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

An Empirical Study of Patch Uplift in Rapid Release Development Pipelines

Marco Castelluccio · Le An · Foutse Khomh

Received: date / Accepted: date

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 the uplifted patches that did not effectively fix the targeted problem and that introduced regressions. Through statistical and manual analyses, a series of problems were investigated, including the reasons behind patch uplift decisions, the root causes of ineffective uplifts, the characteristics of uplifted patches that introduced regressions, and whether these regressions can be prevented. Additionally, three Mozilla release managers were interviewed in order 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. Certain uplifts did not effectively address their problems because they did not completely fix the problems or lead to regressions. Uplifted patches that lead to regressions 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

Marco Castelluccio
Mozilla Corporation, United Kingdom
University of Napoli Federico II, Italy
E-mail: mcastelluccio@mozilla.com

Le An
Polytechnique Montreal, Canada
E-mail: le.an@polymtl.ca

Foutse Khomh
Polytechnique Montreal, Canada
E-mail: foutse.khomh@polymtl.ca

components, and–or that are submitted by certain specific developers. About 25% to 30% of the regressions due to Beta or Release uplifts could have been prevented as they could be reproduced by developers and were found in widely used feature/website/configuration or via telemetry.

Keywords Patch uplift · Urgent update · Mining software repositories · Release engineering

1 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 re-organized 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 [35], 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 observed 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: How effective are uplift operations?

4% of the subject uplifts did not effectively address the problems but were later reopened, duplicate or cloned into another issue, or required additional uplifts to fix the issue. Two major root causes were observed from the ineffective uplifts: the uplifts only partially fixed the issues or caused regressions. Higher proportion of ineffective uplifts were detected from the Release channel than from Aurora and Beta.

RQ3: What are the characteristics of uplifted patches that introduced faults in the system?

From our analysis, we observed 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.

RQ4: Are regressions caused by uplift more severe than the bugs that were fixed with the uplift?

Through a manual analysis, we observed that 37.5% of the Beta fault-inducing uplifts caused a "more severe regression", *i.e.*, regression that is more severe than the problems they aimed to address. No "more severe regression" was found from the examined Release uplifts, perhaps due to a more strict uplift policy and code review process on this channel.

RQ5: Could some of the regressions have been prevented through more extensive testing on the channels?

We considered regressions to be possibly preventable if they were reproducible not only by the issue reporter and were found either on a widely used feature/website/configuration or via Mozilla's telemetry. We manually examined a sample of regressions due to Beta and Release uplifts, and found that 25% of the regressions due to Beta uplifts and 30% of the regressions due to Release uplifts could have been possibly prevented.

This paper is an extension of an earlier conference paper [5]. Our original work:

1. Quantitatively and qualitatively analyzed the characteristics of the uplifted patches and identify the reasons why these patches were release ahead of time;
2. Quantitatively and qualitatively analyzed the characteristics the uplifted patches that led to regressions and identify the root causes of these failed uplifts.

In this extension work, we expand our analysis in three ways:

1. We identified and analyzed the reoccurrences of the already uplifted patches in forms of reopened, cloned, duplicate issues and issues that were fixed by multiple uplifts;
2. We compared the severity of the regression an uplifted patch led to against the problem the patch intended to address;
3. We identified the failed uplifts that could have been possibly prevented through more extensive testing.

The remainder of this paper is organized as follows. Section 2 provides background information about patch uplift. Section 3 describes the design of our case study. Section 4 presents the results of the case study. Section 5 discusses threats to the validity of this study. Section 6 summarizes related works, and Section 7 concludes the paper.

2 Mozilla Patch Uplift Process

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

Firefox follows a pipelined release process [16], 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 [26] (*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.

3 Case Study Design

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

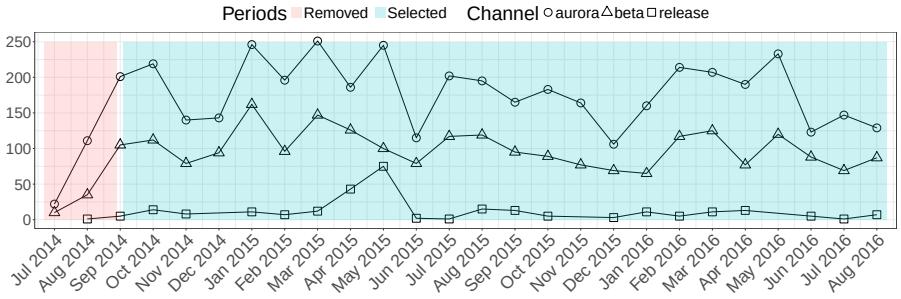


Fig. 1 Number of uplifts during each month from July 2014 to August 2016. Periods with low number of uplifts or not covering all the three channels are removed.

3.1 Data Collection

We collected, 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 relied 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 considered uplift characteristics at the issue level. If an issue contains multiple patches, we bundled 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 did 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 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.

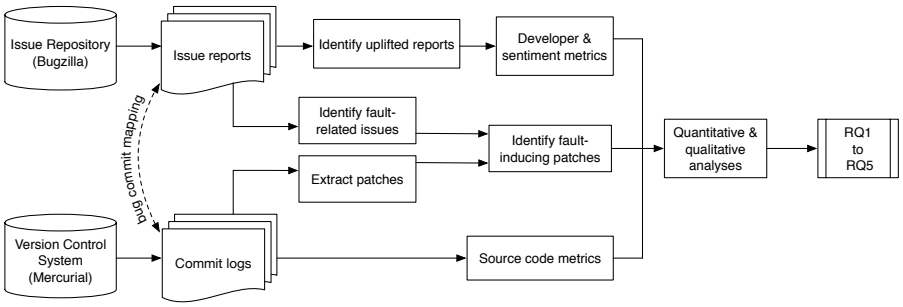


Fig. 2 Overview of our data processing approach.

3.2 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`.

3.2.1 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 [43], 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 used 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* (the true class) and *other* (the false class) categories.

3.2.2 Identification of Fault-inducing Patches

We used the SZZ algorithm [36] 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 [11] 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 used 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 considered modified and deleted lines, since added lines could not have been changed by prior commits. We denoted an issue's fault-fixing file by F_{fix}. Then, for each changed file $f_{fix} \mid f_{fix} \in F_{fix}$, we used 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 examined whether a fault-inducing candidate was submitted before the creation date of its corresponding fault-related issue report. If so, we considered the candidate to be a *fault-inducing commit*, and its related issue to be a *fault-inducing issue*.

3.2.3 Identification of Duplicate Issues

There has not been an approach that can identify duplicate issues[1] with 100% accuracy. Two general threads of approaches were proposed in previous works. The first thread of approaches ranks the similarity between one given issue and other issues in a dataset, such as [32, 37, 44]. The other thread predicts whether two given issue reports are duplicate or not, such as [15, 38, 41]. Inspired by these works, we designed the following approach, which is customized for our dataset.

1. For each subject issue report, we extracted its short description (*i.e.*, title) and long description (*i.e.*, first comment). We performed stemming and stop word removal against these raw texts.
2. As [38, 41], we used *Okapi BM25* algorithm [45] (referred as BM25 in the rest of the paper) to rank of the similarity between any pair of issues: $\{(issue_i, issue_j) \mid i \neq j, issue_i \in uplift\ bugs, issue_j \in all\ bugs\}$. In a given pair of $(issue_i, issue_j)$, we respectively calculated the similarity between their titles and their descriptions. As there are in total 33,664 studied issues and 4,958 unique uplifted issues[2], we should perform

[1] In this paper, "duplicate" issues indicate different issues that aim to address the same problem, rather than DUPLICATE in the Bugzilla sense, which means identical issues.

[2] There are in total 7,267 studied uplift requests, but some requests are across multiple channels.

$(33664 - 4958) \times 4958 + 4958 \times (4958 - 1) \approx 167M$ comparisons (for titles and descriptions respectively). In each of these comparisons, the BM25 algorithm yields a score of similarity, the higher the score the closer the two pieces of information (*i.e.*, titles or descriptions).

3. We ranked the BM25 scores for all pairs of issues by descending order. We removed the pairs where the BM25 scores is 0. The rest of the results were considered as duplicate issue candidates. We intended to manually examine the correctness of each title (respectively description) pair by carefully analyzing the whole issue reports. There are 8.1 million pairs of duplicate issues candidates, our manual validation cannot cover all these but can only focus on the most likely candidates. First, we narrowed down our manual analysis scope to the top 1,000 candidates because correct duplicate cases can hardly be observed beyond the top 1,000 candidates (in which the highest BM25 value is 97.5, and the lowest value is 29.1) through a preliminary analysis. Second, we designed a heuristic to further filter out the pairs in which the two issue reports are not linked to each other: if Issue A is never mentioned in Issue B (either in one of the comments, or in the "Blocking", "Depends On", "See Also" fields), we considered the two issues to be "not linked" (meaning that, in practice, developers did not notice any relationship between the issues). To calculate the false positive rate of this heuristic, we manually examined the top 50 and 100 other randomly selected candidates, and found that only two correct duplicate pairs were misclassified by the "unlinked" heuristic. As a result, 137 candidates survived this step. Our manual validation was then performed on these candidates.
4. Since we separately performed Steps 2 and 3 on the issue titles and descriptions, we combined the results and removed redundancies. We also removed the pairs where an issue is a clone of another one. From the obtained results, we only keep the duplicate pairs where the duplicate issue were opened or resolved after the original patch had been uplifted.

Compared to any fully automated approach, our approach can achieve a very high precision because all of the reported duplicate issue pairs have been carefully examined by two authors of the paper (by understanding the whole context of the issue reports). Although we cannot guarantee a 100% recall, we believe that our reported results covers all possible cases where the titles (respectively descriptions) of a pair of issues are textually similar to each other. In fact, text processing is the base of most aforementioned approaches. BM25 is considered as an advanced measure of ranking similarities, which has a higher performance than the traditional TF-IDF algorithm [41]. Some approaches, such as [38, 41], used additional information (*e.g.*, priority, product, and version fields from the analyzed issue reports), but such information cannot help to retrieve more possible candidate (*i.e.*, it cannot increase the recall). In this work, we only ignored the issue pairs where the titles or descriptions have no relevance (*i.e.*, BM25 value is 0) or have little relevance (*i.e.*, the two issues are not linked and the BM25 value is weak).

Table 1 Developer experience and participation metrics (m_1 - m_5).

Metric	m_i	Description	Type and range
Developer experience	1	Number of previous commits of the patch developer.	Integer, from 0 to 43639.
Reviewer experience	2	Number of previous commits of the patch reviewer.	Integer, from 0 to 43691.
Number of comments	3	Number of comments in the issue report.	Integer, from 3 to 1359.
Comment words	4	Average number of words in the comments to an issue.	Integer, from 0 to 2199.
Review duration	5	Time period (in days) from a patch's submission until its approval.	Float, from 0.0 to around 406.67.

Table 2 Uplift process metrics (m_6 - m_8).

Metric	m_i	Description	Type and range
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. The value can be negative, as sometimes developers request uplift before their patch is applied to Nightly.	Float, from -41.59 to around 153.73.
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).	Float, from 0.0 to around 31.23.
Release delta	8	Time elapsed (in days) between when the developer asked for approval for the uplift and the date of the following release.	Float, from 0.0 to around 42.76.

Table 3 Sentiment metrics (m_9 - m_{10}).

Metric	m_i	Description	Type and range
Developer sentiment	9	The highest negative sentiment score in the developers' comments on an issue.	Integer, from -5 to 0.
Owner sentiment	10	The highest negative sentiment score in module owners' comments on an issue.	Integer, from -5 to 0.

3.2.4 Mining Issue Reports

We mined 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.

3.2.5 Computing Metrics

To capture the characteristics of patches that were uplifted, we computed the 22 metrics described in Tables 1 to 5. 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

Table 4 Code complexity metrics (m_{11} - m_{19}).

Metric	m_i	Description	Type and range
Patch size	11	Number of lines in a patch (excluding test patches).	Integer, from 0 to 301114.
Test patch size	12	Number of lines in a test patch.	Integer, from 0 to 127155.
Prior changed times	13	Number of previous commits that modified the same files that the patch is modifying.	Integer, from 0 to 114051.
LOC	14	Average lines of code in all files in a patch.	Integer, from 0 to 27727.
Average cyclomatic	15	Average cyclomatic complexity of the functions in a file.	Integer, from 0 to 128.
Number of functions	16	Average number of files' functions in a patch.	Integer, from 0 to 3878.
Maximum nesting	17	Average maximum level of nested functions in all files in a patch.	Integer, from 0 to 13.
Comment ratio	18	Average ratio of the lines of comments over the total lines of code in all files in a patch.	Integer, from 0 to 99.
Module number	19	Number of modules (as defined by Mozilla in [24]) involved by a patch.	Integer, from 0 to 76.

Table 5 Code centrality (SNA) metrics (m_{20} - m_{22}).

Metric	m_i	Description	Type and range
PageRank	20	Time fraction spent to "visit" a node (*i.e.*, file) in a random walk in the call graph.	Float, from 0.0 to 1158.91.
Betweenness	21	Number of classes passing through a node among all shortest paths.	Float, from 0.0 to 6.2e+07.
Closeness	22	The average length of the shortest path between a node and all other nodes.	Float, from 0.0 to 3.21.

may have a higher 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 computed 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 computed 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* [22], 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 [22, 42]. To adapt this tool to the software engineering context, we ignored a group of words that have negative meanings in general but do not represent any negative sentiment in Bugzilla discussions, *e.g.*, *bug*, *error*, *issue*, *regression*, *failure*, *fail*, *leak*, *crash*[3]. To further filter out irrelevant information from the comments, we used regular expressions to ignore hyperlinks and referred texts (*i.e.*, lines starting with "`>`"). In addition to developers' sentiments, we also computed module owners' sentiments.

Code Complexity Previous works, such as [18], have shown that complex code is likely to introduce faults. We calculated code complexity metrics to understand how uplifting decisions and their success are affected by the complexity of the uplifted patches. We extracted the files changed in each patch and use the static code analysis tool *Understand* [33] 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. [18] observed that functions close to the centre of a call graph are likely to experience more faults. Hence, we computed metrics capturing the centrality of functions involved in uplifted patches and uplifted patch candidates. We used the network analysis tool, *igraph* [8], in combination to *Understand* [33], as in [3], to compute the following *Social Network Analysis* (SNA) metrics: PageRank, betweenness, and closeness. When computing complexity and SNA metrics, we only considered 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 computed metrics by releases and map a given patch to its latest major release as in our previous work [3]. To make the metric results as precise as possible, we considered all major releases from Firefox 32.0 until Firefox 48.0, which cover the system's history from September 2014 until August 2016.

4 Case Study Results

This section presents and discusses the results of our five 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.

[3] Please refer to our data repository to see the whole list of ignored words: `https://github.com/swatlab/uplift-analysis`

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 among different channels. Although Mozilla has published rules to guide the patch uplift process [25], 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 Tables 1 to 5, we statistically compared 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 3.1), we calculated 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 formulated 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 used 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 used a 95% confidence level (*i.e.*, $\alpha = 0.05$) to accept or reject the hypotheses. Since we performed more than one comparison on the same dataset, to reduce the chances of obtaining false-positive results, we used Bonferroni correction [10] to control the familywise error rate. Concretely, we calculated the adjusted p-value, which is multiplied by the number of comparisons. Whenever we obtained statistically significant differences between metric values, we computed the Cliff's Delta effect size [6] to measure the magnitude of the difference. Given a result of the Cliff's Delta, d, we use the following thresholds to decide its magnitude: $|d| < 0.147$ "negligible", $|d| < 0.33$ "small", $|d| < 0.474$ "medium", otherwise "large" [31]. In the following, we report only the metrics for which there is a statistically significant difference between accepted and rejected patch uplift candidates.

Results. Table 6 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 observed 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,

Table 6 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

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 observe significant structural differences between the code

Table 7 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.

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 chose respectively 459 and 154 issues as our samples (which correspond to a confidence level of 95% and a confidence interval of 5%). Inspired by Tan et al.'s work [39], we classified 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 broke 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 combined them into the "other" category. Table 7 shows the uplift reasons used in our classification. We performed 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 organizational factors that play a role in patch uplift decisions, we interviewed three of the current five Mozilla release managers (the other remaining two were new to the role) one at a time (to avoid them influencing each other), asking them the following questions:

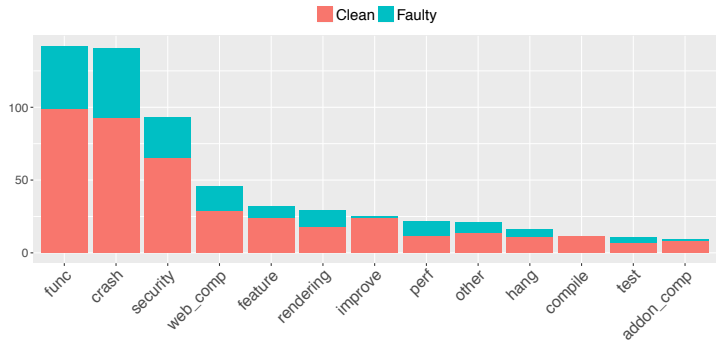


Fig. 3 Distribution of uplift reasons in Beta.

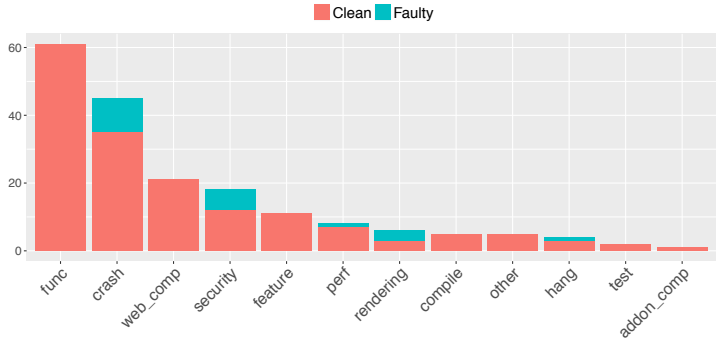


Fig. 4 Distribution of uplift reasons in Release.

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?*

After this first more structured interview with the questions above, we performed a semi-structured one, showing the results of our quantitative analysis to the release managers and asking them for their feedback.

The questions of the both interviews were open-ended, so we had to perform an analysis to extrapolate interesting elements and to group together similar ones (*e.g.*, if an interviewee mentioned "a really important issue reported multiple times" as being one of the factors and another mentioned "a bug affecting many users", we considered these factors to be the same and grouped them together in "Importance of the issue").

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 observed 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 intro-

duced 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 observed 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 [47], 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: How effective are uplift operations?

Motivation. Previous studies showed that some issues cannot be effectively fixed by one patch, but need additional fixing efforts. These issues can be detected by seeking **reopened** [21], **cloned** [40], **duplicate**, or **resolved by multiple patches** [28] (which also includes backouts made by tree sheriffs, [27]) issues. In this research question, we want to examine whether it happens that patch uplift operations require multiple attempts (we refer to such uplifts

Table 8 Root causes of the ineffective uplifts.

Category	Description
Not fixed	The issue was completely not fixed, *i.e.*, the uplifted patch did not have any effect.
Partially fixed	The issue was only partially fixed, *i.e.*, the uplifted patch had an effect but did not completely resolve the problem.
Need more QA	The uplifted patch had not gone through enough manual verification.
Need more tests	There were no tests added with the uplifted patch, but they were required.
Diagnostics	An uplift was made to gather more data on a problem, then another uplift was made to actually fix it.
Regressions	The uplifted patch caused other defects.
Test failure	The uplifted patch did not pass a certain test.
Build failure	The uplifted patch caused a build error.
Other	Other reasons, *e.g.*, an issue was fixed by an uplift, but then appeared again because of another patch; or the patch depended on other patches to be uplifted first.

as "ineffective uplifts"). Since such outcome is not desirable, it would be useful to help developers identify the characteristics of such patch uplifts, so that they can take the necessary steps to avoid reoccurrences of issues addressed by uplift operations.

Approach. To identify issues that were reopened, we used the `REOPENED` Bugzilla resolution type. To identify issues that were cloned, we used a regular expression to match the following pattern, which Bugzilla adds automatically when a user clones a bug.

```
+++ This bug was initially created as a clone of Bug #ISSUE_ID +++
```

To identify issues that were fixed by more than one uplift, we used regular expressions to detect uplifts in issue reports (see Section 3.1), and initially marked issues where at least two uplifts occurred (at a distance of at least three days between them). We chose three days because the distance between two beta builds is three days. A shorter time would likely have caught simple follow-up fixes that we are not interested in. A longer time would likely have missed some cases of multiple uplifts.

From the obtained results, we removed the issues that were reopened or cloned before their corresponding patches had been uplifted. We also removed the issues with multiple uplifted patches, which were actually uplifted together (or at the same time) or where one of the multiple uplifts was a simple test-only fix (identified by `a=test-only` in the commit message). From the user side, these issues were resolved by only one shot.

To identify issues duplicate of a previous issue fixed by patch uplift, we used the approach described in Section 3.2.3.

For each identified and verified issue that was not effectively fixed by an uplift, two of the authors independently card sorted the root causes of the ineffective uplift into one or multiple categories. They first defined categories separately, and then merged similar categories into one. Next, they standardized the category names as shown in Table 8. Finally, they used these standardized

Table 9 Number of ineffective uplifts in the three channels.

	Aurora	Beta	Release	Unique count
Reopened	70	49	10	77
Cloned	28	16	3	32
Duplicate created after an uplift	15	10	2	16
Duplicate resolved after an uplift	5	3	2	7
Resolved by multiple uplifts	50	42	3	78

categories to compare their classification differences and resolve conflicts until reaching an agreement for each of the issues.

Results. Table 9 shows the number of ineffective uplifts detected from the three development channels. Since some patches were uplifted into multiple channels, the table also shows the unique number of the ineffectively uplifted patches in a specific manner (*e.g.*, reopened, cloned, or duplicate). Figure 5 depicts the root causes of the ineffective uplifts and shows the prevalence of each root cause. In this figure, if the patch of an issue was uplifted to multiple channels, we only counted it once. **In general, 196 out of the 4,958 (4%) studied issues were not effectively fixed by one patch uplift and required additional efforts.** In previous studies, Park et al. [28] and An et al. [4] respectively detected 32.8% and 23.8% general Mozilla issues (in different time periods) that were resolved by multiple patches. Shihab et al. [34] detected 6.5% to 26% reopened issues from Eclipse, Apache HTTP, and OpenOffice. Compared to these results, uplifted patches are more likely to fix a problem in one shot than other patches, even though we analyzed ineffective uplifted patches from different angles, including reopened, cloned, duplicate issues, and issues fixed by multiple uplifts. This implies that uplifted patches have a better general quality than other patches.

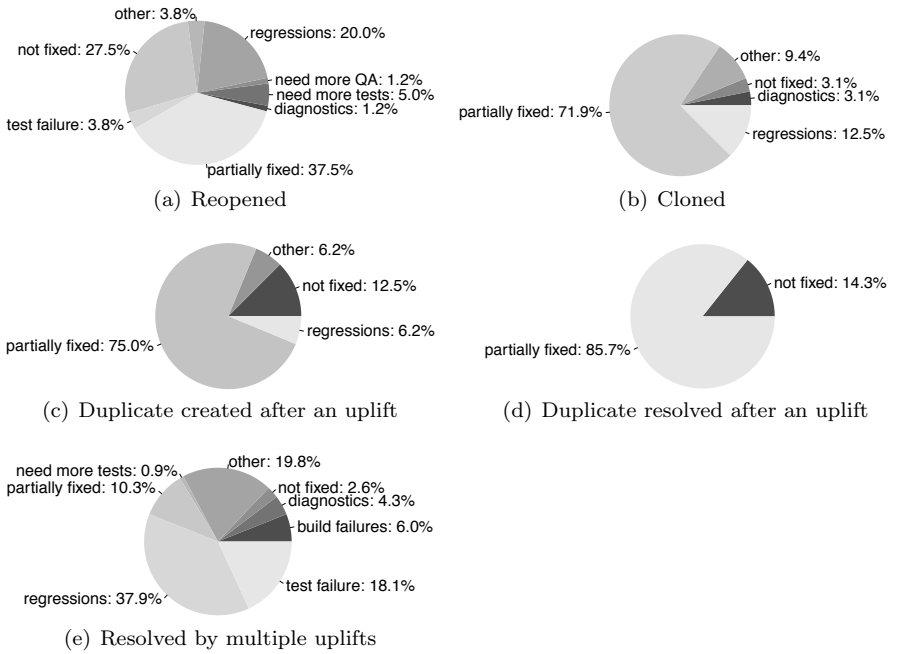
"The original uplifted patches did not completely fix the problem" is the most frequent root cause behind the issues that were ineffectively fixed and were later reopened, cloned, or duplicate. An example of such case is issue #1156182; the original uplifted patch of issue #1156182[4] only fixed the crash problem on Windows. The issue was reopened to further fix crashes on Linux.

"Leading to regressions" is another important frequent root cause of the issues that were reopened, cloned, and were resolved by multiple uplifts. An example of such case is issue #1044975; after uplifting and landing a patch to the Aurora and Release channels to fix crashes of issue #1044975[5], developers noticed an increase of crashes with another stack trace in the field. They had to uplift another patch to address the regressions.

In addition, among the ineffective uplifts, 27.5% of the issues were reopened after patch uplifts because these patches did not resolve the issues at all. 18.1%

[4] `https://bugzilla.mozilla.org/show_bug.cgi?id=1156182`

[5] `https://bugzilla.mozilla.org/show_bug.cgi?id=1044975`



(a) Reopened (b) Cloned

(c) Duplicate created after an uplift (d) Duplicate resolved after an uplift

(e) Resolved by multiple uplifts

Fig. 5 Root causes of the ineffective uplifts.

of the issues were resolved by multiple uplifts because their first uplifted patch did not pass a test case. Test and build failures happen because the patch from the Nightly version is applied to an earlier version (Beta or Aurora), so the rest of the code might be different. In the current workflow, the uplift is published only after the uplift is accepted. In other words, build or test failures can only be detected after an uplift is approved. If a developer does not fix a problem quickly enough, the uplift might be published later than it could have, thus missing one or more Beta builds (which are made twice a week), which means reducing the time dedicated to manual testing. In the data we have collected, build or test failures caused on average around four days lost on Aurora and around three days lost on Beta. This means loosing four days of testing on Aurora, and almost one week of testing on Beta (since there are only two Beta builds per week). We suggest that Mozilla performs "uplift simulations", *i.e.*, notifying developers whether their patch causes build or test failures as soon as they request an uplift, instead of after the uplift is approved.

Moreover, we observed that 9 out of the 77 reopened issues did not completely get resolved, which were further filed as cloned or duplicate issues. For example, issue #1154003[6] was created due to crashes in the drawing method `DrawingContext::FillRectangle`. After uplifting a patch to the Aurora and Release channels, developers still observed a high volume of crashes with the same signature. To address the missing edge cases of these crashes, developers

6 `https://bugzilla.mozilla.org/show_bug.cgi?id=1154003`

cloned the issue into issue #1162520[7]. This finding inspired us to investigate whether the cloned and duplicate issues were resolved in the same version as their original issues or resolved in a later version. We found that 23 out of the 54 (32+15+7) cloned or duplicate issues were resolved in the same version as their original issues, and the other 32 issues were resolved in a later version.

In this study, we only target for closed issues, but during our manual analysis, we observed that some issues fixed by uplifted patches have not been eventually closed. For example, issue #1297390[8] was created as a follow-up to the crashes fixed in issue #1280110[9]. Issue #1297390 has not been closed because the crash volume decreased again to a relatively low level. The priority of this issue were adjusted to P3, *i.e.*, would like to fix, but waiting for resources [23]. Although it would be interesting to investigate how many issues fixed by ineffective uplifts have been "completely and eventually" resolved, we can hardly get an exact answer because first, our subject dataset is dated from September 2014 to August 2016. Answering this question is beyond the scope of our study. Second, developers and testers can hardly know whether the most recent patch has covered all possible aspects to fix a certain issue, in other words, a "fixed" problem may come back again in the future. A lesson from this finding is that some issues are more difficult to get fixed than others. If an issue has recurred in the field, a proper follow-up is required even after the issue has been closed.

Regarding the differences of the ineffective uplift among channels, we observed that 153 out of the 4,368 (3.5%) Aurora uplifts, 112 out of the 2,614 (4.3%) Beta uplifts, and 16 out of the 285 (5.6%) Release uplifts were ineffective. Although the strictness of the uplift rules increases from Aurora, Beta, and to Release, the prevalence of ineffective uplifts does not decrease accordingly in these channels. The percentages vary among different kinds of ineffective uplifts, in particular "not fixed" uplifts account for 0.5% in Aurora, 0.9% in Beta, and 2.5% in Release. A possible reason could be that patches uplifted to the Release channel are aimed at more critical problems, which might be harder to fix. We looked in more detail at the "not fixed" cases in Release. It turns out that these uplifts indeed often fix very hard issues that occur in not-easily reproducible scenarios (even though they affect many users), thus developers are forced to fumble around in the dark, attempting tentative fixes that sometimes do not work at all. However, we still suggest that release managers enhance the review effort on the Release uplifts, because these patches are targeted to the most stabilized version and most users of the product. Releasing updates to them without fixing the issues might be counterproductive.

According to our results, we suggest that developers and testers should carefully inspect whether a patch has completely resolved an issue and verify whether the patch has covered all possible scenarios of the issue. They also need

7 `https://bugzilla.mozilla.org/show_bug.cgi?id=1162520`

8 `https://bugzilla.mozilla.org/show_bug.cgi?id=1297390`

9 `https://bugzilla.mozilla.org/show_bug.cgi?id=1280110`

to examine whether the patch would lead to new problems (*i.e.*, regressions) before requesting for uplift. Some ineffective uplifts (such as those due to test and build failures) can be prevented by performing uplift simulations.

We have shown the results to the release managers, who observed that many times in order to mitigate risk and especially for very urgent issues, they actually request developers to either implement a workaround or a partial fix, postponing a full fix (and potential refactorings) for a subsequent release.

RQ3: What are the characteristics of uplifted patches that introduced faults in the system?

Motivation. In **RQ2**, we studied ineffective uplifts, *i.e.*, uplifted patches that need additional fixing efforts. We observed that leading to regressions is one of the reasons of these ineffective uplifts. In this research question, we focus on the uplifted patches that introduced new regressions. 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 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. Understanding the characteristics of these "fault-inducing uplifts" can help software organizations focus their QA and code review efforts on specific kinds of uplifts to prevent users' frustration.

1) Quantitative Analysis

Approach. To discover all possible fault-inducing uplifts, we applied the SZZ algorithm (described in Section 3.2.2) on all fault-fixing changes to identify uplifted patches that introduced a fault in the system. Next, we classified the uplifted patches into two groups: fault-inducing uplifts and clean uplifts. We used the 22 metrics listed in Tables 1 to 5 to assess the differences between these two groups. For each (m_i) metric, we tested 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 used the Mann-Whitney U test and Cliff's Delta effect size to accept or reject the hypotheses, and assessed the magnitude of the differences between fault-inducing uplifts and clean uplifts. We also tested the hypotheses for all three channels.

Results. Table 10 summarizes differences between the characteristics of uplifted patches that introduced a fault in the system and those that did not. We observed 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

Table 10 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

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 10), 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 rejected 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 observed that patches with high complexity and centrality tend to lead to faults. Uplift requests submitted by

Table 11 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.

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 conducted 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 [39], we defined seven possible root causes for uplift faults (as shown in Table 11). 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-inducing patches that you are not currently taking enough into account but could be considered in the future?*

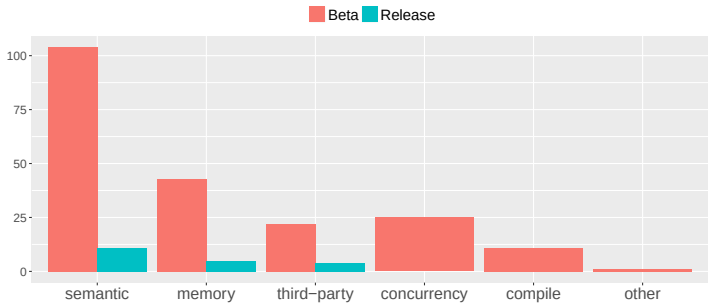


Fig. 6 Reasons of fault-inducing uplifts.

Results. Figure 6 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 found 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 did 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 guessed 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 [7]) 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.

Table 12 Categories of uplift reasons and regression impact. The severity is ranked by descending order (1 represents the most severe reason; while 6 represents the least severe reason).

Reason	Description	Severity
Security	Same as *security* in Table 7.	1
Crash	*crash + hang.*	2
Broken functionality (func)	*func + web compat + addon compat + rendering.*	3
Performance degradation (perf)	Same as *perf* in Table 7.	4
Improvement or new feature (improve)	*improve + feature.*	5
Compile or test problem (compile)	*compile + test.*	6
Other	Same as *other* in Table 7.	6

RQ4: Are regressions caused by uplift more severe than the bugs that were fixed with the uplift?

Motivation. In **RQ3**, we found that some uplift patches lead to regressions. For these patches, following an observation from the release managers, we are curious to compare their potential impact with the impact of the regression they lead to. We would suggest developers to carefully uplift certain kinds of patches if the patches have often caused more severe problems than what they intended to address.

Approach. We performed a manual analysis on the uplifted patches that were examined in **RQ3**. For each of these patches, two of the authors independently identified: 1) the problem the patch aims to address (noted as "original problem"), and 2) the impact of the regression the patch caused (noted as "regression problem"). To facilitate the comparison on the severity level between the original problem and the regression problem, we merged some of the categories (which have the same severity) defined in Table 7 as in Table 12. We also ranked the severity among different uplifted reasons (or regressions).

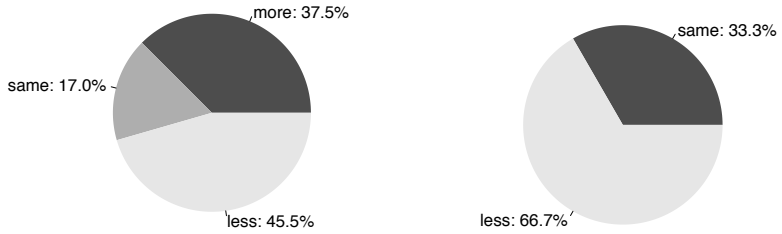
In some cases, the uplift and regression problems belong to the same category, but they affect users to a different extent. For example, issue #1059797[10] (which was uplifted to address a hang problem) caused a regression as issue #1239789[11] (which is a crash problem). Although crash and hang are considered to have the same level of severity, the first issue only happened during test runs, whereas the second one can be reproduced by users. To reduce any biases in the above rule, we also carefully examined the severity of the issues that belong to different categories. For example, issue #1075199[12] (which was uplifted to add a mock GMP plugin for testing) caused issue #1160914[13] (which is a crash). Although the latter is a crash, it only affects the plugin

[10] `https://bugzilla.mozilla.org/show_bug.cgi?id=1059797`

[11] `https://bugzilla.mozilla.org/show_bug.cgi?id=1239789`

[12] `https://bugzilla.mozilla.org/show_bug.cgi?id=1075199`

[13] `https://bugzilla.mozilla.org/show_bug.cgi?id=1160914`



(a) Beta channel (b) Release channel

Fig. 7 Whether the regression an uplift caused is more severe than the problem the uplift aims to address.

Table 13 The frequency and probability of a regression that an uplift in the Beta channel can lead to (rows in *italic* indicates that the regression is more severe than the problem the uplift intended to address).

Uplift	Regression	Frequency	Probability
compile	*crash*	*2*	*0.67*
compile	compile	1	0.33
crash	crash	24	0.50
crash	func	13	0.27
crash	compile	5	0.10
crash	perf	3	0.06
crash	other	2	0.04
crash	*security*	*1*	*0.02*
func	func	35	0.57
func	*crash*	*14*	*0.23*
func	perf	7	0.11
func	compile	4	0.07
func	other	1	0.02
improve	*crash*	*7*	*0.37*
improve	*func*	*7*	*0.37*
improve	compile	2	0.11
improve	*perf*	*2*	*0.11*
improve	*security*	*1*	*0.05*
perf	*func*	*5*	*0.50*
perf	*crash*	*4*	*0.40*
perf	perf	1	0.10
security	func	8	0.33
security	crash	7	0.29
security	security	5	0.21
security	compile	2	0.08
security	other	1	0.04
security	perf	1	0.04

used for testing, *i.e.*, it has no impact on end users. Thus, we considered that the former is more important.

Results. Figure 7 depicts the proportion of uplifted patches that caused a more, same, or less severe regression. Tables 13 and 14 show the frequency and probability of a regression that an uplift on the Beta or Release channel can lead to.

In the Beta channel, more than one third (37.5%) of the manually examined uplifted patches led to a regression that is more severe than the problem they intended to address. Most of these patches were

Table 14 The frequency and probability that an uplift in the Release channel can lead to.

Uplift	Regression	Frequency	Probability
crash	func	6	0.55
crash	crash	5	0.45
func	func	1	0.50
func	perf	1	0.50
security	func	2	0.50
security	security	2	0.50

used to introduce improvements or new features (but caused crashes/hangs and broken functionalities), to fix broken functionalities (but caused crashes/hangs), or to fix performance degradation (but caused crashes/hangs and broken functionalities). In addition, we observed that crash/hang and broken functionality are the most frequent and the most probable regressions, which ranked as the top regression for each type of the analyzed uplifts. Especially, 50% of the patches uplifted to fix a crash caused other crashes, and 50% of the patches uplifted to fix a broken functionality broke other functionalities. Regarding the patches uplifted for security vulnerabilities (which have the worst impact on users), 21% of them caused other severity vulnerabilities and 29% of them caused crashes/hangs.

In the Release channel, none of the examined uplifted patches led to a regression that is more severe than the problem the patches intended to address. This result is expected because patches uplifted for the Release channel should have been more strictly reviewed and approved. The examined patches are only used to fix security vulnerabilities, crashes/hangs, and broken functionalities, which respected the uplift rules for the Release channel. 33.3% of these patches led to a regression as the same type of problem they intended to address. All these patches have a high probability to cause a new broken functionality.

In general, developers and release managers should carefully uplift patches that aim to fix security vulnerabilities, crashes/hangs, or broken functionalities because these patches may lead to the same kind of problems they intend to address and these problems have the worst impact on end users. Uplifting patches that aim to introduce improvement (or new features) or to fix performance degradation should also be prudently inspected because these patches may cause regressions that are more severe than the problem they intended to address. Although none of the examined patches that were uplifted to the Release channel caused a more severe regression than what they intended to address, around half of the patches fixing the top severe problems (*i.e.*, crash/hang or severity problems) caused other severe problems. More QA effort needs to be invested on these patches, to avoid releasing severe regression to users.

Release managers were, as one might have predicted, happy to see our results regarding the release channel, but were not surprised because, compared to other channels, release uplifts are inspected with more QA efforts and are more carefully approved. When using the metrics listed in Tables 1

to 5 to compare the differences between Beta uplifts that caused more severe regressions than they fixed and other manually analyzed Beta uplifts[14], we observed that the former uplifts tended to happen closer to the release date and tended to have a shorter review duration (but these results are not statistically significant as the sample we analyzed is probably small). Release managers thought that these patches might have been uplifted in a rush and under pressure, which would explain both the closeness to the release date and the short review duration.

RQ5: Could some of the regressions have been prevented through more extensive testing on the channels?

Motivation. Given the results of **RQ2**, we set out to find whether any regressions could actually have been prevented by more extensive testing on the stabilization channels. In this research question, we tried to identify, from a selected sample of regressions that hit users, which issues were reproducible and how they were found by Mozilla. Our result can inform developers and release managers whether more extensive testing efforts would be effective in preventing regressions and how many regressions could possibly be prevented. It should be noted that there is an important trade-off that release managers take into account when deciding about uplifts: the necessity of shipping features as fast as possible versus the need to not introduce regressions. More extensive testing efforts might improve the second aspect, but hamper the first.

Approach. To identify regressions that were shipped to users (that is, the regressions caused by patches that were uplifted to a version of Firefox and fixed only in a later version of Firefox; for example, a patch that is uplifted to Firefox 57 and causes a regression that is only fixed in Firefox 58), we used Bugzilla status flags (`cf_status_firefox`), which specify the status of the issue for a given Firefox version (*e.g.*, `cf_status_firefox48` set to "affected" means that the issue affects Firefox 48). In particular, "affected" means that the issue exists for the given version; "wontfix" means that the issue exists and that Mozilla does not plan on fixing it for that specific version; "fixed" means that the issue is fixed in the given version; "verified" means that the issue is fixed in the given version and is also verified to be fixed either by the reporter, QA, a volunteer, or a developer who could reproduce the problem (but not by the developer who fixed it). Given an uplift fixing Issue A and a resulting regression tracked in Issue B, we identified it as being shipped to users if Issue A was set as fixed or verified in an earlier version than Issue B.

We then manually analyzed the identified regressions, categorizing both whether an issue was reproducible and how the issue was found. We have

[14] Please refer to the detailed comparison in our data repository: `https://github.com/swatlab/uplift-analysis`

Table 15 How an uplift regression is reproducible.

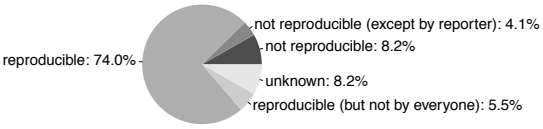
Reproducible	Description
By all	Everybody was able to reproduce.
By some	Somebody was able to reproduce (depending for example on the version of a driver, or a specific version of an operating system, and so on).
By the reporter only	Nobody else except the reporter was able to reproduce.
By no one	Nobody was able to reproduce (and the issue was found, for example, by analyzing crash reports).

Table 16 How a regression was found.

Found	Description
By tooling	The issue was found by fuzzing or static analysis.
By developers	The issue was found by Mozilla developers (by code inspection, by running tests that were not included in Firefox' test suites, or by running special tools such as Valgrind or ASan) or by an external developer (*e.g.*, a security researcher).
On a widely used feature/website/-config	The issue was found by a user (an end-user, a volunteer, or a website developer) on a widely used feature, on a widely used website, or in a widespread configuration.
On a rarely used feature/website/-config	The issue was found by a user on a rarely used feature or rarely used website or on an uncommon configuration.
Via telemetry	The issue was found by analyzing crash reports or performance measurements from the field.

analyzed all Release regressions, and a representative sample of 152 Beta regressions (which corresponds to a confidence level of 95% and a confidence interval of 5%).

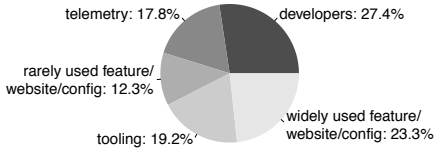
Table 15 and Table 16 show and describe how an uplift regression is reproducible and how it was found. We considered the regressions as *possibly preventable* by additional testing if they were not only reproducible by the issue reporter and were found either on a widely used feature/website/config or via telemetry. If they were reproducible only by the issue reporter, additional testing would not help. The regressions found via telemetry could be prevented if the data (crash reports and measurements) were analyzed in a timely manner (for example if there was an alerting system in place). We considered the regressions as *not easily preventable*, if they were reproducible but found on a rarely used feature/website/configuration, or found via telemetry but not reproducible, since manual testing is likely going to focus on widely used features/websites/configurations rather than seldom used ones, and issues noticed via telemetry are harder to fix if they cannot be reproduced. We consider the remaining regressions as *hardly preventable*: the regressions found by tooling could hardly be prevented, as the specific tooling was not available at the time the uplift was made (they could be prevented now that it is available); the regressions found by developers (*e.g.*, by code inspection) could hardly be prevented by additional testing. They could, in some cases, be mitigated by more detailed code reviews.



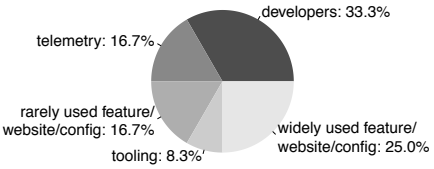
(a) Beta channel

(b) Release channel

Fig. 8 Whether the regressions caused by an uplift were reproducible.



(a) Beta channel



(b) Release channel

Fig. 9 How the regressions caused by uplifts were found.

Results. Figure 8 shows the proportion of reproducibility on the regressions. On Beta, 58 out of 73 regression issues were reproducible by all or by some developers, 9 were not reproducible or reproducible only by the reporter. The reproducibility of the remaining 6 regressions cannot be identified. On Release, 10 out of 12 were reproducible by all or by some developers, 2 were not reproducible or reproducible only by the reporter. To summarize, **79.5% of the regressions caused by Beta uplifts and 83.3% of the regressions caused by Release uplifts were reproducible**.

Figure 9 shows the distribution of ways through which the regressions were found by Mozilla. In Beta, 20 regressions were found by developers, 14 were found by tooling, 13 were found via telemetry, 17 were found by users on widely used features/websites/configurations, 9 were found on rarely used features/websites/configurations. In Release, 4 were found by developers, 1 was found by tooling, 2 were found via telemetry, 3 were found by users on widely used features/websites/configurations, 2 were found on rarely used features/websites/configurations.

Between the two channels, both the reproducibility and how the issues were found have similar characteristics (*i.e.*, the proportions are very similar), as can be seen from the figures mentioned above.

In order to understand the share of regressions that could have possibly been prevented, we compare the numbers of the possibly preventable, not easily preventable, and hardly preventable regressions in each channel. **In Beta, 20 regressions (around 30%) could have been possibly prevented**

according to our definition; 13 regressions (around 20%) could not be prevented easily; 34 regressions (around 50%) could hardly be prevented. **In Release, 3 regressions (around 25%) could have been possibly prevented according to our definition**; 3 regressions (around 25%) could not be prevented easily; 6 regressions (around 50%) could hardly be prevented. We notice that the proportions are similar between the two channels; meaning that our discussion applies to both channels.

From these results, we suggest that developers and release managers should:

1. Try to detect issues via telemetry as early as possible (*e.g.*, using alerting systems), so that they can also be fixed in time;
2. Perform more QA on the stabilization channels, *e.g.*, trying more diverse configurations, as around 24% of the issues were reproducible and found on widely used features.

Coming back to the trade-off aspect we briefly discussed in the "Motivation" part, it applies to our suggestions too. An effective alerting system should not need to collect data for a long time before being able to produce alerts, otherwise if release managers had to wait in order to check whether there are alerts, the release process would be slowed down (in this case, a higher number of users on the stabilization channels might help because the more users the more quickly data is available to make decisions). The same applies to QA, in the best case, the QA efforts should be increased in a parallel way or should be more directed towards widely used features, to avoid slowing down the release process.

Release managers have recently introduced changes to avoid regressions like these to go unnoticed: Mozilla now performs QA on the Nightly channel for new features directly when they are introduced. This allows more time to detect regressions and to fix them. We found (not a statistically significant result probably due to the small size of the sample) that the possibly preventable issues tend to have been on Nightly for longer (higher landing delta), but tend to be uplifted later, closer to the release date (lower release delta)[15]. Given the additional QA on the Nightly channel, the situation of regressions (at least for the issues that could possibly be prevented by additional QA) may be improved soon. Verifying the potential improvement will be a part of our future work.

5 Threats to Validity

In this section, we discuss the threats to validity of our study following the guidelines for case study research [46].

Construct validity threats are concerned with the relationship between theory and observation. In this study, the construct validity threats are mainly due to measurement errors. In **RQ2**, to find ineffective uplifts, we looked for

[15] Please refer to the detailed comparisons in our data repository: `https://github.com/swatlab/uplift-analysis`

cases where an issue linked to the uplift had been, after the uplift operation, reopened, cloned, duplicate, or resolved by multiple patches. To prevent false positive results due to this heuristic, we took a series of measures to remove noisy results from our dataset (see the "Approach" part of **RQ2**) and manually examined all candidates of ineffective uplifts. We believe that the eventually included results have a high precision. In addition, some correct candidates might not be detected by our heuristic, *i.e.*, the false negatives. For example, some ineffective uplifts can be beyond our expected cases (such as reopened, cloned or duplicated issues) or mislabelled by developers in Bugzilla. However, instead of finding all possible ineffective uplifts, the aim of this research questions is to identify precise and representative ineffectively uplifted patches, analyzing their characteristics and propose methods for software practitioners to avoid them. In **RQ3**, we observed that uplifted patches with more lines of code are more likely to be fault-inducing. This result is not surprising if we assume that the fault density is uniformly distributed in the studied system. Nevertheless, as suggested by previous studies, software practitioners should always carefully approve patches that modify a large number of lines.

Internal validity threats concern factors that affect the independent variable with respect to causality. Since we do not draw any casual conclusion, threats to the internal validity are not applicable for our study.

Conclusion validity threats concern the relationship between the treatments and the outcome. We paid attention not to violate the assumptions of the statistical tests that are performed in the paper. Specifically, in **RQ1** and **RQ3**, we applied non-parametric tests that do not require making assumptions on the distribution of our dataset. We used SentiStrength as the sentiment detection tool. We compared the performance of this tool with SentiStrengthSE [14], the version tailored for software engineering, and obtained the same results, *i.e.*, no significant differences between accepted and rejected uplifts in any channel, and only a small effect size of the differences on the module owners' sentiment between clean and fault-inducing uplifts. Another reason why we prefer SentiStrength over SentiStrengthSE is that the former tool can be used from the command line and can be easily integrated into our automated scripts. On the contrary, the latter tool can currently only be executed from a user interface. In addition, when ingesting a large dataset such as the one we used in this study, the latter tool cannot be as easily deployed into a distributed environment. 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 manual classifications on the uplift reasons, the root causes of uplift regressions and reoccurrences, the reproducibility of the uplift regressions, and the way by which developers

were discovered the regressions. We also manually compared the severity of the issues that the uplifts intended to address with the severity of the regressions that they led to. 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. Last, we used a heuristic to detect issues that duplicate a previous issue fixed by uplifted patches, which was inspired by Tian et al.'s approach [41]. Besides the automated detection, we manually confirmed every case used in our analyses to answer **RQ2**. Although some true positive cases might have been missed, the goal of **RQ2** is not to find all duplicate cases, but to understand why some uplifted patches did not completely resolve a problem and re-occurred in the field.

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 desirable to further validate our results. To facilitate future replication studies, we share our datasets and scripts at: `https://github.com/swatlab/uplift-analysis`. Another issue is that, in the manual classification, 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.

6 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" [2].

Since the adoption of the rapid release model [16] by Mozilla in 2011, a plethora of studies have focused on the impact of rapid release strategies on software quality. Khomh et al. [16] 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. Khomh et al. observed, in their extended work [17], that one of the major challenges of fast release cycles is the automation of the release engineering process. Da Costa et al. [9] 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. Adams et al. [1] analyzed the six major phases of release engineering practices and proposed a roadmap for future research, highlighting the need for more empirical studies that validate the best practices and assess the impact of release engineering processes on software quality.

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. [12] 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 suggest 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. [19] 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. [29] 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. [30] observed that feature toggles [20] can be 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.

7 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 (RQ1). We found that 4% of the issues fixed by patch uplift were not effectively resolved but were later reopened, cloned, duplicated, or fixed by additional uplifts. Two frequent root causes were identified from our manual analysis, *i.e.*, the original uplifts only partially fixed the issues or caused regressions (RQ2). 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 (RQ3). In addition, through a manual analysis on a sample of the uplifts that introduced regressions, we found that more than one third of the fault-inducing Beta uplifts led to a regression that is more severe than the problem they aimed to address (RQ4). Last but not least, we observed that 25% to 30% of the regressions due to Beta and Release uplifts could be possibly prevented because they can be reproduced not only by the issue reporter but also by developers and were found on widely used feature/website/configuration or via the Mozilla telemetry (RQ5). We hope that software organizations take our findings and suggestions as a reference to improve their uplift (or urgent patch approval) strategy.

References

1. Adams B, McIntosh S (2016) Modern release engineering in a nutshell–why researchers should care. In: Software Analysis, Evolution, and Reengineering (SANER), 2016 IEEE 23rd International Conference on, IEEE, vol 5, pp 78–90
2. Adams B, Bellomo S, Bird C, Marshall-Keim T, Khomh F, Moir K (2015) The practice and future of release engineering: A roundtable with three release engineers. IEEE Software 32(2):42–49
3. An L, Khomh F (2015) 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

4. An L, Khomh F, Adams B (2014) Supplementary bug fixes vs. re-opened bugs. In: Source Code Analysis and Manipulation (SCAM), 2014 IEEE 14th International Working Conference on, IEEE, pp 205–214
5. Castelluccio M, An L, Khomh F (2017) Is it safe to uplift this patch?: An empirical study on mozilla firefox. In: Software Maintenance and Evolution (ICSME), 2017 IEEE International Conference on, IEEE, pp 411–421
6. Cliff N (1993) Dominance statistics: Ordinal analyses to answer ordinal questions. Psychological Bulletin 114(3):494
7. coverity.com (2017) Coverity tool. `http://www.coverity.com`, online; Accessed March 31st, 2017
8. Csardi G, Nepusz T (2006) The igraph software package for complex network research. InterJournal, Complex Systems 1695(5):1–9
9. Da Costa DA, McIntosh S, Kulesza U, Hassan AE (2016) 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), pp 374–385
10. Dmitrienko A, Molenberghs G, Chuang-Stein C, Offen W (2005) Analysis of Clinical Trials Using SAS: A Practical Guide. SAS Institute, URL `http://www.google.ca/books?id=G5ElnZDDm8gC`
11. Fischer M, Pinzger M, Gall H (2003) Populating a release history database from version control and bug tracking systems. In: Proceedings of the 29th International Conference on Software Maintenance (ICSM), IEEE, pp 23–32
12. Hassan S, Shang W, Hassan AE (2016) An empirical study of emergency updates for top android mobile apps. Empirical Software Engineering pp 1–42
13. Hollander M, Wolfe DA, Chicken E (2013) Nonparametric statistical methods, 3rd edn. John Wiley & Sons
14. Islam MR, Zibran MF (2017) Leveraging automated sentiment analysis in software engineering. In: Mining Software Repositories (MSR), 2017 IEEE/ACM 14th International Conference on, IEEE, pp 203–214
15. Jalbert N, Weimer W (2008) Automated duplicate detection for bug tracking systems. In: Dependable Systems and Networks With FTCS and DCC, 2008. DSN 2008. IEEE International Conference on, IEEE, pp 52–61
16. Khomh F, Dhaliwal T, Zou Y, Adams B (2012) 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, pp 179–188
17. Khomh F, Adams B, Dhaliwal T, Zou Y (2014) Understanding the impact of rapid releases on software quality. Empirical Software Engineering pp 1–38
18. Kim D, Wang X, Kim S, Zeller A, Cheung SC, Park S (2011) Which crashes should I fix first?: Predicting top crashes at an early stage to prioritize debugging efforts. IEEE Transactions on Software Engineering 37(3):430–447

19. Lin D, Bezemer CP, Hassan AE (2016) Studying the urgent updates of popular games on the steam platform. Empirical Software Engineering pp 1–32
20. martinfowler.com (2017) Feature toggle. https://martinfowler.com/bliki/FeatureToggle.html, online; Accessed March 22nd, 2017
21. MDN Web Docs (2017) A Bug's Life. https://developer.mozilla.org/en-US/docs/Mozilla/QA/A_Bugs_Life, online; Accessed May 20th, 2018
22. Mike T, Kevan B, Georgios P, Di C, Arvid K (2010) Sentiment in short strength detection informal text. JASIST 61(12):2544–2558
23. Mozilla wiki (2016) Priority Field. https://wiki.mozilla.org/Bugmasters/Projects/Folk_Knowledge/Priority_Field, online; Accessed May 20th, 2018
24. Mozilla wiki (2018) Mozilla Modules. https://wiki.mozilla.org/Modules, online; Accessed September 22nd, 2018
25. Mozilla wiki (2018) Mozilla Release Management Uplift Rules. https://wiki.mozilla.org/Release_Management/Uplift_rules, online; Accessed May 20th, 2018
26. Mozilla wiki (2018) Mozilla Tree Sheriffs. https://wiki.mozilla.org/Sheriffing, online; Accessed May 20th, 2018
27. Mozilla wiki (2018) Mozilla Tree Sheriffs - Backouts. https://wiki.mozilla.org/Sheriffing/How_To/Backouts, online; Accessed September 22nd, 2018
28. Park J, Kim M, Ray B, Bae DH (2012) An empirical study of supplementary bug fixes. In: Proceedings of the 9th IEEE Working Conference on Mining Software Repositories, IEEE Press, pp 40–49
29. Rahman MT, Rigby PC (2015) Release stabilization on linux and chrome. IEEE Software 32(2):81–88
30. Rahman MT, Querel LP, Rigby PC, Adams B (2016) Feature toggles: practitioner practices and a case study. In: Proceedings of the 13th International Conference on Mining Software Repositories, ACM, pp 201–211
31. Romano J, Kromrey JD, Coraggio J, Skowronek J (2006) Appropriate statistics for ordinal level data: Should we really be using t-test and cohensd for evaluating group differences on the nsse and other surveys. In: annual meeting of the Florida Association of Institutional Research, pp 1–33
32. Runeson P, Alexandersson M, Nyholm O (2007) Detection of duplicate defect reports using natural language processing. In: Proceedings of the 29th international conference on Software Engineering, IEEE Computer Society, pp 499–510
33. scitools.com (2016) Understand tool. https://scitools.com, online; Accessed March 31st, 2016
34. Shihab E, Ihara A, Kamei Y, Ibrahim WM, Ohira M, Adams B, Hassan AE, Matsumoto Ki (2012) Studying re-opened bugs in open source software. Empirical Software Engineering pp 1–38
35. SlideShare (2016) A. Laforge. Chrome release cycle. Job title: Technical Program Manager (Chrome) at Google. http://www.slideshare.net/

Jolicloud/chrome-release-cycle, online; Accessed 06 February 2016
36. Śliwerski J, Zimmermann T, Zeller A (2005) When do changes induce fixes? In: ACM sigsoft software engineering notes, ACM, vol 30, pp 1–5
37. Sun C, Lo D, Wang X, Jiang J, Khoo SC (2010) A discriminative model approach for accurate duplicate bug report retrieval. In: Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1, ACM, pp 45–54
38. Sun C, Lo D, Khoo SC, Jiang J (2011) Towards more accurate retrieval of duplicate bug reports. In: Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering, IEEE Computer Society, pp 253–262
39. Tan L, Liu C, Li Z, Wang X, Zhou Y, Zhai C (2014) Bug characteristics in open source software. Empirical Software Engineering 19(6):1665–1705
40. The Bugzilla Guide (2017) The Bugzilla Guide. https://www.bugzilla.org/docs/2.20/html/bugreports.html, online; Accessed May 20th, 2018
41. Tian Y, Sun C, Lo D (2012) Improved duplicate bug report identification. In: Software Maintenance and Reengineering (CSMR), 2012 16th European Conference on, IEEE, pp 385–390
42. Tourani P, Adams B (2016) The impact of human discussions on just-in-time quality assurance: An empirical study on openstack and eclipse. In: Software Analysis, Evolution, and Reengineering (SANER), 2016 IEEE 23rd International Conference on, IEEE, vol 1, pp 189–200
43. VMware (2017) JIRA. https://jira.atlassian.com/, accessed March 30th, 2017
44. Wang X, Zhang L, Xie T, Anvik J, Sun J (2008) An approach to detecting duplicate bug reports using natural language and execution information. In: Software Engineering, 2008. ICSE'08. ACM/IEEE 30th International Conference on, IEEE, pp 461–470
45. Wikipedia (2018) Okapi BM25. https://en.wikipedia.org/wiki/Okapi_BM25, online; Accessed May 20th, 2018
46. Yin RK (2002) Case Study Research: Design and Methods - Third Edition, 3rd edn. SAGE Publications
47. YouTube (2014) Keynote of the 2014 Release Engineering conference. https://www.youtube.com/watch?v=Nffzkkdq7GM, online; Accessed March 30th, 2017