Do Faster Releases Improve Software Quality?

—An Empirical Case Study of Mozilla Firefox—

Foutse Khomh[1], Tejinder Dhaliwal[1], Ying Zou[1], Bram Adams[2]

[1] Dept. of Elec. and Comp. Engineering, Queen's University, Kingston, Ontario, Canada
[2] GIGL, École Polytechnique de Montréal, Québec, Canada
{foutse.khomh, tejinder.dhaliwal, ying.zou}@queensu.ca, bram.adams@polymtl.ca

***Abstract*—Nowadays, many software companies are shifting from the traditional 18-month release cycle to shorter release cycles. For example, Google Chrome and Mozilla Firefox release new versions every 6 weeks. These shorter release cycles reduce the users' waiting time for a new release and offer better marketing opportunities to companies, but it is unclear if the quality of the software product improves as well, since shorter release cycles result in shorter testing periods. In this paper, we empirically study the development process of Mozilla Firefox in 2010 and 2011, a period during which the project transitioned to a shorter release cycle. We compare crash rates, median uptime, and the proportion of post-release bugs of the versions that had a shorter release cycle with those having a traditional release cycle, to assess the relation between release cycle length and the software quality observed by the end user. We found that (1) with shorter release cycles, users do not experience significantly more post-release bugs and (2) bugs are fixed faster, yet (3) users experience these bugs earlier during software execution (the program crashes earlier).**

***Keywords*-Software release; release cycle; software quality; testing; bugs.**

I. Introduction

In today's fast changing business environment, many software companies are aggressively shortening their release cycles (*i.e.*, the time in between successive releases) to speed up the delivery of their latest innovations to customers [1]. Instead of typically working 18 months on a new release containing hundreds of new features and bug fixes, companies reduce this period to, say, 3 months by limiting the scope of the release to the new features and fixing only the most crucial bugs. For example, with a rapid release model (*i.e.*, a development model with a shorter release cycle), Mozilla could release over 1,000 improvements and performance enhancements with Firefox 5.0 in approximately 3 months [2]. Under the traditional release model (*i.e.*, a development model with a long release cycle), Firefox users used to wait for a year to get some major improvements or new features.

The concept of rapid release cycle was introduced by agile methodologies like XP [3], which claim that shorter release cycles offer various benefits to both companies and end users. Companies get faster feedback about new features and bug fixes, and releases become slightly easier to plan (short-term vs. long-term planning). Developers are not rushed to complete features because of an approaching release date, and can focus on quality assurance every 6 weeks instead of every couple of months. Furthermore, the higher number of releases provide more marketing opportunities for the companies. Customers benefit as well, since they have faster access to new features, bug fixes and security updates.

However, the claim that shorter release cycles improve the quality of the released software has not been empirically validated yet. Baysal et al. [4] found that bugs were fixed faster (although not statistically significantly) in versions of Firefox using a traditional release model than in Chrome, which uses a rapid release model. Porter et al. reported that shorter release cycles make it impossible to test all possible configurations of a released product [5]. Furthermore, anecdotal evidence suggests that shorter release cycles do not allow enough time to triage bugs from previous versions, and hence hurt the developers' chances of catching persistent bugs [6]. This is why Firefox's current high number of unconfirmed bugs has been attributed to the adoption of the 6 week-release cycle [6]. In August 2011, Firefox had about $2,600$ bugs that had not been touched since the release of Firefox 4 five months earlier. The number of Firefox bugs that were touched, but not triaged or worked on was even higher and continues to grow everyday [6].

To understand whether and how transitioning to a rapid release model can affect the quality of a software system as observed by users, we empirically study the historical field testing data of Mozilla Firefox. Firefox is a hugely popular web browser that has shifted from the traditional development model to a rapid release model. This allows us to compare the quality of traditional releases to that of rapid releases, while controlling for unpredictable factors like development process and personnel (since those largely remained constant). As measures of the quality of Firefox, we analyze the number of post-release bugs, the daily crash counts and the uptime of Firefox (*i.e.*, the time between a user starting up Firefox and experiencing a failure).

We studied the following three research questions:

RQ1) Does the length of the release cycle affect the software quality?

There is only a negligible difference in the number of post-release bugs when we control for the time interval between subsequent release dates. However, the median uptime is significantly lower for versions developed in short release cycles, *i.e.*, failures seem to occur faster at run-time.

RQ2) Does the length of the release cycle affect the fixing of bugs?

Bugs are fixed significantly faster for versions developed in a rapid release model.

RQ3) Does the length of the release cycle affect software updates?

Versions developed in a rapid release model are adopted faster by customers, *i.e.*, the proportion of customers running outdated versions that possibly contain closed security holes is reduced.

A better understanding of the impact of the release cycle on software quality will help decision makers in software companies to find the right balance between the delivery speed (release cycle) of new features and the quality of their software.

The rest of the paper is organized as follows. Section II provides some background on Mozilla Firefox. Section III describes the design of our study and Section IV discusses the results. Section V discusses threats to the validity of our study. Section VI discusses the related literature on release cycles and software quality. Finally, Section VII concludes the paper and outlines future work.

II. Mozilla Firefox

Firefox is an open source web browser developed by the Mozilla Corporation. It is currently the third most widely used browser, with approximately 25% usage share worldwide [7]. Firefox 1.0 was released in November 2004 and the latest version, Firefox 9, was released on December 20, 2011. Figure 1(a) shows the release dates of major Firefox versions. Firefox followed a traditional release model until version 4.0 (March 2011). Afterwards, Firefox adopted a rapid release model to speed up the delivery of its new features. This was partly done to compete with Google Chrome's rapid release model [8], [9], which was eroding Firefox's user base. The next subsections discuss the Firefox development and quality control processes.

A. Development Process

Before March 2011, FireFox supported multiple releases in parallel, not only the last major release. Every version of FireFox was followed by a series of minor versions, each containing bug fixes or minor updates over the previous version. These minor versions continued even after a new major release was made. Figure 1(b) shows the release dates of the minor versions of Firefox.

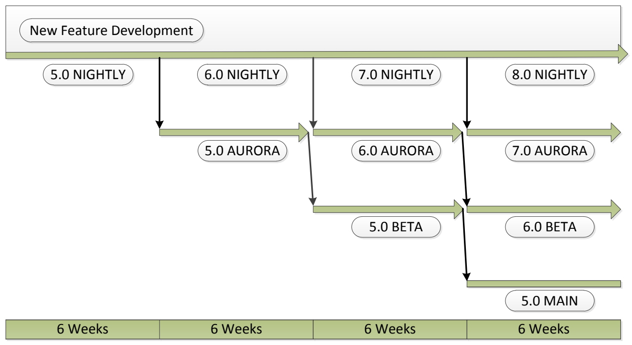

Figure 2. Development and Release Process of Mozilla Firefox

With the advent of shorter release cycles in March 2011, new features need to be tested and delivered to users faster. To achieve this goal, Firefox changed its development process. First, versions are no longer supported in parallel, *i.e.*, a new version supersedes the previous ones. Second, every FireFox version now flows through four release channels: NIGHTLY, AURORA, BETA and MAIN. The versions move from one channel to the next every 6 weeks [10]. To date, five major versions of Firefox (*i.e.*, 5.0, 6.0, 7.0, 8.0, 9.0) have finished the new rapid release model.

Figure 2 illustrates the current development and release process of Firefox. The NIGHTLY channel integrates new features from the developers' source code repositories as soon as the features are ready. The AURORA channel inherits new features from NIGHTLY at regular intervals (*i.e.*, every 6 weeks). The features that need more work are disabled and left for the next import cycle into AURORA. The BETA channel receives only new AURORA features that are scheduled by management for the next Firefox release. Finally, mature BETA features make it into MAIN. Note that at any given time (independent from the 6 week release schedule) unscheduled releases may be performed to address critical security or stability issues.

Firefox basically follows a pipelined development process. At the same time as the source code of one release is imported from the NIGHTLY channel into the AURORA channels, the source code of the next release is imported into the NIGHTLY channel. Consequently, four consecutive releases of Firefox migrate through Mozilla's NIGHTLY, AURORA, BETA, and MAIN channels at any given time. Figure 2 illustrates this migration.

B. Quality Control Process

One of the main reasons for splitting Firefox' development process into pipelined channels is to enable incremental quality control. As changes make their way through the release process, each channel makes the source code available for testing to a ten-fold larger group of users. The estimated number of contributors and end users on the

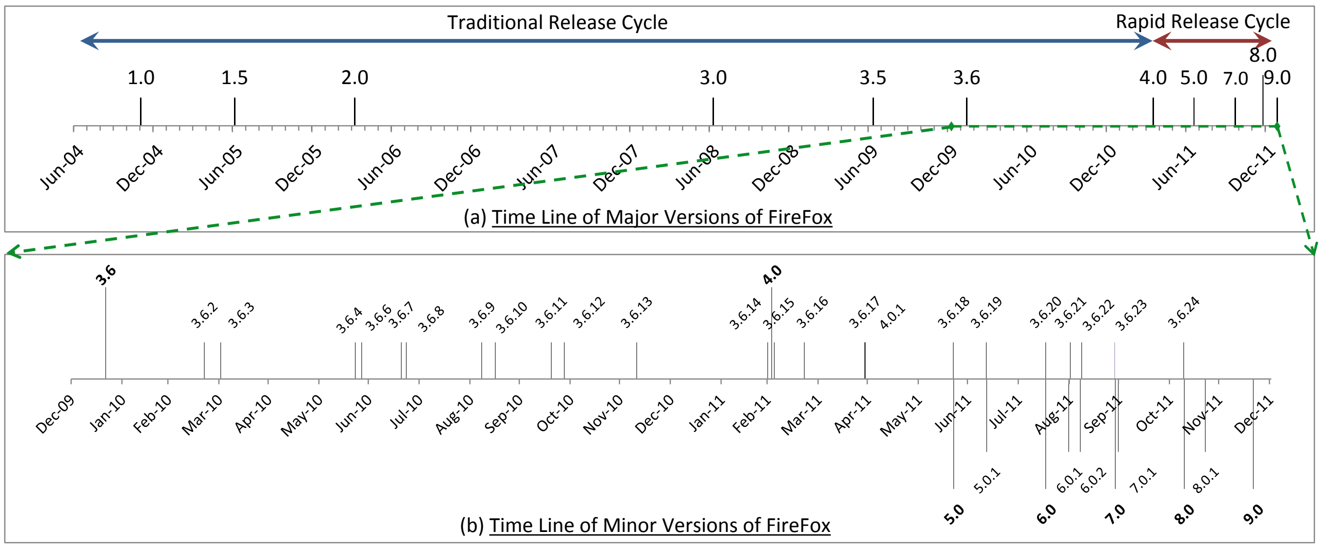

Figure 1. Timeline of FireFox versions.

channels are respectively 100,000 for NIGHTLY, 1 million for AURORA, 10 million for BETA and 100+ millions for a major Firefox version [11]. NIGHTLY reaches Firefox developers and contributors, while other channels (*i.e.*, AURORA and BETA) recruit external users for testing. The source code on AURORA is tested by web developers who are interested in the latest standards, and by Firefox add-on developers who are willing to experiment with new browser APIs. The BETA channel is tested by Firefox's regular beta testers.

Each version of Firefox in any channel embeds an automated crash reporting tool, *i.e.*, the Mozilla Crash Reporter, to monitor the quality of Firefox across all four channels. Whenever Firefox crashes on a user's machine, the Mozilla Crash Reporter [12] collects information about the event and sends a detailed crash report to the Socorro crash report server. Such a crash-report includes the stack trace of the failing thread and other information about a user environment, such as the operating system, the version of Firefox, the installation time, and a list of plug-ins installed.

Socorro groups similar crash-reports into crash-types. These crash-types are then ranked by their frequency of occurrence by the Mozilla quality assurance teams. For the top crash-types, testers file bugs in Bugzilla and link them to the corresponding crash-type in the Socorro server. Multiple bugs can be filed for a single crash-type and multiple crash-types can be associated with the same bug. For each crash-type, the Socorro server provides a crash-type summary, *i.e.*, a list of the crash-reports of the crash-type and a set of bugs that have been filed for the crash-type.

Firefox users can also submit bug reports in Bugzilla manually. A bug report contains detailed semantic information about a bug, such as the bug open date, the last modification date, and the bug status. The bugs are triaged by bug triaging developers and assigned for fixing. When a developer fixes a bug, he typically submits a patch to Bugzilla. Once approved, the patch code is integrated into the source code of Firefox on the corresponding channel and migrated through the other channels for release. Bugs that take too long to get fixed and hence miss a scheduled release are picked up by the next release's channel.

III. Study Design

This section presents the design of our case study, which aims to address the following three research questions:

1) Does the length of the release cycle affect the software quality?
2) Does the length of the release cycle affect the fixing of bugs?
3) Does the length of the release cycle affect software updates?

A. Data Collection

In this study, we analyze all versions of Firefox that were released in the period from January 01, 2010 to December 21, 2011. In total, we study 25 alpha versions, 25 beta versions, 29 minor versions and 7 major versions that were released within a period of one year before or after the move to a rapid release model. Firefox 3.6, Firefox 4 and their subsequent minor versions were developed following a traditional release cycle with an average cycle time of 52 weeks between the major version releases and 4 weeks between the minor version releases. Firefox 5, 6, 7, 8, 9 and their subsequent minor versions followed a rapid release model with an average release time interval of 6 weeks between the major releases and 2 weeks between the minor releases. Table I shows additional descriptive statistics of the different versions.

Table I
STATISTICS FROM THE ANALYZED FIREFOX VERSIONS (THE CYCLE TIME IS GIVEN IN DAYS).

	Version	Release date	Cycle time	LOC	Alpha Versions (#)	Beta Versions (#)	Minor Versions (#)
Traditional release model	3.6	21-01-2010	425	4,076,624	3.6a1pre–3.6b6pre (8)	3.6b1–3.6b6 (6)	3.6.2–3.6.24 (22)
	4.0	22-03-2011	91	4,738,536	4.0.b1pre–4.0.b12pre (12)	4.0.b1beta–4.0.1beta (14)	4.0.1 (1)
Rapid release model	5.0	21-06-2011	56	4,702,874	5.0Aurora (1)	5.0Beta (1)	5.0.1 (1)
	6.0	16-08-2011	42	4,667,335	6.0Aurora (1)	6.0Beta (1)	6.0.1, 6.0.2 (2)
	7.0	27-09-2011	42	4,653,081	7.0Aurora (1)	7.0Beta (1)	7.0.1 (1)
	8.0	08-11-2011	42	4,635,064	8.0Aurora (1)	8.0Beta (1)	8.0.1 (1)
	9.0	20-12-2011	42	4,687,901	9.0Aurora (1)	9.0Beta (1)	9.0.1 (1)

B. Data Processing

Figure 3 shows an overview of our approach. First, we check the release notes of Firefox and classify the versions based on their release model (*i.e.*, traditional release model and rapid release model). Then, for each version, we extract the necessary data from the source code repository (*i.e.*, Mercurial), the crash repository (*i.e.*, Socorro), and the bug repository (*i.e.*, Bugzilla). Using this data, we compute several metrics, then statistically compare these metrics between the traditional release (TR) model group and the rapid release (RR) model group. The remainder of this section elaborates on each of these steps.

1) Analyzing the Mozilla Wiki: For each version, we extract the starting date of the development phase and the release date from the release notes on the Mozilla Wiki. The release cycle is the time period between the release dates of two consecutive versions. We also compute the development time of the version by calculating the difference between the release date and the starting date of the development phase. The development time is slightly longer than the release cycle because the development of a new version is started before the release of the previous one.

2) Mining the Mozilla Source Code Repository: On the source code of each downloaded version, we use the source code measurement tool, SourceMonitor, to compute the number of Total Lines of Code and the Average Complexity. SourceMonitor[1] can be applied on $C{+}{+}$, C, $C\sharp$, $VB.NET$, $Java$, $Delphi$, $VisualBasic(VB6)$, and $HTML$ source code files. Such a polyvalent tool is necessary, given the diverse set of programming languages used by Firefox.

3) Mining the Mozilla Crash Repository: We downloaded the summaries of crash reports for all versions of Firefox that were released between January 21, 2010 and December 21, 2011. From these summaries, we extracted the date of the crash, the version of Firefox that was running during the crash, the list of related bugs, and the uptime (*i.e.*, the duration in seconds for which Firefox was running before it crashed).

4) Analyzing the Mozilla Bug Repository: We downloaded all Firefox bug reports related to the Firefox crashes. These reports contain both pre-release and post-release bugs. We parse each of the bug reports to extract information about the bug status (*e.g.*, UNCONFIRMED, FIXED), the bug open and modification dates, the priority of the bug and the severity of the bug. However, we cannot directly identify the major or minor version of Firefox for which the bug was raised, since this is not recorded.

Since the analyzed bugs are related to crashes, and crashes are linked to specific versions, we instead use this mapping to link the bugs to Firefox versions. For each bug, we check the crash-types for which the bug is filed. Then, we look at the crash reports of the corresponding crash-type(s) to identify the version that produces the crash-type, and we link the bug to that version. When the same crash-type contains crash reports from users on different versions, we consider that the crash-type is generated by the oldest version.

[1]http://www.campwoodsw.com/

IV. CASE STUDY RESULTS

This section presents and discusses the results of our three research questions. For each research question, we present the motivation behind the question, the analysis approach and a discussion of our findings.

A. RQ1: Does the length of the release cycle affect the software quality?

Motivation. Despite the benefits of speeding up the delivery of new features to users, shorter release cycles could have a negative impact on the quality of software systems, since there is less time for testing. Many reported issues are likely to remain unfixed until the software is released. This in turn might expose users to more post-release bugs. On the other hand, with fast release trains (*e.g.*, every 6 weeks), developers are less pressured to rush half-baked features into the software repository to meet the deadline. Hence, a rapid release model could actually introduce less bugs compared to traditional release models. Clearing up the interaction between both factors is important to help decision makers in software organizations find the right balance between the speed of delivery of new features and maintaining software quality.

Approach. We measure the quality of a software system using the following three well-known metrics:

- **Post-Release Bugs:** the number of bugs reported after the release date of a version (lower is better).

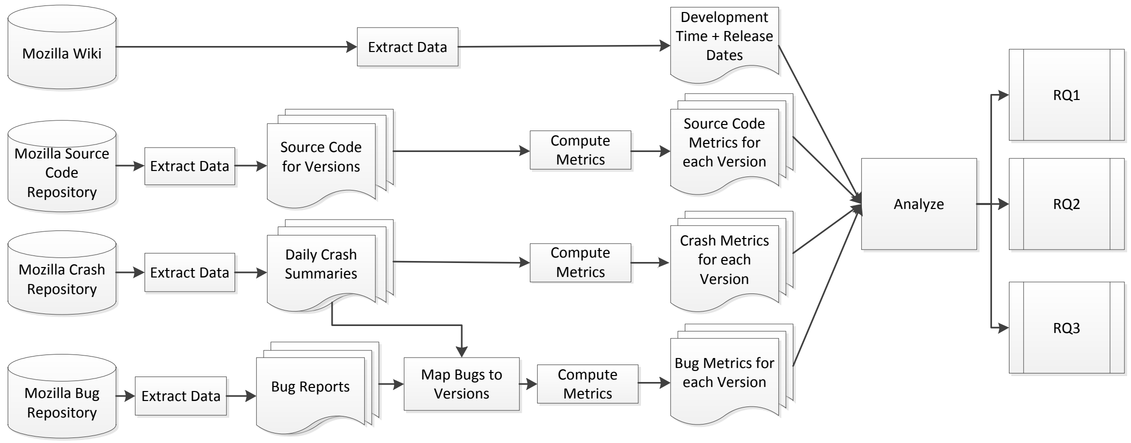

Figure 3. Overview of our approach to study the impact of release cycle time on software quality.

- **Median Daily Crash Count:** the median of the number of crashes per day for a particular version (lower is better).
- **Median Uptime:** the median across the uptime values of all the crashes that are reported for a version (higher is better).

We answer this research question in three steps. First, we compare the number of post-release bugs between the traditional release (*i.e.*, TR) and rapid release (*i.e.*, RR) groups. For each Firefox version, we consider all bugs reported after its release date. Note that we cannot perform this comparison directly. Herraiz *et al.* [13] have shown that the number of reported post-release bugs of a software system is related to the number of deployments. In other words, a larger number of deployments increases the likelihood of users reporting a higher number of bugs. Since the number of deployments is affected by the length of the period during which a release is used, and this usage period is directly related to the length of the release cycle, we need to normalize the number of post-release bugs of each version to control for the usage time. Hence, for each version, we divide the number of reported post-release bugs by the length of the release cycle of the version, and test the following null hypothesis:

H^1_{01}: *There is no significant difference between the number of post-release bugs of RR versions and TR versions.*

Second, we analyze the distribution of the median daily crash counts for RR and TR versions, and test the following null hypothesis:

H^1_{02}: *There is no significant difference between the median daily crash count of RR versions and TR versions.*

Third, we compare the median uptime of RR versions to TR versions. We test the following null hypothesis:

H^1_{03}: *There is no significant difference between the median uptime values of RR versions and TR versions.*

We use the Wilcoxon rank sum test [14] to test H^1_{01}, H^1_{02}, and H^1_{03}. The Wilcoxon rank sum test is a non-parametric statistical test used for assessing whether two independent distributions have equally large values. Non-parametric statistical methods make no assumptions about the distributions of the assessed variables.

Findings. **When controlled for the length of the release cycle of a version, there is no significant difference between the number of post-release bugs of rapid release and traditional release versions**. Figure 4 shows the distribution of the normalized number of post-release bugs for TR and RR versions, respectively. We can see that the medians are similar for RR and TR versions. The Wilcoxon rank sum test confirms this observation ($p - value = 0.3$), therefore we cannot reject H^1_{01}.

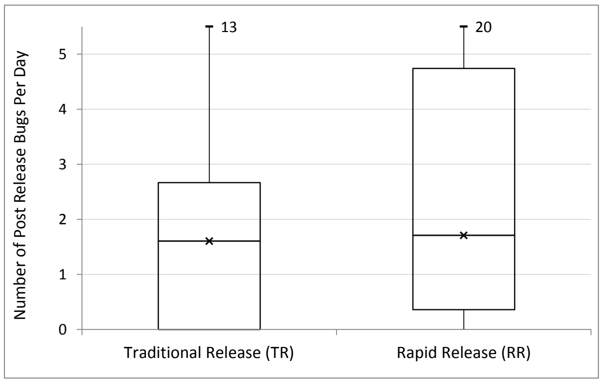

Figure 4. Boxplot of the number of post release bugs raised per day.

There is no significant difference between the median daily crash count of rapid release versions and traditional release versions. The Wilcoxon rank sum test yielded a p-value of 0.73. Again, we cannot reject H^1_{02}.

The median uptime is significantly lower for rapid release versions. Figure 5 shows the distribution of the median uptime across TR and RR versions, respectively. We can observe that the median uptime is lower for RR versions. We ran the Wilcoxon rank sum test to decide if the observed difference is statistically significant or not, and obtained a $p - value$ of $6.11e - 06$. Therefore, we reject H^1_{03}.

In general, we can conclude that although the median of daily crash counts and the number of post-release bugs are

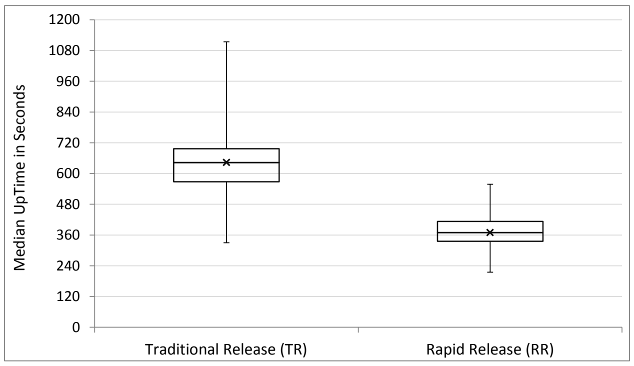

Figure 5. Boxplot of the median uptime.

comparable for RR versions and TR versions, the median uptime of RR versions is lower. In other words, although rapid releases do not seem to impact software quality directly, end users do get crashes earlier during execution (H^1_{03}), *i.e.*, the bugs of RR versions seem to have a higher show-stopper probability than the bugs of TR versions. It is not clear why exactly this happens, *i.e.*, because of a quality assurance problem or by accident (*i.e.*, one or more show-stopper bugs with a high impact).

Users experience crashes earlier during the execution of versions developed following a rapid release model.

B. RQ2: Does the length of the release cycle affect the fixing of bugs?

Motivation. For **RQ1**, we found that when one controls for the cycle time of versions, there is no significant difference between the number of post-release bugs of traditional release and rapid release versions reported per day. However, since a shorter release cycle time allows less time for testing and there was no substantial change in the development team of Firefox when switching to shorter release cycles, we might expect that the same group of developers now have less time to fix the same stream of bugs. Hence, in this question, we investigate the proportion of bugs fixed and the speed with which the bugs are fixed in the rapid release model.

Approach. For each alpha, beta, and major version, we compute the following metrics:

- **Fixed Bugs:** the number of post-release bugs that are closed with the status field set to FIXED (higher is better).
- **Unconfirmed Bugs:** the number of post-release bugs with the status field set to UNCONFIRMED (lower is better).
- **Fix Time:** the duration of the fixing period of the bug (*i.e.*, the difference between the bug open time and the last modification time). This metric is computed only for bugs with the status FIXED (lower is better).

We test the following null hypothesis to compare the efficiency of testing activities under traditional and rapid release models:

H^2_{01}: *There is no significant difference between the proportion of bugs fixed during the testing period of a RR version and the proportion of bugs fixed during the testing of a TR version.*

We consider the testing period of a version v_i to be the period between the release date of the first alpha version of v_i and the release date of v_i. As such, bugs opened or fixed during this period correspond to post-release bugs of the alpha or beta versions of v_i. To compute the proportion of bugs fixed during the testing period, we divided the number of bugs fixed in the testing period by the total number of bugs opened during the testing period. We do not further divide by the length of the testing period, since, as discussed in **RQ1**, both the number of fixed bugs and the number of opened bugs depend on the length of the testing period.

To assess and compare the speed at which post-release bugs are fixed under traditional and rapid release models, we test the following null hypothesis:

H^2_{02}: *There is no significant difference between the distribution of Fix Time values for bugs related to TR versions and bugs related to RR versions.*

We also investigate a similar hypothesis for high priority bugs only. Because high priority bugs are likely to impede or prevent the use of core functionalities, we expect that they will be fixed with the same timely manner under traditional and rapid release models.

For this, we classify all the bugs based on their priority, *i.e.*, for each bug, we extract priority and severity values from the corresponding bug report. Since only 5% of Mozilla bugs from our data set are filed with priority values, we rely on the severity value of a bug report if the priority value is absent. Severity values are always available in the bug reports from our data set. In our analysis, we consider a bug to have a high priority if the bug was filed explicitly with a high priority value or if the bug's severity level is either "critical", "major", or "blocker". We used this heuristic before, with good results [15]. We can then test the following null hypothesis:

H^2_{03}: *There is no significant difference between the distribution of Fix Time values for high-priority bugs related to TR versions and high-priority bugs related to RR versions.*

Similar to **RQ1**, hypotheses H^2_{01}, H^2_{02} and H^2_{03} are two-tailed. We perform a Wilcoxon rank sum test to accept or refute them.

Findings. **When following a rapid release model, the proportion of bugs fixed during the testing period is lower than the proportion of bugs fixed in the testing period under the traditional release model.** Figure 6 shows the distribution of the proportion of bugs fixed during the testing period of TR and RR versions. We can observe that the proportion of bugs fixed is higher under the traditional release model. The Wilcoxon rank sum test returned a significant $p-value$ of 0.003. Therefore, we reject H_{01}^2.

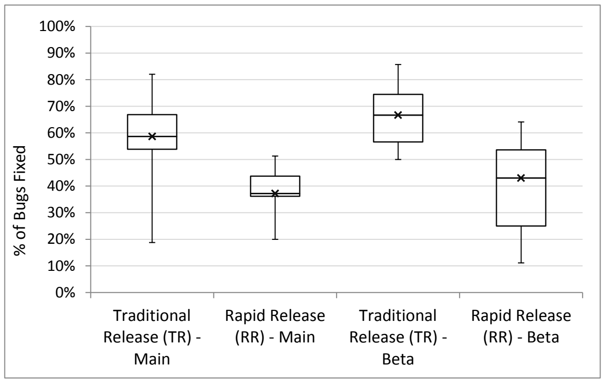

Figure 6. Boxplot of the proportion of bugs fixed.

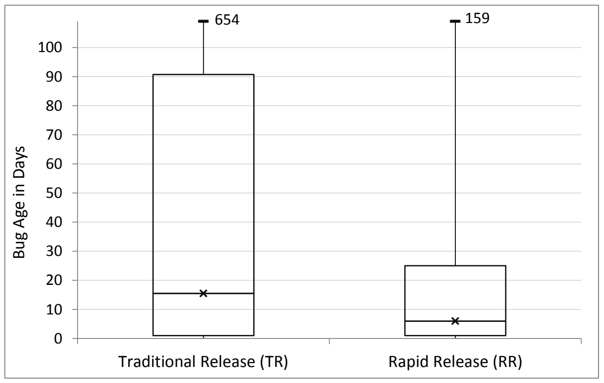

Figure 7. Boxplot of Bug Fixing Time.

Bugs are fixed faster under a rapid release model. Figure 7 shows the distributions of the bug fixing time for TR and RR versions, respectively. We can see that developers take almost three times longer to fix bugs under the traditional release cycle. The medians of bug fixing times under traditional release and rapid release models are respectively 16 days and 6 days. The result of the Wilcoxon rank sum test shows that the observed difference is statistically significant ($p-value = 5.22e-08$). Therefore, we reject H_{02}^2.

When limiting our comparison to high priority bugs, we obtain again a statistically significant difference, with a smaller $p-value(< 2.2e-16)$. Hence, we can also reject H_{03}^2.

In order to see if the observed difference in the bug fixing time and the proportion of bugs fixed is caused by source code size or complexity, we compute the following source code metrics on TR and RR versions. We compute the metrics on all files contained in a version.

- **Total Lines of Code:** the total number of lines of code of all files contained in a version.
- **Average Complexity:** the average of the McCabe Cyclomatic Complexity of all files contained in a version. The McCabe Cyclomatic Complexity of a file is the count of the number of linearly independent paths through the source code contained in the file.
- **Development Time:** the duration in days of the development phase of a version.
- **Rate of New Code:** the total number of new lines of code added in the version divided by the Development Time.

We found no significant difference between the complexity of traditional release and rapid release versions. Also, the rate of new code in major RR versions is similar to the rate of new code in minor TR versions. This finding is consistent with our other finding that the development time of major RR versions is similar to the development time of minor TR.

In summary, we found that although bugs are fixed faster during a shorter release cycle, a smaller proportion of bugs is fixed compared to the traditional release model, which allows a longer testing period. We analyzed the bugs reported during the testing period (*i.e.*, excluding post-release bugs), and found that, when testing under a rapid release model, bugs are reported at a slightly higher rate compared to the traditional model, *i.e.*, the project gets more feedback. The average (respectively median) number of bugs reported when testing under a rapid release model is 10.7 bugs (respectively 1.8 bugs), while the average (respectively median) number of bugs reported when testing under the traditional release model is 2.6 bugs (respectively 1.6 bugs). Similar to other projects [16], Firefox seems to experience a flood of user feedback that, given the limited length of the release cycle, cannot be triaged and fixed in timely fashion.

The Firefox rapid release model fixes bugs faster than using the traditional model, but fixes proportionally less bugs.

C. RQ3: Does the length of the release cycle affect software updates?

Motivation. One of the main arguments of advocates of rapid release models is the possibility to speed up the delivery of brand new features to users in order to keep them updated with the latest features as soon as development is completed. However, to achieve this goal, it is important that users quickly update to the new release. A fast adoption of new versions is also very critical for quality improvement. Users need to adopt new versions quickly in order to test the fixes and allow the maintenance team to discover new bugs to work on instead of spending developers' time testing buggy features reported already by others on earlier channels. In this research question, we investigate how fast new features and bug fixes reach users of TR and RR versions.

Approach. For each version v_i of Firefox in our data set, we compute the following metric:

Table II
STALENESS OF FIREFOX VERSIONS.

	Version released	# Weeks before staleness below 70%	60%	50%
Traditional release model	3.6	6	7	9
	4	6	7	7
Rapid release model	5	7	7	8
	6	4	4	4
	7	3	3	5
	8	2	2	3

- **Staleness**: the number of days the version v_i is still in use after a newer version v_{i+1} has been released.

This metric is inspired by the work of Baysal *et al.* [4]. However, instead of relying on the logs of a set of web servers to compute staleness values, we use the collected user crash reports. We compute the staleness of a Firefox version v_i by calculating the time period between the release date of a new version v_{i+1} and the disappearance of crash reports related to the version v_i from the field. Since full disappearance of a version is hard to achieve, we measure the time before the proportion of crashes from the users of the version v_i falls below respectively 70%, 60%, and 50%, of the total number of reported crashes per day. Staleness basically captures the speed with which the users are moving to new versions.

Findings. **Users switch faster to a newer rapid release version than to a new traditional release version**. Table II shows low staleness for RR versions 6, 7 and 8. Four weeks after the release of Firefox 7, 50% of the collected crash reports were already coming from the newly released version. In contrast, for Firefox 4 (developed according to the traditional model), it took 9 weeks before the mark of 50% of reported crash reports was reached.

With a rapid release model, users adopt new versions faster compared to the traditional release model.

Curiously, as shown on Figure 8, almost 20% of the users remain on a stalled version for a very long time. According to Firefox specialist Mike Kaply [17], a large share of these users are companies. In fact, many companies have remained on older versions of Firefox (*e.g.*, 3.6) because the rapid release schedule does not give them enough time to stabilize their platforms [9] and customer support costs are increasing because of the frequent upgrades [18].

To address this issue, Mozilla has initiated parallel versions of Firefox for companies. These versions are released at a slower schedule [19]. For other users, Mozilla has adopted silent updates for minor fixes of Firefox and it plans to integrate a complete silent update feature in the upcoming releases of Firefox, to keep all users on the latest versions.

V. THREATS TO VALIDITY

We now discuss the threats to validity of our study following common guidelines for empirical studies [20].

Construct validity threats concern the relation between theory and observation. In this work, these threats are mainly due to measurement errors. We compute source code metrics using the open source tool SourceMonitor. We extract crash and bug information by parsing their corresponding HTML (crash reports) and XML (bug reports) files. We use the occurrence of crashes to capture stalled usages of a version. The proportional drop of a version's crash count indicates that users have upgraded to a newer version. However, as the new version might have substantially more (or less) crash reports, the old version might drop faster (or slower) to for example 50%. In our data set, we have found no significant differences between the median daily crash count of the different versions.

Threats to internal validity concern our selection of subject systems, tools, and analysis method. Although we selected Firefox to control for development process and other changes before and after the migration to a rapid release cycle, some of the findings might still be specific to Firefox's development process.

Conclusion validity threats concern the relation between the treatment and the outcome. We paid attention not to violate assumptions of the constructed statistical models.

Reliability validity threats concern the possibility of replicating this study. We attempt to provide all the necessary details to replicate our study. The Mercurial repository of Firefox is publicly available to obtain the source code of Mozilla Firefox. Both the Socorro crash repository and Bugzilla repository are also publicly available. SourceMonitor is an open source code measurement tool [21].

Threats to external validity concern the possibility to generalize our results. Although this study is limited to Mozilla Firefox, our results on the time it takes users to adopt a new version are consistent with the findings of previous studies on Google Chrome, which has been following a rapid release model for a much longer time [4]. Nevertheless, further studies on different systems are desirable. Also, we only studied bug reports that were linked to crashes. Further studies on all bug reports are needed.

VI. RELATED WORK

To the best of our knowledge, this study is the first attempt to empirically quantify the impact of release cycle time on software quality in a controlled setting.

Since open source projects have been using agile methods for a long time, many projects adopted short release cycles. Ten years ago, Zhao et al. found that 54% of the open source apps released at least once per month. Five years later, Otte et al. [22] found slightly contrasting numbers (on a different set of apps), *i.e.*, 49.3% released at least once per 3 months. Although this is still relatively rapid, it is not clear why this shift has happened. In any case, modern commercial software projects [23], [24] and open source projects backed by a company [8], [25] have embraced shorter release cycles.

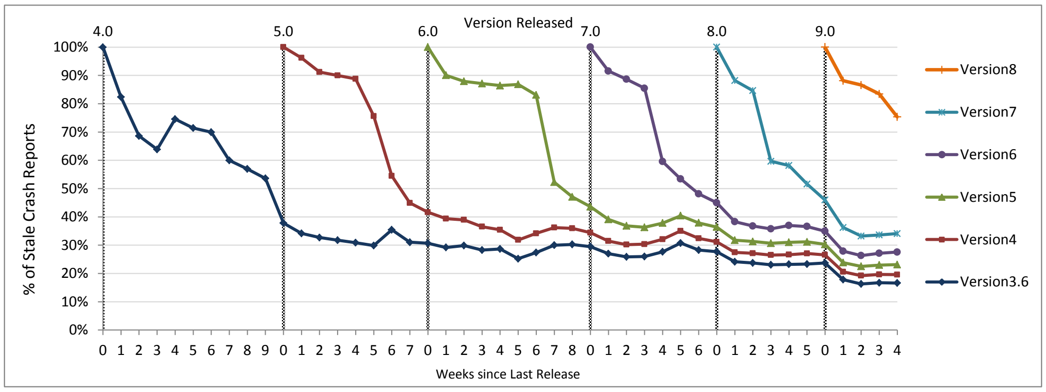

Figure 8. Staleness of major versions of FireFox.

A lot of work has focused on enabling consistent, short release cycles. For example, van der Storm [26] and Dolstra et al. [27] developed infrastructure to automatically build, package and deploy individual code changes. Mainstream continuous integration servers [28] automatically run sets of unit, integration or even acceptance tests after each check-in, while more advanced environments are able to run such tests in a massively parallel way in the shorter time in between releases [5]. The combination of these ideas and practices have led to the concept of continuous delivery [29], which uses highly automated infrastructure to deploy new releases in record time. Amazon, for example, deploys on average every 11.6 seconds [24], achieving more than 1,000 deployments per hour.

Despite all this work on achieving and pushing for shorter release cycles, there is hardly any empirical evidence that it really improves product quality, except for various developer surveys [30], [31]. Escrow.com reduced its release cycle to iterations of 2 weeks [32], resulting in a reduction of the number of defects by 70%. However, since many agile techniques and team restructurings were introduced at once, this improvement in quality cannot be related to shorter release cycles alone. Marschall [16] found that short release cycles require a steady flow of releases in order to control the number of reported bugs.

Kuppuswami et al. [33] built a simulation model to analyze the effects of each XP practice on development effort. Small, incremental releases reduce the development effort needed by 2.67%, but no link with software quality was made. Stewart et al. [34] tried to relate code quality to release frequency, number of releases and the change of size across releases, but could not derive any conclusions.

Releasing too frequently not only decreases the time to run tests, but it also might make customers weary of yet another update to install [5], [35]. For this reason, many projects do not automatically distribute each release to their customers. For example, although the Eclipse project uses 6-week release cycles, the resulting milestone releases are only available to interested users and developers [25], similar to how the NIGHTLY, AURORA and BETA Firefox channels are only targeted at specific groups of users. Clear communication about each channel/release is necessary to make sure that the intended user group deploys the new release and provides feedback about it [25], [35].

The work that is most closely related to ours is that of Baysal et al. [4]. It compares the release and bug fix strategies of Mozilla Firefox and Google Chrome based on browser usage data from web logs. At that time, Firefox was still in the 3.x series, *i.e.*, before its transition to a shorter release cycle, whereas Chrome had been following a short release cycle since its birth. Although the different profiles of both systems made it hard to compare things, the median time to fix a bug in the TR system (Firefox) seemed to be 16 days faster than in the RR system (Chrome), but this difference was not significant. We found the opposite, *i.e.*, Firefox RR fixes bugs faster than Firefox TR. However, the findings about staleness in the TR system confirm our findings.

Our paper eliminates the inconsistency between the two compared systems, by focusing on one project (Firefox) that only modified its release cycle, but otherwise remained the same. We believe that this allows to make more accurate claims regarding RR versus TR models. Furthermore, we use actual field crash data to assess the quality perceived by customers.

VII. Conclusion

The increased competitiveness of today's business environment has prompted many companies to adopt shorter release cycles, yet the impact of this adoption on software quality has not been established thus far. In this paper, we analyze the evolution of Mozilla Firefox during the period in which it shifted from a traditional release model to a rapid release model. We find that similar amounts of crashes occur, yet users seem to experience crashes earlier during run-time. Furthermore, bugs are fixed faster under rapid release models, but proportionally less bugs are fixed compared to the traditional release model. This could not be explained by differences in the complexity of the source code developed under both models. However, we found indications that the migration to a shorter release cycle could have triggered too many crash reports at once, flooding the triagers and bug fixers. Finally, as expected, users of a software system developed following a rapid release model tend to adopt new versions faster compared to the traditional release model.

Although more case studies are needed, our results provide some warnings for decision makers in software organizations that should be taken into account when changing the release cycle of their software systems.

References

[1] "Shorten release cycles by bringing developers to application lifecycle management," *HP Applications Handbook, Retrieved on Febuary 08, 2012*. [Online]. Available: http://bit.ly/x5PdXl

[2] "Mozilla puts out firefox 5.0 web browser which carries over 1,000 improvements in just about 3 months of development," *InvestmentWatch on June 25th, 2011. Retrieved on January 12, 2012*. [Online]. Available: http://bit.ly/aecRrL

[3] K. Beck and C. Andres, *Extreme Programming Explained: Embrace Change (2nd Edition)*. Addison-Wesley, 2004.

[4] O. Baysal, I. Davis, and M. W. Godfrey, "A tale of two browsers," in *Proc. of the 8th Working Conf. on Mining Software Repositories (MSR)*, 2011, pp. 238–241.

[5] A. Porter, C. Yilmaz, A. M. Memon, A. S. Krishna, D. C. Schmidt, and A. Gokhale, "Techniques and processes for improving the quality and performance of open-source software," *Software Process: Improvement and Practice*, vol. 11, no. 2, pp. 163–176, 2006.

[6] T. Downer, "Some clarification and musings," *Accessed on January 06, 2012*. [Online]. Available: http://bit.ly/q8RCuw

[7] "Web browsers (global marketshare)," *Roxr Software Ltd. Retrieved on January 12, 2012*. [Online]. Available: http://bit.ly/81klgi

[8] S. Shankland, "Google ethos speeds up chrome release cycle," http://cnet.co/wlS24U, July 2010.

[9] ——, "Rapid-release firefox meets corporate backlash," http://cnet.co/ktBsUU, June 2011.

[10] "New channels for firefox rapid releases," *The Mozilla Blog. 2011-04-13. Retrieved on January 12, 2012*. [Online]. Available: http://bit.ly/hc1zmY

[11] R. Paul, "Mozilla outlines 16-week firefox development cycle," 2011. [Online]. Available: http://bit.ly/fLHEfo

[12] "Socorro: Mozilla's crash reporting system," *Accessed on March 29, 2011*. [Online]. Available: http://bit.ly/9A9zKP

[13] I. Herraiz, E. Shihab, T. H. D. Nguyen, and A. E. Hassan, "Impact of installation counts on perceived quality: A case study on debian." in *Proc. of the 18th Working Conf. on Reverse Engineering (WCRE)*, 2011, pp. 219–228.

[14] M. Hollander and D. A. Wolfe, *Nonparametric Statistical Methods*, 2nd ed. John Wiley and Sons, inc., 1999.

[15] F. Khomh, B. Chan, Y. Zou, and A. E. Hassan, "An entropy evaluation approach for triaging field crashes: A case study of mozilla firefox," in *Proc. of the 18th Working Conf. on Reverse Engineering (WCRE)*, 2011.

[16] M. Marschall, "Transforming a six month release cycle to continuous flow," in *Proc. of the conf. on AGILE*, 2007, pp. 395–400.

[17] "Understanding the corporate impact," *Retrieved on January 12, 2012*. [Online]. Available: http://bit.ly/mBzP37

[18] "Why do companies stay on old technology?" *Retrieved on January 12, 2012*. [Online]. Available: http://bit.ly/k3fruK

[19] "Mozilla proposes not-so-rapid-release firefox," *CNET, Retrieved on February 08, 2012*. [Online]. Available: http://cnet.co/mQZ6Tf

[20] R. K. Yin, *Case Study Research: Design and Methods - Third Edition*, 3rd ed. SAGE Publications, 2002.

[21] "Sourcemonitor," *Accessed on January 12, 2012*. [Online]. Available: http://bit.ly/9AKzN8

[22] T. Otte, R. Moreton, and H. D. Knoell, "Applied quality assurance methods under the open source development model," in *Proc. of the 32nd Annual IEEE Intl. Computer Software and Applications Conf. (COMPSAC)*, 2008, pp. 1247–1252.

[23] A. W. Brown, "A case study in agile-at-scale delivery," in *Proc. of the 12th Intl. Conf. on Agile Processes in Software Engineering and Extreme Programming (XP)*, vol. 77, May 2011, pp. 266–281.

[24] J. Jenkins, "Velocity culture (the unmet challenge in ops)," Presentation at O'Reilly Velocity Conference, June 2011.

[25] E. Gamma, "Agile, open source, distributed, and on-time – inside the eclipse development process," Keynote at the 27th Intl. Conf. on Software Engineering (ICSE), May 2005.

[26] T. van der Storm, "Continuous release and upgrade of component-based software," in *Proc. of the 12th intl. wrksh. on Softw. configuration management (SCM)*, 2005, pp. 43–57.

[27] E. Dolstra, M. de Jonge, and E. Visser, "Nix: A safe and policy-free system for software deployment," in *Proc. of the 18th USENIX conf. on System admin.*, 2004, pp. 79–92.

[28] P. Duvall, S. M. Matyas, and A. Glover, *Continuous Integration: Improving Software Quality and Reducing Risk*. Addison-Wesley Professional, 2007.

[29] J. Humble and D. Farley, *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation*, 1st ed. Addison-Wesley Professional, 2010.

[30] S. Kong, J. E. Kendall, and K. E. Kendall, "The challenge of improving software quality: Developers' beliefs about the contribution of agile practices," in *Proc. of the Americas Conf. on Information Systems (AMCIS)*, August 2009, p. 12p.

[31] VersionOne, "4th annual state of agile survey," http://bit.ly/6BPw5, 2009.

[32] P. Hodgetts and D. Phillips, *eXtreme Adoption eXperiences of a B2B Start Up*. Addison-Wesley Longman Publishing Co., Inc., 2002, ch. 30, extreme Programming Perspectives.

[33] S. Kuppuswami, K. Vivekanandan, P. Ramaswamy, and P. Rodrigues, "The effects of individual xp practices on software development effort," *SIGSOFT Softw. Eng. Notes*, vol. 28, p. 6p., November 2003.

[34] K. J. Stewart, D. P. Darcy, and S. L. Daniel, "Observations on patterns of development in open source software projects," *SIGSOFT Softw. Eng. Notes*, vol. 30, pp. 1–5, May 2005.

[35] S. Jansen and S. Brinkkemper, "Ten misconceptions about product software release management explained using update cost/value functions," in *Proc. of the Intl. Workshop on Software Product Management*, 2006, pp. 44–50.