Release Cycles And Software Quality

Vivek Gopalakrishnan
Vgopala2@ncsu.edu

Muthu Arvind Lakshmanan
mlakshm@ncsu.edu

ABSTRACT

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. The release frequency of software projects has increased in recent years. Adopters of so-called rapid release cycles claim that they can deliver addressed issues (i.e., bugs, enhancements, and new features) to users more quickly. Although gut feeling is important in such decisions, it's increasingly important to leverage existing data, such as bug reports, source code changes, code reviews, and test results, both to support decisions and to help evaluate current practices. The exploration of software engineering data to obtain insightful information is the need for the day. In this paper, we have studied the various papers in software engineering that have empirically explored the impact of release cycle lengths on software quality in terms of metrics such as post-release bugs. The dataset used by all papers in the study have been predominantly open source data. Specifically, most papers have used Mozilla Firefox's data. We explore what the papers have said and provide improvements and suggestions for each paper.

Keywords
Rapid Release Cycle, Traditional Release Cycle, Post Release bugs, Software Quality

1. INTRODUCTION

With software products being increasingly used today more than ever, it has become necessary for organizations to constantly improve the quality of their software in a fast manner. User behavior suggests that faster resolution of bugs and addition of features to consumer facing software products is important to grow and retain the user base. And so an increasing number of software companies are moving to faster release cycles. The research papers we explored covered a variety of aspects of release models that could affect software right from software quality in terms of number of bugs and software capability in terms of number of features added.

In this study, we aim to compare the effects of long traditional cycles on software with that of rapid release cycles.

Tradition release cycles follow a sequence of stages of development ranging from requirement planning to eventual release. The priority in traditional models is to fix bugs and add features that were decided during the requirement planning phase. Usually, release is deferred until all features have been developed and thoroughly tested.

Traditional release cycles involve two stages of testing alpha and beta. In the alpha phase, developers usually test their software using white box techniques. Additional black box testing is done by another testing team. In the beta phase, a subset of users are allowed to perform usability testing of the software. This usually happens when the feature is complete but is likely to contain an unknown number of bugs. The goal of the development team is to uncover as many bugs as possible in this phase, fix them and be ready with a release candidate. A *release candidate* is a beta version with potential to be a final product, which is ready to release unless significant_bugs emerge. In this stage of product stabilization, all product features have been designed, coded and tested through one or more beta cycles with no known showstopper-class bugs.

Rapid release cycles usually follow the same sequence of stages of development as the traditional release cycle, except that the focus here is to quickly build and test features or fix bugs within a short but fixed period of time rather than wait till a feature is complete. Incomplete features are released in subsequent release cycles. More companies are moving to faster release cycles. Companies get faster feedback about bugs and how well users have responded to a new feature. Besides this, releases also become slightly easier to plan as they span a very short period of time. Developers are not rushed to release half baked features and are encouraged to focus on quality assurance. Most importantly, the higher number of releases can benefit both the consumer and the organization as it will lead to faster access to relevant features, bug fixes and security updates..

2. MOTIVATION

Release methodology is an interesting area of research for software scientists. Especially since enough data has emerged from both types of release models. Our aim was to conduct a survey of literature in this field, while hoping to encounter a variety of analysis performed on data to uncover insights produced both intuitive and unintuitive results[2].

3. RELATED WORK

A. Impact Of Faster Releases On Software Quality

In the first paper we read[1], Khomh et al. explored the impact of faster releases on various metrics that software quality.

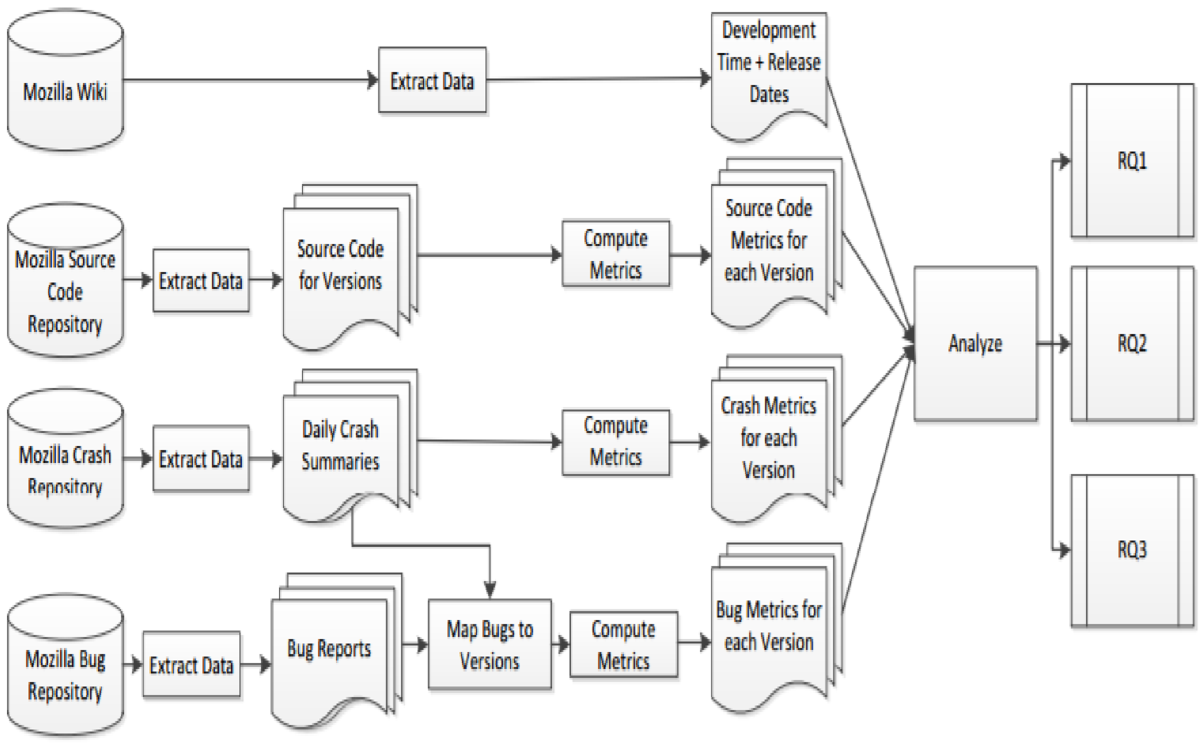


Figure 1 – Overview of the approach to study the impact of release cycle time on software quality

Specifically, they studied the following three research questions:

Q1) Does the length of the release cycle affect the quality in terms of number of post release bugs?

Q2) Does the length of the release cycle affect the fixing of bugs?

Q3) Does the length of the release cycle affect software updates?

To answer these questions the authors designed their study around the Mozilla Firefox dataset. It contained 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 rapid release model. A traditional release had an average cycle time of 52 weeks between every major release and 4 weeks between every minor release. For each version in the data(both traditional and rapid releases) the authors extracted the necessary data from the source code repository, the crash repository(socorro) and the bug repository(bugzilla).

The following metrics were obtained in the data processing phase.

1) *Development Time* - The development time of the version is computed by calculating the difference between the release date and the starting date of the development phase. It was found that the development time was slightly longer than the release cycle. This was because the development of a new version is usually started before the release of the previous one.

2) *Code Metrics* - Metrics such as average complexity, lines of code and branch coverage were computed using the source code measure tool, Source monitor.

3) *Crash and Bug Metrics* - From the mozilla crash repository, the authors extracted the date of the crash, the version of Firefox that was running during the crash, the list of related bugs, the duration in seconds for which Firefox was running before it crashed(uptime) and the bug status(e.g. UNCONFIMRED, FIXED).

Using these metrics the authors proceeded to answer the three research questions. The first research question was answered in three steps. First, the authors compared the number of post release bugs between traditional and rapid release versions. However the authors note that this comparison cannot be done directly as the it been shown by Herraiz et al. [11] that the number of reported post-release bugs is proportional to the number of deployments. Therefore the number of post-release bugs was divided by the length of the release cycle. It was found out that

1) There is no significant difference between the number of post-release bugs of Rapid Release versions and Traditional Release versions.

2) There is no significant difference between the median daily crash count of RR versions and TR versions.

3) There is no significant difference between the median uptime values of RR versions and TR versions.

It was concluded that were controlled for the length of the release cycle of the version, there is no significant difference between the number of post-release bugs of traditional and rapid release models.

To answer the second question, the authors' approach was to compute the number of fixed bugs, the number of unconfirmed bugs and the fix time(the difference between the bug open time and the last modification time).

By studying the metrics it was concluded that when following a rapid release model, the proportion of bugs fixed during the testing period was lower than the proportion of bugs fixed in the testing period under the traditional release model. However, it was found out that bugs were fixed under a rapid release model.

For the third research question, the authors approached the problem by computing staleness(the number of days a version is still in use after a newer version has been released[2]. It was found that users switched to a rapid release version than to a new traditional release version.

B A Tale of Two Browsers

In the second paper we read, Baysal et al.[2] compare two browsers, Google Chrome and Mozilla Firefox, using information about their respective release histories and usage patterns. A comparative analysis of the systems was done by mining their bug repositories and web traffic data to determine the trends in user acceptance and adoption of a browser that could help to explain factors behind the popularity of an open source browsers. Similar to the first paper[1], the authors of this paper approached the study by aiming to answer a few research questions that concern release frequency, defect rates, time to fix bugs, user update likelihood and market share. Also, this study was conducted before Mozilla switched to the rapid release model and therefore, all numbers concerning firefox relate to the traditional release cycle in which they released a new version every 10 months. On the other hand, Google chrome used a relatively faster rapid release model in which they released a new version every 2.5 months. Among other questions, the following important questions were addressed.

Q1) Which browser is more defect prone?

Based on the data that was collected the authors concluded that the ratio of number of important bugs to the total number of bugs per release is nearly stable for both Firefox and Chrome. By analyzing the bug history the authors plotted(see Figure) the growth rate of defects of both browsers over time. The total number of bugs to be fixed during a Firefox release lifespan contained on an average about 20% high severity bugs, while Chrome had about 10% high severity bugs

Q2) How quickly are the bugs fixed?

The median and average times of a bug fix for both the systems were calculated by comparing open and close dates from the bug report. It was assumed that the time to fix higher importance bugs is a better measure of responsiveness tan lower importance bugs and in this regard, it was found that Chrome developers are faster in fixing bugs, particularly the high priority ones.

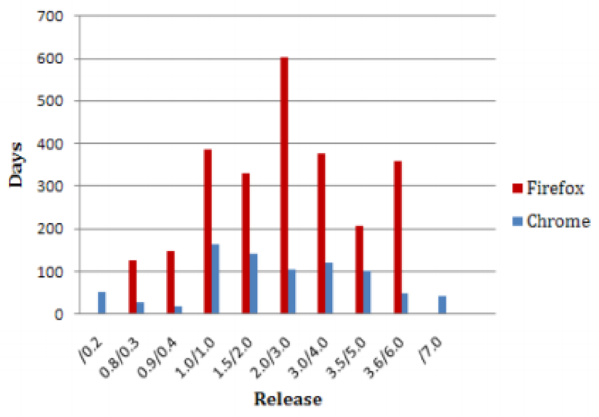


Figure 2: Lifespan of major releases for firefox and chrome

Q3) How stale is your browser?

Surprisingly, Firefox, even its traditional release process in which it released a more stable version of the product, is commonly used as stale versions sometimes years after its initial release. The results suggest that Chrome users enthusiastically upgrade their browser as soon as a new version is released but Firefox users do not.

Q4) Does the volume of defects affect the popularity of a browser?

To address this question the authors performed a linear correlation analysis, which can be defined as the measure of dependence between two random variables, between the number of bugs and the number of page visits. It was found that while firefox had a negative correlation with the number of defects, Chrome had a moderate positive correlation.

C QA Methods On Open Source Projects

The third paper we read[3](by Otte et al.) did not talk about the type of release cycle. Despite this, we chose to study the paper because it had information about various QA methods concerning bug detection in open source projects including Mozilla Firefox. The authors used a survey research method to gain empirical evidence about applied QA practices under the OSSD model. Furthermore the paper also analysed successful projects in order to find common patterns, which distinguish these projects and indicates key processes that contribute to their success. Similar to the papers we have seen so far, the authors of this paper designed their study around research questions.

Q1) How is Quality Assurance (QA) under the OSSD model in mid- to large sized OSS projects conducted?

Q2) what key practices do we learn from successful approaches?

Recent studies [9][20] show that among open source projects user participation is extremely high, defect handling processes follow mainly structured approaches, testing takes a significant portion of the software life cycle and there is a high usage of configuration and bug tracking tools. However project documentation was often rare and design documents were lacking. To provide an empirical view of QA practices, the authors of this paper conducted a survey combining a quality characteristics with process success measures to identify key practices in OSSD projects. The target group of the survey were individual developers who took part in such projects and they had to answer some open ended questions.

Among other insights that were gained in the survey, it was found out that the average testing time compared to the whole development time was 38.6% and more than half of the projects show a structural testing approach. Large projects have strict quality checks, as 75% of them report rework or even reject inappropriate code.

The authors concluded that their investigation into more mature projects provided important insight into applied practices and may have indicated reasons for the success of the OSSD model. These projects considered modularity of code already during design. Quality control activities before code commit have a higher importance. More time is spent on testing and testing approach is better structured. These projects efficiently leverage their community, benefiting from efficient user testing. Internal communication is rated as remarkable, which contributes positively to these processes. Defect handling processes seems better structured and include source code defects, requirements and documentation issues.

Finally the authors concluded that the relations to project success criteria indicated a correlation between quality and succession but no causalty and that further research was required to explore QA practices and determine their relation to project success.

D Improving Quality and Performance of Open Source Software

In paper 4[4], Yilmax et al. offer their contributions to quality oriented open source software systems through their distributed continuous quality assurance techniques. This paper like the previous one touches upon the quality aspects of open source systems.The authors approached their research study by identifying problems with current open source processes and them through their DCQA system. It consisted of the following 5 problems

Problem 1 - Hard to maintain software quality in the face of short development cycles.

Problem 2: Lack of global view of system constraints.

Problem 3: Unsystematic and redundant QA activities.

Problem 4: Lack of diversity in test environments.

Problem 5: Manually intensive execution of QA processes

The above problems were addressed by building Skoll, who's DCQA process are (1) distributed, i.e., a given QA task is divided into several subtasks that can be performed on a single user machine, (2) opportunistic, i.e., when a user machine becomes available, one or more subtasks are allocated to it and the results are collected and fused together at central collection sites to complete the overall QA process, and (3) adaptive, i.e., earlier subtask results are used to schedule and coordinate future subtask allocation. Skoll leverages important open-source project assets, such as the technological sophistication and extensive computing resources of worldwide user communities, open access to source, and ubiquitous web access, to improve the quality and performance of open-source software significantly by automating the division of labor.

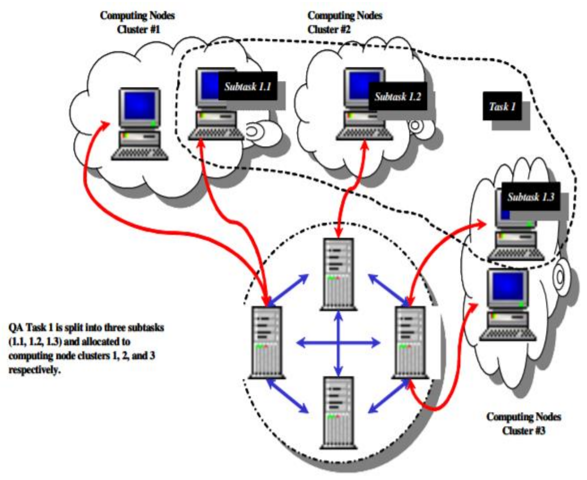

Computing Nodes Cluster #1
Computing Nodes Cluster #2
Subtask 1.1
Subtask 1.2
Task 1
Subtask 1.3
QA Task 1 is split into three subtasks (1.1, 1.2, 1.3) and allocated to computing node clusters 1, 2, and 3 respectively.
Computing Nodes Cluster #3

Figure 3: Skoll Tasks/Subtasks Allocated to Computing Nodes

E Rapid Releases and Software Testing

The fifth paper we read[5], talked about the explores the degree of investigation that is possible in a rapid release environment. The authors used the same dataset as the authors of the first paper, but the scope of their study varies. Apart, from the dataset they also got feedback from a Mozilla QA engineer. Their research builds upon the study that found that enterprises following rapid release model currently lack time to stabilize their performance. Porter et al. noted that, since there is less time available, testers have less time to test all possible configurations of a released product, which can have a negative effect on software quality[4]. Based on the feedback from the QA engineer, the authors formulated the following questions and set out to study them.

Q1) Do RRs affect the amount of testing performed?

Q2) Do RRs affect the number of testers working on a project?

Q3) Do RRs affect the frequency of testing activity?

Q4) Do RRs affect the number of configurations being tested?

To answer the first question the authors calculated the amount of tests executed and the functional coverage of those tests for each alpha ,beta, release-candidate, major and minor Firefox versions. Functional coverage is the degree to which different features of a software are tested by the test suite. The authors found that the RR model executes almost twice as many tests per day (median) compared to TR models. Also RR models had functional coverages similar to TR, but had lower coverage overall.

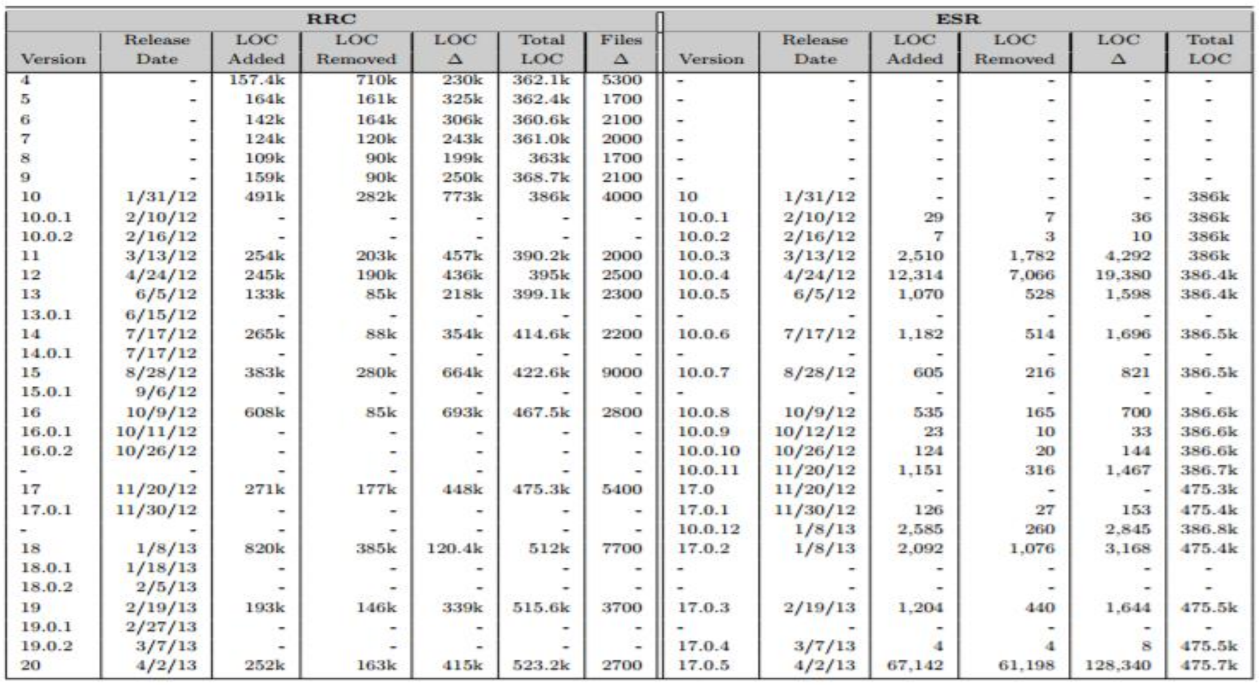
For the second question the authors calculated the number of testers per day for each version of the release. TR had a median of 1.67 testers per day, while RR had 1 tester per day. Therefore it was concluded that fewer testers conduct testing for RR releases.

For the third question, the authors calculated the number of tested builds per day and the number of commits to the mercurial repository per day. This was computed for all versions of the release. It was concluded that less number of rapid builds are tested per day and that RR build contains more code commits than TR builds. To understand this, the authors analyzed whether these builds contain more commits relative to TR builds.

For the final question, the authors calculated the number of f tested locales per day and the number of operating systems tested per day. To test the null hypothesis they used the Wilcoxon rank-sum test[19] using a 1% confidence level. It was concluded that Rapid Release tests are conducted on only one locale manually and that a slightly lower number of platforms are being tested, but more thoroughly. However, the total number of tested operating systems has dropped slightly, with most of the RR releases testing 9 operating systems compared to 12 to 17 for TR releases.
It must be noted that each of the above conclusions was evaluated using the feedback from the Mozilla QA Engineer.

F. Security and Rapid-Release in Firefox

In the sixth paper we read[6], Clark et al. correlate reported vulnerabilities in multiple rapid release versions of Firefox code against those in corresponding extended release versions of the same system; using a common software base with different release cycles eliminates many causes other than RR for the observables. As with other papers in the study, the authors of



RRC							ESR					
Version	Release Date	LOC Added	LOC Removed	LOC Δ	Total LOC	Files Δ	Version	Release Date	LOC Added	LOC Removed	LOC Δ	Total LOC
4	-	157.4k	710k	230k	362.1k	5300	-	-	-	-	-	-
5	-	164k	161k	325k	362.4k	1700	-	-	-	-	-	-
6	-	142k	164k	306k	360.6k	2100	-	-	-	-	-	-
7	-	124k	120k	243k	361.0k	2000	-	-	-	-	-	-
8	-	109k	90k	199k	363k	1700	-	-	-	-	-	-
9	-	159k	90k	250k	368.7k	2100	-	-	-	-	-	-
10	1/31/12	491k	282k	773k	386k	4000	10	1/31/12	-	-	-	386k
10.0.1	2/10/12	-	-	-	-	-	10.0.1	2/10/12	29	7	36	386k
10.0.2	2/16/12	-	-	-	-	-	10.0.2	2/16/12	7	3	10	386k
11	3/13/12	254k	203k	457k	390.2k	2000	10.0.3	3/13/12	2,510	1,782	4,292	386k
12	4/24/12	245k	190k	436k	395k	2500	10.0.4	4/24/12	12,314	7,066	19,380	386.4k
13	6/5/12	133k	85k	218k	399.1k	2300	10.0.5	6/5/12	1,070	528	1,598	386.4k
13.0.1	6/15/12	-	-	-	-	-	-	-	-	-	-	-
14	7/17/12	265k	88k	354k	414.6k	2200	10.0.6	7/17/12	1,182	514	1,696	386.5k
14.0.1	7/17/12	-	-	-	-	-	-	-	-	-	-	-
15	8/28/12	383k	280k	664k	422.6k	9000	10.0.7	8/28/12	605	216	821	386.5k
15.0.1	9/6/12	-	-	-	-	-	-	-	-	-	-	-
16	10/9/12	608k	85k	693k	467.5k	2800	10.0.8	10/9/12	535	165	700	386.6k
16.0.1	10/11/12	-	-	-	-	-	10.0.9	10/12/12	23	10	33	386.6k
16.0.2	10/26/12	-	-	-	-	-	10.0.10	10/26/12	124	20	144	386.6k
-	-	-	-	-	-	-	10.0.11	11/20/12	1,151	316	1,467	386.7k
17	11/20/12	271k	177k	448k	475.3k	5400	17.0	11/20/12	-	-	-	475.3k
17.0.1	11/30/12	-	-	-	-	-	17.0.1	11/30/12	126	27	153	475.4k
-	-	-	-	-	-	-	10.0.12	1/8/13	2,585	260	2,845	386.8k
18	1/8/13	820k	385k	120.4k	512k	7700	17.0.2	1/8/13	2,092	1,076	3,168	475.4k
18.0.1	1/18/13	-	-	-	-	-	-	-	-	-	-	-
18.0.2	2/5/13	-	-	-	-	-	-	-	-	-	-	-
19	2/19/13	193k	146k	339k	515.6k	3700	17.0.3	2/19/13	1,204	440	1,644	475.5k
19.0.1	2/27/13	-	-	-	-	-	-	-	-	-	-	-
19.0.2	3/7/13	-	-	-	-	-	17.0.4	3/7/13	4	4	8	475.5k
20	4/2/13	252k	163k	415k	523.2k	2700	17.0.5	4/2/13	67,142	61,198	128,340	475.7k

Table 1 : RRC changes from the previous version, and total LOC per version

paper designed their research process by addressing a few questions. Specifically they addressed the following questions,

1. Does a switch to Agile RR development introduce large numbers of new vulnerabilities into software, given that the focus is on new features, rather than on improving existing code?

Conclusion - Overall, the total number of active vulnerabilities disclosed per LOC in each Firefox version since the advent of rapid release mirrors the defect discovery. Similar to defects, there is no significant jump in the number of vulnerabilities disclosed. This means that, contrary to expectations, the large volume of code added does not appear to contain more than its share of vulnerabilities.

2. Where in the code base are vulnerabilities being discovered? (i.e., are they in code written prior to the switch to RR, in code introduced in previous iterations of RR or in code added in the current version?)

Conclusion - The overwhelming majority of active vulnerabilities disclosed in Firefox RR also affect TR. This does not mean that the new code in RR does not contain new vulnerabilities, but rather, that 90% of the vulnerabilities disclosed in the RR versions released during the lifetime of each ESR version must be in the older, shared code.

3. Are vulnerabilities being discovered more quickly since the switch to RR?

Conclusion - With traditional defect and vulnerability discovery models, the expectation is that the easy-to-find vulnerabilities in new code are found and patched quickly. Looking at traditional non-RR software, Clark, et al [18] suggested that these models do not accurately represent the early life cycle of vulnerability disclosure. Instead, there appears to be a relatively long period before the first vulnerability in new software is disclosed, after which the rate of vulnerability disclosure in that version of code increases. The authors speculated that this period corresponds to the attacker's learning curve.The authors have concluded that Firefox's RR strategy did not increase the rate of vulnerability discovery and disclosure and, that the vulnerabilities disclosed while a particular RR version was current were not easy to find. It does indeed appear that during the RR lifecycle, the time to find vulnerabilities and learn how to exploit them in new code compensates for the presumed increase in the density of vulnerabilities in immature code.

G. Rapid Releases and Patch Backouts

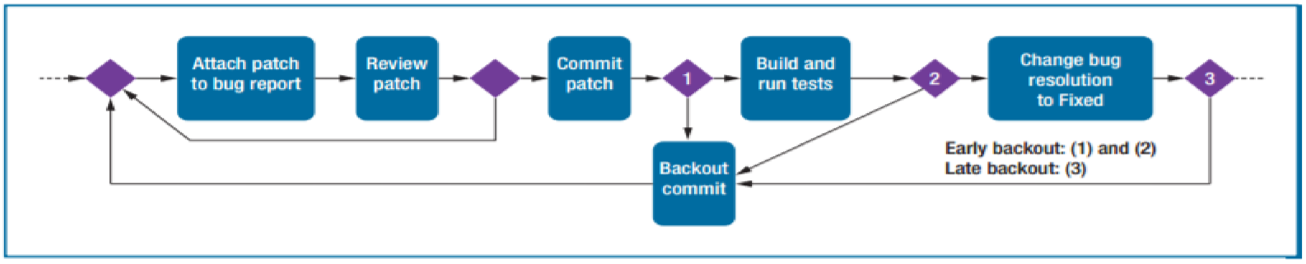
In the seventh paper we read[7], Rodrigo Souza talks about how often developers are backing out their changes. A backout is the process of reverting a patch because it broke the build. There are two types of backouts, early backout and late backout. If the backout occurs before changing the status of bug report, it is an early backout. Otherwise, it is a late backout.The authors hypothesized that developers are backing out the broken patches earlier making the release process more stable. Also, the study surveyed a few Mozilla engineers. In the study, a few Mozilla engineers pointed out that the backout rate might have been underestimated under traditional releases because the backout culture became more prevalent after the introduction of sheriff-managed integration repositories. Before that, developers usually fixed a broken commit by recommitting, without explicitly backing out the first commit

The conclusions of the authors can be summarized by the following points.

1) Some engineers explained the increase in the overall backout rate by suggesting that because the code base grew over time, code con icts became more likely. The number of supported platforms also increased because Firefox must support both new platforms and older ones.

2) The increasing early-backout rate and decreasing late-backout rate were due partly to the evolution of the automated testing toolset. According to Mozilla engineers, the emergence of better testing tools promoted earlier detection of problems and improved even detection of problems that would have otherwise gone unnoticed, such as hard-to-detect memory leaks.

3) The increasing early-backout rate was also due to the sheriff managed integration repositories and their effect on how developers test their code. Before 2011, because developers pushed changes directly to m-c, the changes had to be thoroughly tested to avoid breaking the builds or introducing bugs. From 2011 to 2013, developers committed to integration repositories, and the sheriff backed out problematic patches before merging changes to m-c, thus keeping it stable.


Attach patch to bug report
Review patch
Commit patch
1
Build and run tests
2
Change bug resolution to Fixed
3
Backout commit
Early backout: (1) and (2)
Late backout: (3)

4) Every backout induces rework by requiring development of a new, improved patch. However, in Mozilla's case, the increase of early backouts didn't seem to cause overhead. Instead, it re ected a cultural change toward committing patches before testing them comprehensively, therefore reducing the effort required to test patches.

H Impact of RR on Integration

In the eighth paper we read[8], the authors explored the impact that a shift from a traditional to a rapid release cycle has on the speed of integration of addressed issues. Such an investigation is important to empirically check if adopting a rapid release cycle really does lead to quicker delivery of addressed issues. The authors set out to comparatively study the integration delay of addressed issues in the traditional and rapid release of the Firefox system by addressing a few questions, a trait that we observed in almost every paper we studied so far. The research questions were

Q1: Are addressed issues integrated more quickly in rapid releases?

Q2: Why can traditional releases integrate addressed issues more quickly?

Q3: Does the change in release strategy have an impact on the characteristics of delayed issues?

The authors approached the first question by considering the last RESOLVED-FIXED status as the moment that a particular issue was addressed. They observed the lifetime of the issues of traditional and rapid releases and then looked at the time span the

Dimension	Attributes	Value	Definition (d)|Rationale (r)
Reporter	Experience	Numeric	**d:** the number of previously integrated issues that were reported by the reporter of a particular addressed issue.
			r: The greater the experience of the reporter the higher the quality of his reports and the solution to his/her reports might be integrated more quickly [28].
	Reporter integration	Numeric	**d:** The median in days of the previously integrated addressed issues that were reported by a particular reporter.
			r: If a particular reporter usually reports issues that are integrated quickly, his/her future reported issues might be integrated quickly as well.
Resolver	Experience	Numeric	**d:** the number of previously integrated addressed issues that were addressed by the resolver of a particular addressed issue. We consider the assignee of the issue to be the resolver of the issue.
			r: The greater the experience of the resolver, the greater the likelihood that his/her code will be integrated faster [28].
	Resolver integration	Numeric	**d:** The median in days of the previously integrated addressed issues that were addressed by a particular resolver.
			r: If a particular resolver usually address issues that are integrated quickly, his/her future addressed issues might be integrated quickly as well.
Issue	Stack trace attached	Boolean	**d:** We verify if the issue report has an stack trace attached in its description.
			r: A stack trace attached may provide useful information regarding the cause of the issue, which may quicken the integration of the addressed issue [27].
	Severity	Nominal	**d:** The severity level of the issue report. Issues with higher severity levels (*e.g.*, blocking) might be integrated faster than other issues.
			r: Panjer observed that the severity of an issue has a large effect on its time to be addressed in the Eclipse project [25].
	Priority	Nominal	**d:** The priority level of the issue report. Issues with higher severity levels (*e.g.*, P1) might be integrated faster than other issues.
			r: Higher priority issues will likely be integrated before lower priority issues.
	Description size	Numeric	**d:** The number of words in the description of the issue.
			r: Issues that are well described might be more easy to integrate than issues that are difficult to understand.
Project	Queue rank	Numeric	**d:** A rank number that represents the moment when an issue is addressed compared to other addressed issues in the backlog. For instance, in a backlog that contains 500 issues, the first addressed issue has rank 1, while the last addressed issue has rank 500
			r: An issue with a high *queue rank* is an recently addressed issue. An addressed issue might be integrated faster/slower depending of its rank.
	Cycle queue rank	Numeric	**d:** A rank number that represents the moment when an issue is addressed compared to other addressed issues of the same release cycle. For example, in a release cycle that contains 300 addressed issues, the first addressed issue has a rank of 1, while the last has a rank of 300.
			r: An issue with a high *cycle queue rank* is an recently addressed issue compared to the others of the same release cycle. An issue addressed close to the upcoming release might be integrated faster.
	Queue position	Numeric	**d:** $\frac{\text{queue rank}}{\text{all addressed issues}}$. The *queue rank* is divided by all the issues that are addressed by the end of the next release. A *queue position* close to 1 indicates that the issue was addressed recently compared to others in the backlog.
			r: An addressed issue might be integrated faster/slower depending of its position.
	Cycle queue position	Numeric	**d:** $\frac{\text{cycle queue rank}}{\text{addressed issues of the current cycle}}$. The *cycle queue rank* is divided by all of the addressed issues of the release cycle. A *cycle queue position* close to 1 indicates that the issue was addressed recently in the release cycle.
			r: An issue addressed close to a upcoming release might be integrated faster.
Process	Number of Impacted Files	Numeric	**d:** The number of files linked to an issue report.
			r: An integration delay might be related to a high number of impacted files because more effort would be required to properly integrate the modifications [17].
	Churn	Numeric	**d:** The sum of added lines plus the sum of deleted lines to address the issue.
			r: A higher churn suggests that a great amount of work was required to address the issue, and hence, verifying the impact of integrating the modifications may also be difficult [17, 24].
	Fix time	Numeric	**d:** Number of days between the date when the issue was triaged and the date that it was addressed [9].
			r: If an issue is addressed quickly, it may have a better chance to be integrated faster.
	Number of activities	Numeric	**d:** An activity is an entry in the issue's history.
			r: A high number of activities might indicate that much work was required to address the issue, which may impact the integration of the issue into a release [17].
	Number of comments	Numeric	**d:** The number of comments of an issue report.
			r: A large number of comments might indicate the importance of an issue or the difficulty to understand it [9], which might impact the integration delay [17].
	Interval of comments	Numeric	**d:** The sum of the time intervals (hour) between comments divided by the total number of comments of an issue report.
			r: A short *interval of comments* indicates that an intense discussion took place, which suggests that the issue is important. Hence, such issue may be integrated faster.
	Number of tosses	Numeric	**d:** The number of times that the assignee has changed.
			r: Changes in the issue assignee might indicate that more than one developer have worked on the issue. Such issues may be more difficult to integrate, since different expertise from different developers might be required [15, 17].

Table 2Metrics used in the explanatory models.

issue. The authors then used Cliff's delta, a non-parametric effect size measure to verify how often values in one distribution are larger than values in another distribution. The test showed that there is no significant difference between traditional and rapid releases regarding issue lifetime.

To tackle the second question the authors group traditional and rapid releases into major and minor releases and study their integration delay. It was concluded that Minor-traditional releases tend to have less integration delay than major/minor-rapid releases. When considering both minor and major releases, the

time span between traditional and rapid releases are roughly the same. It was observed that integration delay was shorter on an average in traditional release cycles.

To answer the third question the authors used an interesting approach. They build explanatory models such as logistic regression for the traditional and rapid releases data with the intent to explain why a given addressed issue has its integration delay. The authors followed the guidelines of Harrel Jr[16] to build the explanatory model. They assesed the fit of the models using ROC curve and Brier Score[17] and then evaluated the impact that each metric has on the models by using Wald chi-squared maximum likelihood tests. Having obtained Brier score of 0.05-0.16 and ROC areas of 0.81-0.83 the authors concluded that traditional releases prioritize the integration of backlog issues, while rapid releases prioritize the integration of issues of the current release cycle.

4. IMPROVEMENTS

In the first paper, we believed that the effect that a shorter release cycle produces on developer productivity can be checked. Although the authors have mentioned that developers are less pressured to rush half-baked features into the software repository to meet the deadline, they could add a source to back this claim. It has been observed that agile practices, which have shorter release cycles, are quite intense on developers. Also, a very important metric to track the quality of a software is code coverage. The authors could have efficiently observed the quality of unit tests written for both the models by comparing their code coverage numbers. Furthermore, The authors can check for number of features released in traditional release model compared to multiple rapid releases for the same time period. They could then use this information and observe the number of bugs with relation to the number of features released in both the models.

In the second paper, we felt that the authors could have categorized the bug based on their type(eg. Rendering bug, crash etc) and compare if similar bugs occur in both the browsers and also Compare the code size of both the browsers and check if they have any correlation with the number of bugs reported.

In the third paper, which was significantly different from the rest, the paper could have talked about what kind of survey questions turned about to be the most helpful for their research. Also, The paper could have taken into account the number of issues and build failures in the respective repositories of the open source projects and check if the QA practice that is followed, has any effect on the software quality(issues).

The fifth paper, like the 2012 paper, does not talk about how shorter release cycles affect the code coverage numbers. Comparing code coverage for rapid releases can help observe the quality of unit tests.

In the sixth paper, we thought the authors could have checked the correlation between the number of vulnerabilities and the number of crash reports for each version of firefox since the inception of RRC. Also, The authors could have checked whether the results they got matched with the findings in other products (by running their test on other publicly available datasets).

In the seventh paper, the authors could have proven the effect of code size changes using the data. Besides this, the paper does not take into account the priority of the bug.

5. CONCLUSION

Today more companies are facing the heat from consumers and this increased competitiveness has prompted them to adopt shorter release cycles. In this essay, we set out to study various papers which covered the impact that shorter cycles had on software quality and the frequency of updates. Most papers performed informed analysis on the openly available firefox dataset. While a good portion of them expressed positive results overall for switching to the rapid release model. None of the papers suggested that one was better than the other. In other words, most papers have, for the most part, found out that good software quality is positively correlated with shorter release cycles, but none of them have proven causation. Besides, some of the results suggest that rapid release cycle isn't always good.

6. REFERENCES

[1] F. Khomh, T. Dhaliwal, Y. Zou and B. Adams. 2012.Do Faster Releases Improve Software Quality?: An Empirical Case Study of Mozilla Firefox.Proceedings of the 9th IEEE Working Conference on Mining Software Repositories. IEEE Press, 2012.

[2] 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

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

[4] 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, 2006..

[5] Mika V. Mantyla, Foutse Khomh, Bram Adams, Emelie Engstrom and Kai Petersen, "On Rapid Releases and Software Testing",Software Maintenance (ICSM) 2013 29th IEEE International Conference

[6] Sandy Clark, Micheal Collis, Matt Blaze and Jonathan M. Smith, "Security and Rapid-Release in Firefox", Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications

[7] Souza, Rodrigo, Christina Chavez, and Roberto A. Bittencourt. "Rapid releases and patch backouts: A software analytics approach." IEEE Software, 2015

[8] J. Feller and B. Fitzgerald, A framework analysis of the open source software development paradigm, in Proceedings of ICIS 2000, 2000, pp. 58-69.

[9] L. Zhao and S. Elbaum, A survey on software quality related activities in open source. ACM SIGSOFT Software Engineering Notes, 25 (3), 2000, pp. 54-57

[10] X. Zhang and H. Pham, An analysis of factors affecting software reliability. Journal of Systems and Software, 50(1), 2000, pp. 43-56.

[11] 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.

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

[13] 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.

[14] 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.

[15] D. Zhang et al., "Software Analytics in Practice", IEEE Software, vol. 30, no. 5, pp. 30-37, 2013.

[16] F. E. Harrell. Regression modeling strategies: with applications to linear models, logistic regression, and survival analysis. Springer, 2001.

[17] B. Efron. How biased is the apparent error rate of a prediction rule? Journal of the American Statistical Association, 81(394):461–470, 1986.

[18] Sandy Clark, Stefan Frei, Matt Blaze, and Jonathan Smith. Familiarity breeds contempt: the honeymoon effect and the role of legacy code in zero-day vulnerabilities. In Proceedings of the 26th Annual Computer Security Applications Conference, ACSAC ’10, pages 251–260, New York, NY, USA, 2010. ACM

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

[20] T.J. Halloran and W.L. Scherlis, High Quality and Open Source Software Practices. in Proceedings of ICSE 2002, Orlando, FL, USA, 2002