then compared their classification results and observed that 41 issue reports were classified into different categories by the two authors. To resolve these discrepancies, we created an online document for the 41 issues; allowing all of the authors to comment and discuss the issues. After this round, a consensus was reached for 35 out of the 41 issues. For the remaining 6 issues, we organized a meeting and discussed the classification of each of them until a consensus was found. The result of our manual classification shows that our keyword-based heuristic achieves a precision of 87.3% and a recall of 78.2%, when classifying issues into *fault-related* and *other* categories.

*2) Identification of Fault-inducing Patches:* We use the SZZ algorithm [5] to identify patches (these patches could be fault-fixing patches or patches related to features or improvements) that introduced faults in the system. First, we used Fischer et al.'s heuristics [6] to map each studied issue to its corresponding patch(es) (*i.e.*, commits). This heuristic consists in looking for issue IDs in commit messages using regular expressions. Next, for each fault-related issue, we use the following Mercurial command to extract the list of files that were changed to fix the issue:

```
hg log --template {commit},{file_mods},{file_dels}
```

In this step, we only consider modified and deleted lines, since added lines could not have been changed by prior commits. We denote an issue's fault-fixing file by $F_{fix}$. Then, for each changed file $f_{fix} \mid f_{fix} \in F_{fix}$, we use Mercurial's `annotate` command as follow to check which prior commits changed the lines that were modified by the fault-fixing commits. The SZZ algorithm assumes that the fault is located in these lines.

```
hg annotate commit^ -r f_fix -c -l -w -b -B
```

We refer to the obtained commits as *fault-inducing candidates*. Finally, we examine whether a fault-inducing candidate was submitted before the creation date of its corresponding fault-related issue report. If so, we consider the candidate to be a *fault-inducing commit*, and its related issue to be a *fault-inducing issue*.

*3) Mining Issue Reports:* We mine several kinds of metrics from Bugzilla issue reports: information about the review process (*e.g.*, how long a review took, how many reviewers inspected a patch), information about the uplift process (*e.g.*, whether an uplift was accepted, how long before a release manager decided to accept or reject an uplift request), the developer assigned to an issue, and the component(s) affected by an issue.

*4) Computing Metrics:* To capture the characteristics of patches that were uplifted, we computed the 22 metrics described in Table I. These metrics correspond to the following five dimensions:

Developer experience and participation metrics. Our rationale for computing these metrics is that patches written or reviewed by experienced developers may have a higher

Table I: Metrics used to compare patches.

| Metric | $m_i$ | Description |
|---|---|---|
| **Developer experience and participation metrics** ($m_1$ - $m_5$) | | |
| Developer experience | 1 | Number of previous commits of the patch developer. |
| Reviewer experience | 2 | Number of previous commits of the patch reviewer. |
| Number of comments | 3 | Number of comments in the issue report. |
| Comment words | 4 | Average number of words in the comments to an issue. |
| Review duration | 5 | Time period (in days) from a patch's submission until its approval. |
| **Uplift process metrics** ($m_6$ - $m_8$) | | |
| Landing delta | 6 | Time elapsed (in days) between when the patch was applied to the Nightly version and when the developer asked for approval of an uplift. |
| Response delta | 7 | Time elapsed (in days) between when the developer asked for approval for the uplift and when the release manager decided (approved or rejected). |
| Release delta | 8 | Time elapsed (in days) between when the developer asked for approval for the uplift and the date of the following release. |
| **Sentiment metrics** ($m_9$ - $m_{10}$) | | |
| Developer sentiment | 9 | The highest negative sentiment score in the developers' comments on an issue. |
| Owner sentiment | 10 | The highest negative sentiment score in module owners' comments on an issue. |
| **Code complexity metrics** ($m_{11}$ - $m_{19}$) | | |
| Patch size | 11 | Number of lines in a patch (excluding test patches). |
| Test patch size | 12 | Number of lines in a test patch. |
| Prior changed times | 13 | Number of previous commits that modified the same files that the patch is modifying. |
| LOC | 14 | Average lines of code in all classes in a patch. |
| Average cyclomatic | 15 | Average cyclomatic complexity of the functions in a class. |
| Number of functions | 16 | Average number of classes' functions in a patch. |
| Maximum nesting | 17 | Average maximum level of nested functions in all classes in a patch. |
| Comment ration | 18 | Average ratio of the lines of comments over the total lines of code in all classes in a patch. |
| Module number | 19 | Number of modules involved by a patch. |
| **Code centrality (SNA) metrics** ($m_{20}$ - $m_{22}$) | | |
| PageRank | 20 | Time fraction spent to "visit" a class in a random walk in the call graph. |
| Betweenness | 21 | Number of classes passing through a class among all shortest paths. |
| Closeness | 22 | The average length of the shortest path between a class and all other classes. |

chance to be accepted for uplift, and may be less fault-prone. Long comments and long review durations may indicate the complexity of an issue and developers' uncertainty about it, which may explain its rejection or fault-proneness.

Uplift process metrics. We compute metrics capturing the uplift process for the following reasons. Release managers may be more inclined to accept patches with higher landing delta (as the more time a patch has been on the Nightly channel, the more time it has been tested by Nightly users). Patches with low release delta are likely to be refused uplifts, since patches that are developed closer to the date of release might pose more risk (as there is less time to fix potential regressions). Patches with low response delta may also be rejected (since developers have less time to evaluate the risks associated with the patch). Patches with low landing delta, release delta, and low response delta may also lead to faults if uplifted.

Sentiments. We compute sentiment metrics because we believe that sentiments can affect uplift decisions and their success rate. From each studied issue, we extract developers' comments to compute their sentiments. We leverage the sentiment mining tool, *SentiStrength* [7], to estimate the extent of developers' positive and negative sentiments toward a specific issue. As one of the state-of-the-art sentiment mining tool, SentiStrength is easy to apply, and it has achieved a reasonable performance in prior works [7]. In addition to developers' sentiments, we also computed module owners' sentiments.

Code Complexity. Previous works, such as [8], have shown that complex code is likely to introduce faults. We calculate code complexity metrics to understand how uplifting decisions and their success are affected by the complexity of the uplifted patches. We extract the files changed in each patch and use the static code analysis tool *Understand* [9] to calculate the following complexity metrics on the files: lines of code (LOC), average cyclomatic complexity, number of functions, maximum nesting, and ratio of the comment lines over the total code lines.

Code centrality (SNA) metrics. Kim et al. [8] observed that functions close to the centre of a call graph are likely to experience more faults. Hence, we compute metrics capturing the centrality of functions involved in uplifted patches and uplifted patch candidates. We use the network analysis tool, *igraph* [10], in combination to *Understand* [9], as in [11], to compute the following *Social Network Analysis* (SNA) metrics: PageRank, betweenness, and closeness. When computing complexity and SNA metrics, we only consider the C/C++ code since Firefox contains 86% of C/C++ code. Computing code complexity and SNA metrics is a very time-consuming task. Instead of computing the metrics for each patch, we compute metrics by releases and map a given patch to its latest major release as in our previous work [11]. To make the metric results as precise as possible, we consider all major releases from Firefox 32.0 until Firefox 48.0, which cover the system's history from September 2014 until August 2016.

## IV. Case Study Results

This section presents and discusses the results of our two research questions. For each question, we discuss the motivation, the approach designed to answer the question, and the findings. To get a deeper insight of the patch uplift process, we perform both quantitative and qualitative analyses for each research question.

*RQ1: What are the characteristics of patches that are uplifted?*

***Motivation.*** This question aims to understand the characteristics of patches that are uplifted. We are particularly interested in understanding what differentiates patch uplifts

Table II: Accepted vs. rejected patch uplift candidates.

| Channel | Metric | Accepted | Rejected | $p$-value | Effect size |
|---|---|---|---|---|---|
| *Aurora* | Comment ratio | 0.1 | 0.2 | **0.03** | small |
| | Landing delta | 0.4 | 3.0 | **0.02** | small |
| | Response delta | 0.9 | 2.4 | **1.80e-05** | medium |
| *Beta* | LOC | 529.0 | 1,046.8 | **9.27e-04** | small |
| | Cyclomatic | 2.0 | 3.0 | **0.04** | negligible |
| | # of functions | 20.0 | 35.2 | **9.62e-04** | small |
| | Comment ratio | 0.1 | 0.2 | **8.86e-05** | small |
| | Betweenness | 2,789.0 | 20,586.3 | **0.01** | negligible |
| | PageRank | 1.4 | 1.7 | **0.01** | negligible |
| | Max. nesting | 2.3 | 3.0 | **7.72e-03** | negligible |
| | Module number | 1.0 | 1.0 | **7.13e-03** | negligible |
| | Response delta | 0.7 | 1.0 | **6.28e-04** | small |
| *Release* | Response delta | 0.02 | 3.1 | **1.39e-12** | large |

among different channels. Although Mozilla has published rules to guide the patch uplift process [12], it is unclear if and how these rules are enforced in practice. The answer to this research question can help discover hidden factors that affect the uplift process, and help software practitioners make this process more predictable.

### 1) Quantitative Analysis.

***Approach.*** Using the metrics from Table I, we statistically compare 22 numerical characteristics of patch uplift candidates that were accepted and those that were rejected. As Mozilla release managers take a whole issue report into account during the uplift process (see Section III-A), we calculate the average values of the code complexity and SNA metrics for all patches in a subject issue report.

For each of the 22 metrics $m_i$, we formulate the following null hypothesis:
$H_i^{01}$: *there is no difference between the values of* $m_i$ *for patch uplift candidates that were accepted and those that were rejected*, where $i \in \{1, \ldots, 22\}$

We use the Mann-Whitney U test [13] to accept or reject these hypotheses. The Mann-Whitney U test is a non-parametric statistical test that measures whether two independent distributions have equally large values. We use a 95% confidence level (*i.e.*, $\alpha = 0.05$) to accept or reject the hypotheses. Since we perform more than one comparison on the same dataset, to reduce the chances of obtaining false-positive results, we use Bonferroni correction [14] to control the familywise error rate. Concretely, we calculate the adjusted $p$-value, which is multiplied by the number of comparisons. Whenever we obtain statistically significant differences between metric values, we compute the Cliff's Delta effect size [15] to measure the magnitude of the difference. Due to the page limit, we will only report the metrics for which there is a statistically significant difference between accepted and rejected patch uplift candidates.

***Results.*** Table II summarizes differences between the characteristics of patches that were accepted for an uplift and those that were rejected. We show the median value of accepted and rejected uplifts for each metric, as well as the $p$-value of the Mann Whitney U test and the effect size. For all three channels, rejected uplifts have longer response delta ($m_7$) than accepted uplifts. We attribute this outcome to the high complexity of the rejected patches, which required longer time for risk assessment. We summarize the different results among the channels as follows:

- *Aurora*: We observe that rejected uplift requests have significantly higher landing delta; this might imply that the rejected patches are landing at the end of the Aurora cycle, and so have less time for stabilization. Also, rejected uplift requests have higher ratio of comment in the source code, although we expected that a higher comment ratio might help release managers understand the code. A high comment ratio could also indicate a high code complexity. Release managers may hesitate to release patches with complex code ahead of schedule.
- *Beta*: Compared to accepted patches, rejected patches tend to have higher code complexity in terms of LOC and number of functions, as well as higher SNA values in terms of PageRank. This result is expected, because we assume that complex code and code connected with many other classes is less likely to be accepted for urgent releases. As in the Aurora channel, rejected patches also contain code with higher ratio of comment. Although accepted and rejected patches have significant differences on some other metrics such as cyclomatic complexity, the magnitude of these differences is negligible.

**According to the results, we can only reject $H_7^{01}$, meaning that the response delta can significantly affect the decision to uplift a patch or not. The impact of other metrics, including code complexity and SNA metrics, is channel dependent.**

We quantified the acceptance rate of uplift requests for different components and observed that certain components enjoy a 100% acceptance rate (perhaps because they rarely experienced faults); while other components have lower acceptance rates (perhaps because they are inherently more complex, *e.g.*, the implementation of JavaScript, or because release managers have had bad experience with some of them). This difference between the acceptance rates of components is more pronounced in the Release channel. Some components that are involved in a large number of uplifts (*e.g.*, *Audio/Video*, *Graphics*, and *DOM* components) also have the lowest acceptance rate. Perhaps developers of those components tend to ask for uplifts more often, prompting a negative reaction from release managers who may feel that they take too many risks.

2) Qualitative Analysis. Since we did not observed significant structural differences between the code of patch uplift candidates that were rejected and those that were accepted, we conducted a qualitative study to identify and compare the reasons behind successful and failed patch uplift requests.

***Approach.*** From 2,384 uplifted issues in the Beta channel and 231 uplifted issues in the Release channel, we randomly choose respectively 459 and 154 issues as our samples (which

Table III: Uplift reasons and descriptions (abbreviations are shown in parentheses).

| **Reason** | **Description** |
|---|---|
| Security | Security vulnerability exists in the code. |
| Crash | Program unexpectedly stops running. |
| Hang | Program keeps running but without response. |
| Performance degradation (perf) | Functionalities are correct but response is slow or delayed. |
| Incorrect rendering (rendering) | Components or video cannot be correctly rendered. |
| Wrong functionality (func) | Incorrect functionalities besides rendering issues. |
| Web incompatibility (web comp) | Program does not work correctly for a major website or many websites due to incompatible APIs or libraries, or a functionality, which was removed on purpose, but is still used in the wild. |
| Add-on or plug-in incompatibility (addon comp) | Program does not work correctly for a major add-on/plug-in or many add-ons/plug-ins due to incompatible APIs or libraries, or a functionality, which was removed on purpose, but is still used in the wild. |
| Compile | Compiling errors. |
| Feature | Introduce or remove features, including support adding. |
| Improvement (improve) | Minor functional or aesthetical improvement. |
| Test-only problem (test) | Errors that only break tests. |
| Other | Other uplift reasons, *e.g.*, data corruption and license incompatibility. |

correspond to a confidence level of 95% and a confidence interval of 5%). Inspired by Tan et al.'s work [16], we classify the uplift reasons into 14 categories based on their (potential) impact and detected fault types. Some of Tan et al.'s categories are too broad, such as incorrect functionality. We break them into more detailed uplift reasons, *e.g.*, incorrect functionality is split to incorrect rendering and (other) wrong functionality. Some of Tan et al.'s categories, such as data corruption, are with too few occurrences. We combine them into the "other" category. Table III shows the uplift reasons used in our classification. We perform a card sorting on each of the sampled issues. By studying the issue report, the first and the second authors of the paper individually classified each issue into one or multiple uplift reasons (some uplift may be due to multiple reasons). Then we compared their classifications and resolved conflicts through discussions. We discussed each conflict until an agreement was reached.

To connect uplift reasons with the risk of regression, we will show the distribution of the faulty uplifts for each uplift reason.

Moreover, to identify organization factors that play a role in patch uplift decisions, we interviewed three of the current five Mozilla release managers (the other remaining two are new to the role) one at a time (to avoid them influencing each other), asking them the following questions:

1) *Which factors do you take into account when deciding about an uplift?*
2) *Are there differences in how you handle uplifts in different channels, and what are the differences?*
3) *How do you decide which developers you can trust?*

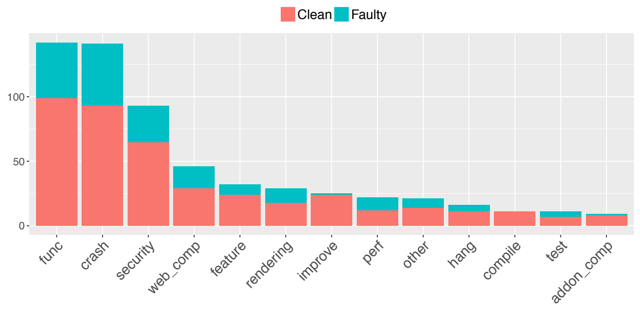

Figure 3: Distribution of uplift reasons in Beta.

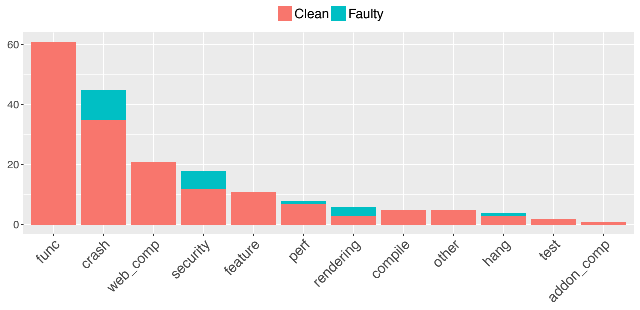

Figure 4: Distribution of uplift reasons in Release.

We also reported the results of our quantitative analysis to them and asked for their feedback.

***Results.*** Figures 3 and 4 show the distribution of the uplift reasons, as well as the distribution of fault-inducing uplifts and clean uplifts for each reason. We observe that, in the Beta channel, most patches are uplifted because of a wrong functionality, crash, security vulnerability, incompatibility with some major websites, or to introduce/remove a feature. Most regressions are introduced by the uplifts that resolved wrong functionalities, crash, and security issues. For some uplift reasons, including improvement, resolving add-on/plug-in incompatibility and compiling errors, few patches lead to faults in our studied sample. However, a high percentage of patches resolving performance and rendering problems introduced new regressions.

In the Release channel, we observe the same top five uplift reasons. Compared to the Beta channel, there are fewer regressions; implying that these uplifted patches may have been more carefully scrutinized, the rules for approval on the Release channel being more strict. The fault-inducing patches only concentrated on five uplift categories: crash, hang, security, performance degradation, and incorrect rendering. Especially, most patches for incorrect rendering lead to future faults. These results suggest that, although developers prudently uplift patches in the Release channel, they still need to carefully review patches belonging to the aforementioned categories in order to prevent delivering faults to users.

Through the interview, we learn that release managers take into account several factors when deciding whether to approve or reject a patch uplift request.

1) Importance of the issue. This is measured through the impact that rejecting the uplift would have on users.
2) Risk associated with the patch. Release managers share the same view on the risks. They generally trust developers' words, unless they have had bad experiences with them (*e.g.*, developers who caused regressions and did not fix them); they evaluate the risk of the patch by looking at its size and complexity, the presence/absence of automated tests, the reviewers of the patch. In case of doubts, release managers consult other release managers or engineering managers to get a clearer picture.
3) Timing of the uplift in the Aurora/Beta cycle. They tend to trust more patches that have been in Nightly for some time and patches that are far from the next release date. They almost always accept uplifts requested during the first weeks of the Aurora cycle.
4) Verification of the patch. In particular for more stable channels, they make sure that the patch has been verified to actually fix the problems it was supposed to fix. If needed, they ask QA to manually verify the patch. If it is a patch that fixes a Nightly crash, before uplifting the patch to Aurora, they will verify if users are no longer reporting the crash.

They remarked that the uplift bar gets higher as they are getting closer to release. After the middle point of the Beta cycle, they only accept patches fixing high security issues, high-volume crashes, severe recent regressions, severe performance issues or memory leaks.

We presented the release managers with the results of our quantitative and qualitative analysis and collected the following observations.

**They found that the response delta information is interesting**. After thinking about it, they all gave us similar replies. When they are evaluating a complex issue and are still undecided, they will not make the call immediately. One release manager said that *"when I reject something, I won't make the call immediately. I will think about it before doing it, in case I change my mind or new facts are coming in the equation"*.

**Regarding the landing delta**, they were surprised, as they thought they were more likely to accept patches with a higher landing delta (that is, patches that have been in Nightly for longer). They have also said that they are almost always accepting patches during the first four weeks of the Aurora cycle, which would explain this discrepancy (as those patches have a small landing delta).

The interviewed release managers also told us that they take into account the fault-proneness of components when making uplift decisions; which is in line with what we found (some components have a smaller acceptance rate). One release manager told us that *"some components always come out as causing the most regressions, e.g., graphics layers, DOM"*. Regarding the trust in developers, they all mentioned the assessment of risk as one of the first factors. One release manager explained that *"when they seem really overconfident or aren't telling me the whole story I lose some trust"*, another one stated that *"some developers are taking a lot of risks, some other less and are super reactive to fix potential fallout"*. This finding is consistent with the uplift criteria followed at Facebook [17], where release managers tend to trust developers who introduced less regressions in the past.

Regarding uplift reasons, release managers were not surprised that test and compile changes are less frequent than others. They argued that these kinds of changes are really hard to move from the Nightly channel to a stabilization channel (build or test failures, unless they happen on really particular configurations, are noticed as soon as a patch is applied, since tests are run for every changeset). For the same reasons, they were not surprised that the uplift regressions are rarely compile-related.

Release managers argued that the information about the distribution of uplift reasons is useful for their future decision-making. They were initially surprised to see that crash and security-related uplifts often caused regressions, but they thought that the urgency of those fixes might degrade their quality. They were also interested in the results regarding the categories where a high proportion of uplift patches caused regressions (*e.g.*, performance uplifts). They said that they will start to take this information into account when deciding about uplifts, and will be more careful with the uplifts in those categories.

*RQ2: What are the characteristics of uplifted patches that introduced faults in the system?*

***Motivation.*** In Firefox' Aurora, Beta and Release channels, we found respectively 8.8%, 8.3%, and 7.9% of uplifted patches that introduced regressions in the system. These patches not only decrease the users-perceived software quality, but also increase development costs, since developers, testers and release managers have to rework the faulty patches. In **RQ1**, we have identified some characteristics of patches that are taken into account by Mozilla release managers during patch uplifts. In this research question, we are interested in identifying the characteristics of uplifted patches that introduced faults in the system.

### 1) Quantitative Analysis.

***Approach.*** We apply the SZZ algorithm (described in Section III-B2) on all fault-fixing changes to identify uplifted patches that introduced a fault in the system. Next, we classify the uplifted patches into two groups: fault-inducing uplifts and clean uplifts. We use the 22 metrics listed in Table I to assess the differences between these two groups. For each ($m_i$) metric, we test the following hypothesis:

$H_i^{02}$: *there is no difference between the values of* $m_i$ *for uplifted patches that introduced a fault in the system and those that did not.*

Similar to **RQ1**, we use the Mann-Whitney U test and Cliff's Delta effect size to accept or reject the hypotheses, and assess the magnitude of the differences between fault-inducing uplifts and clean uplifts. We also test the hypotheses for all three channels.

Table IV: Fault-inducing Uplifts vs. Clean uplifts.

| Channel | Metric | Faulty | Clean | $p$-value | Effect size |
|---|---|---|---|---|---|
| *Aurora* | Patch size | 155.0 | 34.0 | **5.59e-65** | large |
| | Prior changes | 362.5 | 164.0 | **3.80e-10** | small |
| | LOC | 903.6 | 457.4 | **2.23e-06** | small |
| | Cyclomatic | 2.5 | 2.0 | **1.08e-06** | small |
| | # of functions | 34.3 | 17.0 | **2.25e-06** | small |
| | Max. nesting | 2.7 | 2.0 | **5.14e-04** | negligible |
| | Comment ratio | 0.2 | 0.1 | **4.00e-15** | small |
| | Module number | 2.0 | 1.0 | **2.99e-24** | small |
| | Closeness | 1.5 | 1.2 | **2.78e-13** | small |
| | Betweenness | 45,221.9 | 880.7 | **2.65e-14** | small |
| | PageRank | 1.7 | 1.4 | **1.95e-15** | small |
| | # of comments | 26.0 | 20.0 | **1.76e-09** | small |
| | Developer exp. | 28.5 | 10.0 | **1.19e-18** | small |
| | Reviewer exp. | 9.0 | 2.0 | **6.63e-09** | small |
| | Comment words | 10.0 | 2.0 | **9.08e-07** | small |
| | Developer senti. | -3 | -3 | **8.92e-04** | negligible |
| | Owner sentiment | -2 | -1 | **1.66e-04** | negligible |
| *Beta* | Patch size | 141.0 | 32.0 | **6.44e-33** | large |
| | Prior changes | 268.0 | 156.5 | **1.02e-03** | small |
| | LOC | 895.5 | 476.3 | **1.66e-03** | small |
| | Cyclomatic | 2.5 | 2.0 | **3.69e-03** | small |
| | # of functions | 37.0 | 18.0 | **3.13e-03** | small |
| | Max. nesting | 2.7 | 2.2 | **0.01** | negligible |
| | Comment ratio | 0.2 | 0.1 | **4.61e-05** | small |
| | Module number | 2.0 | 1.0 | **7.45e-12** | small |
| | Closeness | 1.6 | 1.2 | **2.87e-07** | small |
| | Betweenness | 35,661.7 | 1,327.8 | **6.00e-08** | small |
| | PageRank | 1.7 | 1.4 | **1.08e-06** | small |
| | # of comments | 28.0 | 22.0 | **1.18e-04** | small |
| | Comment words | 8.0 | 3.0 | **0.04** | negligible |
| | Developer exp. | 29.0 | 10.0 | **1.33e-08** | small |
| | Reviewer exp. | 10.0 | 2.0 | **3.35e-05** | small |
| | Owner sentiment | -2 | -1 | **4.14e-03** | small |
| *Release* | Patch size | 108.0 | 27.0 | **2.07e-03** | large |

***Results.*** Table IV summarizes differences between the characteristics of uplifted patches that introduced a fault in the system and those that did not. We observe that fault-inducing uplifts have significantly larger patch size ($m_{11}$) than clean ones, across all three channels. The effect size of the difference is large. This implies that patches with larger modifications are more likely to introduce a regression if uplifted. We observed the following on the different channels:

- On Aurora and Beta channels, fault-inducing uplifts tend to have more complex code in terms of LOC, cyclomatic complexity, number of functions, and number of modules. These patches often contain classes that are connected to many other classes, in terms of closeness, betweenness and PageRank. Fault-inducing uplifts also tend to have higher comment ratios and tend to change files that were changed more frequently. Interestingly, fault-inducing uplifts are frequently submitted by developers or reviewers with high experience. Fault-inducing uplifts also have a larger amount of comments than clean uplifts. A large number of comments may be a sign that developers are struggling with the patch, which may explain the high fault-proneness. Although fault-inducing uplifts and clean uplifts also display other significant differences (as shown in Table IV), the magnitude of these differences is negligible.
- For the Release channel, we do not observe a significant difference between fault-inducing uplifts and clean uplifts for the above metrics.

**Overall, we reject $H_{11}^{02}$, *i.e.*, fault-inducing uplifts have larger patch size than clean uplifts. Release managers should pay attention to large patches and reviewers should scrutinize them carefully. Although the effect of other characteristics is channel dependent, in Aurora and Beta, we observe that patches with high complexity and centrality tend to lead to faults. Uplift requests submitted by experienced developers and reviewers also tend to lead to regressions.**

Similar to **RQ1**, we examined patch uplifts per component, and observed that patch uplifts affecting certain components (*e.g.*, *Graphics* component) are more likely to cause regressions than others. Some of the components with the highest fault-inducing rates also have a low approval rate; probably because the release managers were acting based on their previous experiences with those components (for example, the *Web Audio* component). Components like the *Audio/Video*, which are involved in multiple patch uplift operations, also have the highest fault-inducing rates; these components would be inherently more prone to faults because of their complexity, or technical debt.

We made a similar observation regarding developers' submitting uplift requests. Many developers who submitted multiple uplift requests appear in the list of developers with high fault-inducing rates; perhaps, by uplifting more patches, they are taking more risks.

### 2) Qualitative Analysis.

To understand the root cause of faults in uplifted patches, we conduct a qualitative study.

***Approach.*** We manually examined uplifted patches (from the samples selected in **RQ1**) that introduced faults, and classified the reasons behind the faults. Inspired by the work of Tan et al [16], we defined seven possible root causes for uplift faults (as shown in Table V). We identified respectively 132 and 17 fault-inducing uplifts from the Beta and Release samples chosen in **RQ1**, and performed a card sorting to classify each of the faults into one or multiple causes. As in **RQ1**, the first and the second authors individually read the issue reports and their fault-fixing patches to understand the root causes of the faults (*i.e.*, the reason why their corresponding uplifted patches caused the faults) and classified these root causes along our seven categories. Similar to **RQ1**, disagreements were resolved through discussions.

We also interviewed release managers, asking them the following question: *What are the characteristics of fault-*

Table V: Fault reasons and descriptions.

| Reason | Description |
|---|---|
| Memory | Memory errors, including memory leak, overflow, null pointer dereference, dangling pointer, double free, uninitialized memory read, and incorrect memory allocation. |
| Semantic | Semantic errors, including incorrect control flow, missing functionality, missing cases of a functionality, missing feature, incorrect exception handling, and incorrect processing of equations and expressions. |
| Third-party | Errors due to incompatibility of drivers, plug-ins or add-ons. |
| Concurrency | Synchronization problems between multiple threads or processes, *e.g.*, incorrect mutex usage. |
| Compile | Compile-time errors. |
| Other | Other errors. |

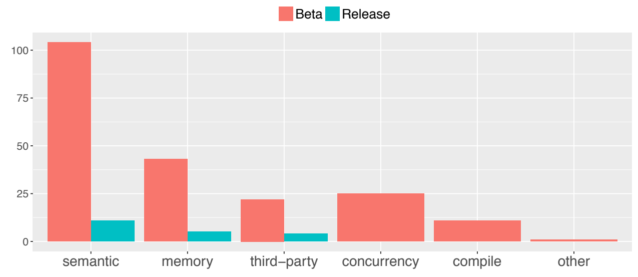

Figure 5: Reasons of fault-inducing uplifts.

*inducing patches that you are not currently taking enough into account but could be considered in the future?*

***Results.*** Figure 5 depicts the distribution of the reasons why fault-inducing uplift introduced regressions. In both channels, semantic and memory-related errors are dominant root causes of the uplift regressions. With a detailed check on the patches, we find that many memory errors are due to null pointer dereference and memory leak. In addition, incompatibility of plug-ins and drivers also cause uplift regressions in both channels. Concurrency issues are ranked as a popular cause for Beta's uplift regressions, but we do not find any example of this category in the Release channel. In general, our results suggest that, when uplifting a patch, **release managers need to carefully check for potential faults on the program's semantic meaning, memory operations, synchronization, and third-party extension's compatibility**.

In the interview, **all the release managers agreed that it would be beneficial for them to have more detailed information about the complexity of the patches they are asked to evaluate and more information about the history of the components involved in these patches**. This resonates with our findings. Release managers were surprised to see that fault-inducing patches were more likely to be written by more experienced developers and reviewed by more experienced reviewers. They guess that these developers/reviewers are assigned to more complex tasks with more complex solutions. A release manager told us that *"if you call in the big guns, then it's a warning sign"*.

The fault categorization was also interesting for the release managers, who told us that Mozilla is about to employ more static analysis tools (*e.g.*, Coverity [18]) and to move some of their code from C++ to a safer language (*e.g.*, Rust). It is promising for them to see how many memory and concurrency faults can be avoided by using these techniques, and how many semantic and third-party faults can be reduced by enhancing code review or testing efforts.

## V. Discussion

According to the results of **RQ1** and **RQ2**, there are statistically significant differences between the characteristics of uplifted patches that introduced regressions and those that were integrated successfully (*i.e.*, clean uplifts that did not induce faults). Also, fault-inducing uplifts are in the majority of cases uplifts that were meant to resolve wrong functionalities, crashes, security vulnerabilities, and incompatibilities with websites. Furthermore, incorrect semantic code and memory operations are the most important root causes of uplift regressions.

We believe that release management teams could leverage these findings to build classifiers capable of automatically assessing the risk associated with patch uplift candidates and recommend patches that can be uplifted safely.

Exploring the possibility of building such classifiers is part of our future work agenda.

## VI. Threats to Validity

*Construct validity* threats are concerned with the relationship between theory and observation. Previous studies [19], [8] suggested that complex code is a good indicator of fault-proneness. We confirm this point in this study. However, we found that fault-inducing patches are more likely to be submitted by experienced developers, which contradicted our expectations. We attribute this outcome to the fact that experienced developers are often assigned to difficult issues, whose resolution tend to be more complex. Also, release managers might overlook risks associated to patches submitted by experienced developers, as these developers are often more trusted than others.

*Internal validity* threats concern factors that may affect a dependent variable and were not considered in the study. We paid attention not to violate the assumptions of the statistical tests that are performed in the paper. Specifically, in **RQ1** and **RQ2**, we applied non-parametric tests that do not require making assumptions on the distribution of our dataset.

*Conclusion validity* threats concern the relationship between the treatments and the outcome. Before conducting the case study, we limited our studied dataset within a duration that covers consecutive series of relatively stable periods on all the three uplift channels. In addition, we used a keyword matching heuristic to identify fault-related issues. We manually validated a random sample of 380 issues. All the authors of this paper participated in the validation. Whenever there were diverging opinions, we set up a meeting and discussed the issue until a consensus was reached. As a result, we found that our heuristic can achieve a precision of 87.3% and a recall of 78.2%, when identifying fault-related issues. Moreover, we performed a manual classification of the uplift reasons and the root causes of uplift regressions. To mitigate potential bias that may result from our subjective opinions, we also discussed on each of our classification conflicts until reaching a consensus. However, as any other taxonomic study, we cannot guarantee a 100% of accuracy on our classification results. Future replications are welcomed to validate our work. Another issue on the manual classification is that, although we randomly chose our samples by applying a confidence level of 95% and a confidence interval of 5%, our samples might not precisely reflect the distributions of the uplift reasons and–or root causes of uplift regressions on the whole Firefox dataset. Further investigations on larger data sets are desirable.

*External validity* threats are concerned with the generalizability of our results. In this paper, we only studied Mozilla Firefox. First, Mozilla Firefox is the most studied system for issues related to rapid releases; moreover, the system's data are publicly available. We also have the opportunity to perform both quantitative and qualitative analyses (including the interviews with release managers) on this system. However, we should recognize that our findings may not be generalizable to other systems. In the future, we plan to collaborate with other software organizations, to validate and extend the results of this work. In addition, more studies on other systems with other programming languages are suitable to further validate our results. To facilitate future replication studies, we share our datasets and scripts at: https://github.com/swatlab/uplift-analysis.

## VII. Related Work

Patch uplift is an activity performed during the release engineering process. Hence, in this section, we present and discuss relevant literature on release engineering.

Release engineering encompasses all the activities aimed at "building a pipeline that transforms source code into an integrated, compiled, packaged, tested, and signed product that is ready for release" [20].

Since the adoption of the rapid release model [2] by Mozilla in 2011, a plethora of studies have focused on the impact of rapid release strategies on software quality. Khomh et al. [2] compared crash rates, median uptime, and the proportion of post-release bugs between the versions of Firefox that followed a traditional release cycle and those that followed a rapid release cycle. They observed that short release cycles do not induce significantly more bugs. However, compared to traditional releases, users experience bugs earlier during software execution. Nevertheless, they also observed that post-release bugs are fixed faster under the rapid release model. Da Costa et al. [21] studied the impact of Mozilla's rapid release cycles on the integration delay of addressed issues. They found that, compared to the traditional release model, the rapid release model does not deliver addressed issues to end users more quickly, which is contrary to expectations.

Another important aspect of release engineering that has been investigated by the community is the integration of urgent patches that are used to fix severe problems, such as frequent crashes or security bugs, or to introduce important features. Urgent patches break the balance between new feature work and software quality, and hence could lead to faults and failures. Hassan et al. [22] investigated emergency updates for top Android apps and identified eight patterns along the following two categories: "updates due to deployment issues" and "updates due to source code changes". They suggested to limit the number of emergency updates that fall in these patterns, since they are likely to have a negative impact on users' satisfaction. In a recent work, Lin et al. [23] empirically analyzed urgent updates in 50 most popular games on the Steam platform, and observed that the choice of the release strategy affects the proportion of urgent updates, *i.e.*, games that followed a rapid release model had a higher proportion of urgent patches in comparison to those that followed the traditional release model. Rahman et al. [24] examined the "rush to release" period on Linux and Chrome. They observed that experienced developers are often allowed to make changes right before stabilization occurs and these changes are added directly to the stabilization line. They also found that there is a rush in the number of commits right before a new release is added to the stabilization channel, to add final features. In a following work, Rahman et al. [25] observed that feature toggles [26] can effectively turned off faulty urgent patches, which limits the impact of faulty patches.

To the best of the authors' knowledge, none of these prior works has empirically investigated how urgent patches in the rapid release model affect software quality in terms of fault-proneness, and how the reliability of the integration of urgent updates could be improved. This paper fills this gap in the literature by investigating the reliability of the Mozilla's uplift process, since uplifted patches are urgent updates.

## VIII. Conclusion

Mozilla follows a rapid release model, which uses 18 weeks to deliver fault fixes and new features to users. Frequently, certain patches that fix critical issues, or implement high-value features are promoted directly from the development channel to a stabilization channel, because they are too urgent and cannot wait for the next release train. This practice, known as *patch uplift*, is risky because the time allowed for the stabilization of the uplifted patches is short. In average, 8% of uplifted patches introduced a regression in the code of Firefox. In this paper, we investigated the decision making process of patch uplift at Mozilla and observed that release managers are more inclined to accept patch uplift requests that concern certain specific components, and–or that are submitted by certain specific developers. We examined the characteristics of uplifted patches that introduced regressions in the code and found that they are more complex than clean uplifts, and they tend to change a higher number of lines of code. Most regressions are caused by patch uplifts aimed at fixing wrong functionalities and crashes. The most common root causes of faults in uplifted patches are semantic and memory errors. Reviewers and release managers should carefully inspect complex patches before allowing their uplift.

## References

[1] "A. Laforge. Chrome release cycle. Job title: Technical Program Manager (Chrome) at Google," http://www.slideshare.net/Jolicloud/chrome-release-cycle, 2016, online; Accessed 06 February 2016.

[2] F. Khomh, T. Dhaliwal, Y. Zou, and B. Adams, "Do faster releases improve software quality? An empirical case study of Mozilla Firefox," in *Proceedings of the 9th IEEE Working Conference on Mining Software Repositories (MSR)*. IEEE, 2012, pp. 179–188.

[3] "Mozilla Tree Sheriffs," https://wiki.mozilla.org/Sheriffing, 2017, online; Accessed February 1st, 2017.

[4] "JIRA," https://jira.atlassian.com/, 2017, accessed March 30th, 2017.

[5] J. Śliwerski, T. Zimmermann, and A. Zeller, "When do changes induce fixes?" in *ACM sigsoft software engineering notes*, vol. 30, no. 4. ACM, 2005, pp. 1–5.

[6] M. Fischer, M. Pinzger, and H. Gall, "Populating a release history database from version control and bug tracking systems," in *Proceedings of the 29th International Conference on Software Maintenance (ICSM)*. IEEE, 2003, pp. 23–32.

[7] T. Mike, B. Kevan, P. Georgios, C. Di, and K. Arvid, "Sentiment in short strength detection informal text," *JASIST*, vol. 61, no. 12, pp. 2544–2558, 2010.

[8] D. Kim, X. Wang, S. Kim, A. Zeller, S.-C. Cheung, and S. Park, "Which crashes should I fix first?: Predicting top crashes at an early stage to prioritize debugging efforts," *IEEE Transactions on Software Engineering*, vol. 37, no. 3, pp. 430–447, 2011.

[9] "Understand tool," https://scitools.com, 2016, online; Accessed March 31st, 2016.

[10] G. Csardi and T. Nepusz, "The igraph software package for complex network research," *InterJournal, Complex Systems*, vol. 1695, no. 5, pp. 1–9, 2006.

[11] L. An and F. Khomh, "An empirical study of highly-impactful bugs in Mozilla projects," in *Proceedings of the IEEE International Conference on Software Quality, Reliability and Security (QRS)*. IEEE, 2015.

[12] "Mozilla uplift rules," https://wiki.mozilla.org/Release_Management/Uplift_rules, 2016, online; Accessed February 5th, 2017.

[13] M. Hollander, D. A. Wolfe, and E. Chicken, *Nonparametric statistical methods*, 3rd ed. John Wiley & Sons, 2013.

[14] A. Dmitrienko, G. Molenberghs, C. Chuang-Stein, and W. Offen, *Analysis of Clinical Trials Using SAS: A Practical Guide*. SAS Institute, 2005. [Online]. Available: http://www.google.ca/books?id=G5ElnZDDm8gC

[15] N. Cliff, "Dominance statistics: Ordinal analyses to answer ordinal questions." *Psychological Bulletin*, vol. 114, no. 3, p. 494, 1993.

[16] L. Tan, C. Liu, Z. Li, X. Wang, Y. Zhou, and C. Zhai, "Bug characteristics in open source software," *Empirical Software Engineering*, vol. 19, no. 6, pp. 1665–1705, 2014.

[17] "Keynote of the 2014 Release Engineering conference," https://www.youtube.com/watch?v=Nffzkkdq7GM, 2014, online; Accessed March 30th, 2017.

[18] "Coverity tool," http://www.coverity.com, 2017, online; Accessed March 31st, 2017.

[19] Y. Kamei, E. Shihab, B. Adams, A. E. Hassan, A. Mockus, A. Sinha, and N. Ubayashi, "A large-scale empirical study of just-in-time quality assurance," *IEEE Transactions on Software Engineering*, vol. 39, no. 6, pp. 757–773, 2013.

[20] B. Adams, S. Bellomo, C. Bird, T. Marshall-Keim, F. Khomh, and K. Moir, "The practice and future of release engineering: A roundtable with three release engineers," *IEEE Software*, vol. 32, no. 2, pp. 42–49, 2015.

[21] D. A. da Costa, S. McIntosh, U. Kulesza, and A. E. Hassan, "The Impact of Switching to a Rapid Release Cycle on Integration Delay of Addressed Issues: An Empirical Study of the Mozilla Firefox Project," in *Proceedings of the 13th International Conference on Mining Software Repositories (MSR)*, 2016, pp. 374–385.

[22] S. Hassan, W. Shang, and A. E. Hassan, "An empirical study of emergency updates for top android mobile apps," *Empirical Software Engineering*, pp. 1–42, 2016.

[23] D. Lin, C.-P. Bezemer, and A. E. Hassan, "Studying the urgent updates of popular games on the steam platform," *Empirical Software Engineering*, pp. 1–32, 2016.

[24] M. T. Rahman and P. C. Rigby, "Release stabilization on linux and chrome," *IEEE Software*, vol. 32, no. 2, pp. 81–88, 2015.

[25] M. T. Rahman, L.-P. Querel, P. C. Rigby, and B. Adams, "Feature toggles: practitioner practices and a case study," in *Proceedings of the 13th International Conference on Mining Software Repositories*. ACM, 2016, pp. 201–211.

[26] "Feature toggle," https://martinfowler.com/bliki/FeatureToggle.html, 2017, online; Accessed March 22nd, 2017.