UNIVERSITÉ DE MONTRÉAL

UNDERSTANDING THE IMPACT OF RELEASE PROCESSES AND PRACTICES ON SOFTWARE QUALITY

LE AN
DÉPARTEMENT DE GÉNIE INFORMATIQUE ET GÉNIE LOGICIEL
ÉCOLE POLYTECHNIQUE DE MONTRÉAL

THÈSE PRÉSENTÉE EN VUE DE L'OBTENTION
DU DIPLÔME DE PHILOSOPHIÆ DOCTOR
(GÉNIE INFORMATIQUE)
AVRIL 2019

UNIVERSITÉ DE MONTRÉAL

ÉCOLE POLYTECHNIQUE DE MONTRÉAL

Cette thèse intitulée :

UNDERSTANDING THE IMPACT OF RELEASE PROCESSES AND PRACTICES ON SOFTWARE QUALITY

présentée par : AN Le
en vue de l'obtention du diplôme de : Philosophiæ Doctor
a été dûment acceptée par le jury d'examen constitué de :

M. MERLO Ettore, Ph. D., président
M. KHOMH Foutse, Ph. D., membre et directeur de recherche
M. ANTONIOL Giuliano, Ph. D., membre
M. ZIMMERMANN Thomas, Ph. D., membre externe

DEDICATION

To my family

ACKNOWLEDGEMENTS

Firstly, I would like to express my deepest gratitude to my advisor, Dr. Foutse Khomh, for his great guidance, encouragement, and patience that he provided along my Master's and Ph.D. studies. Today, Dr. Foutse Khomh is not only a professor, but also a friend, who helped and backed me to go through my hardest time during the last six years.

Secondly, I also would like to thank Marco Castelluccio, a senior software engineer at Mozilla. During my Ph.D. studies, Marco introduced me important background knowledges about Mozilla's development and release processes. He also helped me interview Mozilla managers to complete our research questions and to collect feedback of our findings. With Marco, we have published four papers in the last three years. One of our papers was granted the IEEE TCSE Distinguished paper award at the 33rd International Conference on Software Maintenance and Evolution (ICSME 2017).

Third, I also want to thank Dr. Shane McIntosh, Dr. David Lo, and Dr. Yann-Gaël Guéhéneuc for their help and mentoring on my thesis and other works during my Ph.D. studies.

In addition, I would like to thank the two funding agencies, *Natural Sciences and Engineering Research Council of Canada (NSERC)* and *Fonds de Recherche du Québec – Nature et Technologies (FRQNT)*, which financed me during the last three years.

Finally, I must thank my dearest family for their great support all the way along my postgraduate studies.

RÉSUMÉ

L'ingénierie de production (release engineering) englobe toutes les activités visant à «construire un pipeline qui transforme le code source en un produit intégré, compilé, empaqueté, testé et signé prêt à être publier». La stratégie des production et les pratiques de publication peuvent avoir un impact sur la qualité d'un produit logiciel. Bien que cet impact ait été longuement discuté et étudié dans la communauté du génie logiciel, il reste encore plusieurs problèmes à résoudre.

Cette thèse s'attaque à quelque-uns de ces problèmes non résoulus de l'ingénierie de production en vue de proposer des solutions. En particulier, nous investigons : 1) pourquoi les activités de révision de code (code review) peuvent rater des erreurs de code susceptibles de causer des plantages (crashs); (2) comment prévenir les bogues lors de l'approbation et l'intégration des patches urgents; 3) dans un écosystème logiciel, comment atténuer le risque de bogues dus à des injections de DLL. Nous avons choisi d'étudier ces problèmes car ils correspondent à trois phases importantes des processus de production de logiciels, c'est-à-dire la révision de code, les patches urgents, et la publication de logiciels dans un écosystème. Les solutions à ces problèmes peuvent aider les entreprises de logiciels à améliorer leur stratégie de production et de publication. Ce qui augmentera leur productivité de développement et la qualité générale de leurs produits logiciels.

En général, nous avons constaté que : 1) Les codes susceptibles de plantage sont souvent complexes et dépendent de nombreux autres codes. Pendant la révision de code, les développeurs dépensent beaucoup de temps (à travers de longues discussions) sur ces codes susceptibles de plantage. Utilisant une analyse manuelle sur un échantillon de codes susceptibles de plantage, nous avons constaté que la plupart de ces codes étaient utilisés pour améliorer les performances, faire du refactoring, rajouter des fonctionnalités, ou réparer les plantages précédents. Les erreurs de mémoire et les erreurs sémantiques sont identifiées comme les raisons principales des plantages. 2) La plupart des patches urgents sont utilisés pour fixer une mauvaise fonctionnalité ou un plantage. Certains d'entre eux ne fonctionnent pas comme prévu parce qu'ils n'ont pas résolu leurs problèmes ou ont introduits un nouveau bogue. Les patchs urgents qui ont introduits de nouveaux bogues contiennent souvent de nombreuses lignes de code. La plupart de leurs bogues sont dues à des erreurs de sémantique ou des erreurs de mémoire. Plus de 25% des bogues auraient pu être évités car ils pourraient être reproduits par les développeurs ou trouvées dans les endroits populairs (ex. les fonctionnalités / sites Web / configurations populaires) ou via la télémétrie. 3) Parmi les défaults d'injection de DLL que nous avons étudiés, 88,8% des bogues ont provoqués des plantages et 55,3% ont été causés par un logiciel antivirus. Utilisant un sondage auprès des fournisseurs des logiciels d'injections, nous avons constaté que certains fournisseurs n'effectuaient pas de contrôle qualité avec des versions préliminaires de Firefox, ni avaient l'intention d'utiliser une API publique (ex. WebExtensions).

Nos résultats suggèrent que: 1) Les entreprises de logiciels devraient bien examiner les patches qui contiennent du code complexe et des dependences compliqués. Ils doivent aider les réviseurs à concentrer leurs efforts d'inspection en utilisant des outils d'analyse statique. 2) Les gérants de la publication de logiciel (release managers) doivent soigneusement publier certains types de patches avant la date de publication prévue. Un grand nombre de bogues causés par des patches urgents auraient pu être évitées avec un test extensif. 3) Dans un écosystème, pour réduire les bogues dus aux injections de DLL, les entreprises de logiciel qui fournissent la platforme et qui fournissent les plug-ins doivent renforcer leur collaboration, par example, en créant un cadre de test de validation accessible au public. Les entreprises qui fournissent la platforme peuvent également utiliser une approche de liste blanche pour autoriser uniquement l'injection de DLL vérifiées.

ABSTRACT

Release engineering encompasses all the activities aimed at “building a pipeline that transforms source code into an integrated, compiled, packaged, tested, and signed product that is ready for release”. The strategy of the release processes and practices can impact the quality of a software artefact. Although such impact has been extensively discussed and studied in the software engineering community, there are still many pending issues to resolve.

The goal of this thesis is to study and solve some of these pending issues. More specifically, we examine 1) why code review practices can miss crash-prone code; 2) how urgent patches (also called patch uplift) are approved to release and how to prevent regressions due to urgent patches; 3) in a software ecosystem, how to mitigate the risk of defects due to DLL injections. We chose to study these problems because they correspond to three important phases of software release processes, *i.e.*, code review, patch uplift, and releasing software in an ecosystem. The solutions of these problems can help software organizations improve their release strategy; increasing their development productivity and the overall user-perceived quality of their products.

In general, we found that: 1) Crash-prone code tends to have high complexity and depend on many other classes. In the code review process, developers often spend a long time on and have long discussions about crash-prone code. Through a manual analysis on a sample of reviewed crash-prone code, we observed that most crash-prone patches aim to improve performance, refactor code, add functionality, or fix previous crashes. Memory and semantic errors are identified as major root causes of the crashes. 2) Most patches are uplifted because of a wrong functionality or a crash. Certain uplifts did not effectively address their problems because they did not completely fix the problems or lead to regressions. Uplifted patches that lead to regressions tend to have larger patch size, and most of the faults are due to semantic or memory errors in the patches. More than 25% of the regressions could have been prevented as they could be reproduced by developers and were found in widely used feature/website/configuration or via telemetry. 3) Among our studied DLL injection bugs, 88.8% of the bugs led to crashes and 55.3% of the bugs were caused by antivirus software. Through a survey with the software vendors who performed DLL injections, we observed that some vendors did not perform any QA with pre-release versions nor intend to use a public API (WebExtensions).

Our findings suggest that: 1) Software organizations should apply more scrutiny to patches with complex code and complex dependencies, and provide better support for reviewers to focus their inspection effort by using static analysis tools. 2) Release managers should carefully uplift certain kinds of patches. A considerable amount of uplift regressions could have been prevented through more extensive testing on the channels. 3) In an ecosystem, to reduce DLL injection bugs, host software vendors may strengthen the collaboration with third-party vendors, *e.g.*, build a publicly accessible validation test framework. Host software vendors may also use a whitelist approach to only allow vetted DLLs to inject.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ACRONYMS AND ABBREVIATIONS

DLL	Dynamic-link library
QA	Quality assurance
SZZ algorithm	Algorithm proposed by J. Śliwerski, T. Zimmermann, and A. Zeller, which is used to identify bug-inducing commits.

CHAPTER 1 INTRODUCTION

A good release strategy can help software organizations improve the quality of their product and the satisfaction of users.

The goal of release engineering is to “build a pipeline that transforms source code into an integrated, compiled, packaged, tested, and signed product that is ready for release” [1]. The processes and practices of release engineering have been extensively studied in previous works, including topics on the rapid release model [2], impact of code review on software release process [3], release of urgent patches [4], and software release in an ecosystem [5]. The findings of these studies indicated that a good release strategy can help software organizations improve the quality of their product and the satisfaction of users.

Despite of the fruitful achievements so far, there are still some problems that are closely related to the aforementioned topics but have not been systematically investigated, which motivated us to conduct this research. Specifically, we set out to answer the following questions: 1) why can code review practices miss crash-prone code? 2) how were urgent patches approved to release and how can software organizations prevent regressions due to urgent patches? 3) in a software ecosystem, how can vendors of *host* and *third-party software* (respectively software that allows other software to extend its functionality and software that adds code into another software) reduce defects due to DLL injections?

In the rest of this chapter, we will briefly describe the problems we aim to address and summarize our findings.

1.1 Why can code review practices miss crash-prone code?

Code review, *i.e.*, the practice of having other team members critique changes to a software system, is a pillar of modern software quality assurance approaches. Although this activity aims at improving software quality, some high-impact defects, such as crash-related defects, can elude the inspection of reviewers and escape to the field, affecting user satisfaction and increasing maintenance overhead. In this research, we investigate the characteristics of crash-prone code, observing that such code tends to have high complexity and depend on many other classes. In the code review process, developers often spend a long time on and have long discussions about crash-prone code. We manually classify a sample of reviewed crash-prone patches according to their purposes and root causes. We observe that most crash-prone patches aim to improve performance, refactor code, add functionality, or fix previous crashes. Memory and semantic errors are identified as major root causes of the crashes. Our results suggest that software organizations should apply more scrutiny to these types of patches, and provide better support for reviewers to focus their inspection effort by using static analysis tools.

1.2 How were urgent patches approved to release and how can software organizations prevent regressions due to these patches?

In rapid release development processes, patches that fix critical issues, or implement high-value features are often promoted directly from the development channel to a stabilization channel, potentially skipping one or more stabilization channels. This practice is called *patch uplift*. Patch uplift is risky, because patches that are rushed through the stabilization phase can end up introducing regressions in the code. We examined patch uplift operations at Mozilla, with the aim to identify the characteristics of the uplifted patches that did not effectively fix the targeted problem and that introduced regressions. Through statistical and manual analyses, a series of problems were investigated, including the reasons behind patch uplift decisions, the root causes of ineffective uplifts, the characteristics of uplifted patches that introduced regressions, and whether these regressions can be prevented. Additionally, three Mozilla release managers were interviewed in order to understand organizational factors that affect patch uplift decisions and outcomes. Results show that most patches are uplifted because of a wrong functionality or a crash. Certain uplifts did not effectively address their problems because they did not completely fix the problems or lead to regressions. Uplifted patches that lead to regressions tend to have larger patch size, and most of the faults are due to semantic or memory errors in the patches. Also, release managers are more inclined to accept patch uplift requests that concern certain specific components, and–or that are submitted by certain specific developers. About 25% to 30% of the regressions due to Beta or Release uplifts could have been prevented as they could be reproduced by developers and were found in widely used feature/website/configuration or via telemetry.

1.3 How can host and third-party vendors reduce defects due to DLL injections in a software ecosystem?

DLL injection is a technique used for executing code within the address space of another process by forcing the load of a dynamic-link library. In a software ecosystem, the interactions between the host and third-party software increase the maintenance challenges of the system and may lead to bugs. In this work, we empirically investigate bugs that were caused by third-party DLL injections into the Mozilla Firefox browser. Among the studied DLL injection bugs, we found that 88.8% of the bugs led to crashes and 55.3% of the bugs were caused by antivirus software. Through a survey with third-party software vendors, we observed that some vendors did not perform any QA with pre-release versions nor intend to use a public API (WebExtensions) but insist on using DLL injection. To reduce DLL injection bugs, host software vendors may strengthen the collaboration with third-party vendors, *e.g.*, build a publicly accessible validation test framework. Host software vendors may also use a whitelist approach to only allow vetted DLLs to inject.

1.4 Research Statement

Leveraging Mozilla Firefox as the subject system, this thesis investigated three important problems on release engineering processes and practices: the reasons why crash-prone code eluded from code review; the characteristics of uplifted patches and characteristics of ineffective (including defective) uplifted patches; and the solutions to mitigate the risks of DLL injections in a software ecosystem.

1.5 Thesis Overview

- *Why can code review practices miss crash-prone code (Chapter 3)?*
  We compared the characteristics of the reviewed patches that crashed with the ones that did not crash, and manually investigated the purposes and root causes of crash-prone patches. We made suggestions to improve the code review process in order to catch more crash-prone code.

- *How were urgent patches approved to release and how can software organizations prevent regressions due to these patches (Chapter 4)?*
  We compared the characteristics of patches that were accepted to uplift with the ones that were rejected. We also examined the uplifted patches that did not address the problems they aim at, investigated the purposes of uplifted patches and the root causes of uplift regressions, and finally proposed solutions to prevent regressions due to patch uplift.

- *How can host and third-party vendors reduce defects due to DLL injections in a software ecosystem (Chapter 5)?*
  We quantitatively investigated the characteristics of the defects caused by DLL injections, and qualitatively investigated the factors that trigger DLL injection defects.

Based on a survey with third-party software vendors who injected code into Mozilla Firefox, we proposed solutions to reduce DLL injection defects in Firefox and its equivalent ecosystems.

1.6 Thesis Contribution

In this thesis, we conducted empirical studies on different aspects of software release processes and practices. Our contributions are summarized as follows:

- We observed that reviewers often spend a long time on and have long discussions about crash-prone code. Most crash-prone patches were originally used to improve performance, refactor code, add functionality, or fix previous crashes. Memory and semantic errors are identified as major root causes of the crashes that eluded from the code review process. Our findings suggest that software practitioners should more carefully scrutinize certain types of patches and apply static analysis tools to assist their code review practices.
- Through quantitative and qualitative analyses, we observed that most patches were uplifted in order to fix a wrong functionality or a crash. Among the studied uplifted patches, certain patches did not completely fix the targeted problems or lead to regressions. Uplifted patches that lead to regressions often have larger patch size, and most of the regressions are due to semantic or memory errors, thus static analysis tools can be leveraged to reduce the errors. In addition, more than 25% of the studied regressions could have been prevented through more extensive testing because they can be reproduced by developers.
- We found that 88.8% of the DLL injection bugs led to crashes and 55.3% of the bugs were caused by antivirus software. According to the results of our survey, some third-party software vendors acknowledged that they did not perform any QA with pre-release versions nor intend to use a public API (*i.e.*, WebExtensions as Mozilla recommended) but insist on using DLL injection. To improve the release processes for both host and third-party vendors, we suggest that they should strengthen the collaboration between them, such as building a publicly accessible validation test framework.

1.7 Organization of the Thesis

The rest of this thesis is organized as follows:

- Chapter 2 outlines literature review in the areas of rapid release model, code review, crash report analysis, urgent patches, software ecosystems, and DLL injection.

- Chapter 3 presents our empirical study on reviewed patches that still crashed.
- Chapter 4 presents our empirical study on patch uplift in rapid release development pipelines of Mozilla Firefox.
- Chapter 5 presents our empirical study on the DLL injection bugs in the Firefox ecosystem.
- Chapter 6 summarizes and concludes this thesis and discuss future work.

CHAPTER 2 LITERATURE REVIEW

2.1 Rapid Release Model

Since the adoption of the rapid release model [2] by Mozilla in 2011, a plethora of studies have focused on the impact of rapid release strategies on software quality. Khomh et al. [2] compared crash rates, median uptime, and the proportion of post-release bugs between the versions of Firefox that followed a traditional release cycle and those that followed a rapid release cycle. They observed that short release cycles do not induce significantly more bugs. However, compared to traditional releases, users experience bugs earlier during software execution. Nevertheless, they also observed that post-release bugs are fixed faster under the rapid release model. Khomh et al. observed, in their extended work [6], that one of the major challenges of fast release cycles is the automation of the release engineering process. Da Costa et al. [7] studied the impact of Mozilla's rapid release cycles on the integration delay of addressed issues. They found that, compared to the traditional release model, the rapid release model does not deliver addressed issues to end users more quickly, which is contrary to expectations. Adams et al. [8] analyzed the six major phases of release engineering practices and proposed a roadmap for future research, highlighting the need for more empirical studies that validate the best practices and assess the impact of release engineering processes on software quality.

2.2 Code Review

One important goal of code review is to identify defective code at early stages of development before it affects end users. Software organizations expect that this process can improve the quality of their systems.

Previous studies have investigated the relationship between code review quality and software quality. Jiang et al. [3] cross analyzed reviewed patches from Linux kernel mailing list and landed patches from the kernel's version control system. They found that a third of the reviewed patches were eventually released to end users and most of these patches need three to six months to get released. They also found that patches from more experienced developers tend to be more easily accepted and be faster reviewed and integrated. McIntosh et al. [9, 10] found that low code review coverage, participation, and expertise share a significant link with the post-release defect proneness of components in the Qt, VTK, and ITK projects. Similarly, Morales et al. [11] found that code review activity shares a relationship with design quality in the same studied systems. Thongtanunam et al. [12] found that lax code reviews tend to happen in defect-prone components both before and after defects were found, suggesting that developers are not aware of problematic components. Kononenko et al. [13] observed that 54% of the reviewed changes are still implicated in subsequent bug fixes in Mozilla projects. Moreover, their statistical analysis suggests that both personal and review participation metrics are associated with code review quality. In a recent work, Sadowski et al. [14] conducted a qualitative study of the code review practices at Google. They observed that problem solving is not the only focus for Google reviewers and only a few developers said that code review have helped them catch bugs.

The results of [9, 10, 12, 13, 14] suggest that despite being reviewed, many changes still introduce defects. Therefore, in this thesis, we investigate the relationship between the rigour of the code review that a code change undergoes and its likelihood of inducing a software crash – a type of defect with severe implications. We draw inspiration from these prior studies to design our set of metrics [12, 15] (see Chapter 3).

2.3 Crash Report Analysis

Crashes can unexpectedly terminate a software system, resulting in data loss and user frustration. To evaluate the importance of crashes in real time, many software organizations have implemented automatic crash collection systems to collect field crashes from end users.

Previous studies analyze the crash data from these systems to propose debugging and bug fixing approaches for crash-related defects. Podgurski et al. [16] introduced an automated failure clustering approach to classify crash reports. This approach enables the prioritization and diagnosis of the root causes of crashes. Khomh et al. [17] proposed an entropy-based approach to identify crash-types that frequently occurred and affect a large number of users. Kim et al. [18] mined crash reports and the related source code in Firefox and Thunderbird to predict top crashes for a future release of a software system. To reduce the efforts of debugging crashing code, Wu et al. [19] proposed a framework, ChangeLocator, which can automatically locate crash-inducing changes from a given bucket of crash reports.

In most of these aforementioned works, researchers analyzed data from the Mozilla Socorro crash reporting system [20] because at the time of writing of this paper, only the Mozilla Foundation has opened its crash data to the public [21]. Though Wang et al. [21] studied another system, Eclipse, they could obtain crash information from the issue reports (instead of crash reports). However, the exact crash date are not provided in these issue reports, which hampers our ability to apply the SZZ algorithm [22]. Dang et al. [23] proposed a method, ReBucket, to improve the current crash report clustering technique based on call stack matching. However, their studied collection of crash reports from the Microsoft Windows Error Reporting (WER) system is not accessible for the public.

In this thesis, we leverage crash data from the Mozilla Socorro system to quantitatively and qualitatively investigate the reasons why reviewed code still led to crashes, and make suggestions to improve the code review process.

2.4 Urgent Patches

Another important aspect of release engineering that has been investigated by the community is the integration of urgent patches that are used to fix severe problems, such as frequent crashes or security bugs, or to introduce important features. Urgent patches break the balance between new feature work and software quality, and hence could lead to faults and failures. Hassan et al. [4] investigated emergency updates for top Android apps and identified eight patterns along the following two categories: "updates due to deployment issues" and "updates due to source code changes". They suggest to limit the number of emergency updates that fall in these patterns, since they are likely to have a negative impact on users' satisfaction. In a recent work, Lin et al. [24] empirically analyzed urgent updates in 50 most popular games on the Steam platform, and observed that the choice of the release strategy affects the proportion of urgent updates, *i.e.*, games that followed a rapid release model had a higher proportion of urgent patches in comparison to those that followed the traditional release model. Rahman et al. [25] examined the "rush to release" period on Linux and Chrome. They observed that experienced developers are often allowed to make changes right before stabilization occurs and these changes are added directly to the stabilization line. They also found that there is a rush in the number of commits right before a new release is added to the stabilization channel, to add final features. In a following work, Rahman et al. [26] observed that feature toggles [27] can effectively turn off faulty urgent patches, which limits the impact of faulty patches.

To the best of our knowledge, none of these prior works has empirically investigated how urgent patches in the rapid release model affect software quality in terms of fault proneness, and how the reliability of the integration of urgent updates could be improved. This work fills this gap in the literature by investigating the reliability of the Mozilla's uplift process, since uplifted patches are urgent updates (see Chapter 4).

2.5 Software Ecosystems

When a software organization increasingly allows other software to join and extend its software platform, an ecosystem is gradually formed. Many software organizations have realized that either creating or joining into such an ecosystem can be beneficial because they no longer have to produce an entire system but only need to work for a part of it. Recently, we have seen an increase in the number of software ecosystems and the number of research studies that have focused on them. Bosch [28] observed the emerging trend of the transition from traditional software product lines to software ecosystems and proposed actions required for this transition. He also discussed the implications of adopting a software ecosystem approach on the way organizations build software. Hanssen [5] conducted an empirical study of the CSoft system, which transitioned from a closed and plan-driven approach towards an ecosystem. He observed that transitioning to a software ecosystem improved the cross organizational collaboration and the development of a shared value (*i.e.*, technology and business) in the collaboration. Jansen et al. [29] discussed the challenges of software ecosystems at the levels of software ecosystems themselves, software supply network, and software vendors. This early work provided a guideline for software vendors to make their software adaptable to new business models and new markets, and help them to choose appropriate strategy to succeed in an ecosystem. Later on, Van Den Berk et al. [30] built models to quantitatively assess the status of a software ecosystem as well as the success of decisions taken by the host vendors in the past.

Researchers have also empirically studied various popular open source ecosystems, including Linux kernel (*e.g.*, [31]), Debian distribution (*e.g.*, [32, 33]), Eclipse (*e.g.*, [34, 35]), and R (*e.g.*, [36]) ecosystems. The host software in these ecosystems are respectively operating system, integrated development environment, and mathematical software. However, as far as we know, there is no previous study that empirically investigates a browser-based open source ecosystem (*e.g.*, Firefox, Chrome). Although Liu et al. [37] studied the extension security model of Chrome and Karim et al. [38] studied the Jetpack Extension Framework of Mozilla, their research focused on the extension techniques rather than on the ecosystems. We contribute to filling this gap by conducting an empirical study of DLL injection bugs in the Firefox ecosystem. Another difference between our work and these previous works [37, 38] is that DLL injection is completely arbitrary, *i.e.*, third-party software can execute whatever it requires; while the extension API can constrain third-party software's behaviour.

2.6 DLL Injection

DLL injection is one of the popular ways to insert code into other software. It can force a process to load external code in a manner that the author of the process does not anticipate or intend. Leveraging the DLL injection technique, Andersson et al. [39] proposed a framework to detect code injection attacks [40]. Lam et al. [41] proposed an approach that uses DLL injection to isolate the execution of the incoming email attachments and web documents on a physically separate machine rather than on the user machine. Their approach can help reduce the risk that user machines are attacked. Berdajs et al. [42] analyzed the limitations of multiple existing DLL injection techniques (including CreateRemoteThread, proxy DLL, Windows hooks, using a debugger, and reflective injection) and introduced a new approach that combines DLL injection and API hooking (a technique by which we can modify the behaviour and flow of an API call [43]). The improved approach can inject code even when the application is not fully initialized.

As DLL injection allows a program to inject arbitrary code into arbitrary processes [44], malware producers can also take advantage of this technique to exploit computers. Jang et al. [45] proposed an approach to help identify malicious DLLs in Windows. Windows maintains a list of all loaded modules, including DLL modules. Some software checks this list to detect DLLs injected from another process and take corresponding measures, *e.g.*, block it if a DLL is suspicious. However, an approach called Reflective injection [46] can inject DLLs in a stealthy manner, which increases the difficulty of detecting suspicious DLLs.

Like a double-edged sword, DLL injection is a useful (even indispensable) programming technique, but can also cause severe damages due to its arbitrary nature. To the best of our knowledge, we are not aware of any existing work that empirically studied the root causes and counterplans of the bugs or defects caused by DLL injection. Particularly, in a software ecosystem, this kind of bugs can hardly be predicted but can affect a large number of users. To help software practitioners understand the root causes of DLL injection bugs and propose solutions to reduce them, we conduct a case study on the Firefox ecosystem (see Chapter 5).

2.7 Chapter Summary

In this chapter, we briefly discussed literatures on release engineering processes and practices that are related to the problems we aim to address: the reasons why reviewed patches still crashed; the characteristics of uplifted patches and the solutions to reduce regressions due to uplifted patches; the reasons why certain third-party vendors insist on DLL injections rather than the recommended API and the solutions to mitigate the risk of DLL injections.

Specifically, we reviewed related works in the following fields: rapid release model, code review, crash analysis, urgent patches, software ecosystems, and DLL injection techniques.

In the following chapters, we will describe our empirical studies on the aforementioned problems and propose solutions to help software organizations improve their release strategy; decreasing the number of bugs and increasing the overall user-perceived quality.

CHAPTER 3 WHY DID THIS REVIEWED CODE CRASH?[*]

> *Despite being reviewed, many changes still introduce defects, including crashes.*

A software crash refers to an unexpected interruption of software functionality in an end user environment. Crashes may cause data loss and frustration of users. Frequent crashes can decrease user satisfaction and affect the reputation of a software organization. Practitioners need an efficient approach to identify crash-prone code early on, in order to mitigate the impact of crashes on end users. Nowadays, software organizations like Microsoft, Google, and Mozilla are using crash collection systems to automatically gather field crash reports, group similar crash reports into crash-types, and file the most frequently occurring crash-types as bug reports.

Code review is an important quality assurance activity where other team members critique changes to a software system. Among other goals, code review aims to identify defects at early stages of development. Since reviewed code is expected to have better quality, one might expect that reviewed code would tend to cause few severe defects, such as crashes. However, despite being reviewed, many changes still introduce defects, including crashes. For example, Kononenko et al. [13] find that 54% of reviewed code changes still introduce defects in Mozilla projects.

In this chapter, we want to understand the reasons why reviewed code still led to crashes. To achieve these goals, we mine the crash collection, version control, issue tracking, and code reviewing systems of the Mozilla Firefox project. More specifically, we address the following two research questions:

RQ1: *What are the characteristics of reviewed code that is implicated in a crash?*

We find that crash-prone reviewed patches often contain complex code, and classes with many other classes depending on them. Crash-prone patches tend to take a longer time and generate longer discussion threads than non-crash-prone patches. This result suggests that reviewers need to focus their effort on the patches with high complexity and on the classes with a complex relationship with other classes.

RQ2: *Why did reviewed patches crash?*

[*] Part of the content of this chapter is published in “Why Did This Reviewed Code Crash? An Empirical Study of Mozilla Firefox”, Le An, Foutse Khomh, Shane McIntosh, and Marco Castelluccio, *Proceedings of the 25th Asia-Pacific Software Engineering Conference (APSEC), December 2018.*

To further investigate why some reviewed code crashes, we perform a manual classification on the purposes and root causes of a sample of reviewed patches. We observe that the reviewed patches that crash are often used to improve performance, refactor code, address prior crashes, and implement new features. These findings suggest that software organizations should impose a stricter inspection on these types of patches. Moreover, most of the crashes are due to memory (especially null pointer dereference) and semantic errors. Software organizations can perform static code analysis prior to the review process, in order to catch these memory and semantic errors before crashes escape to the field.

Chapter Overview

Section 3.1 provides background information on Mozilla crash collection system and code review process. Section 3.2 describes how we identify reviewed code that leads to crashes. Section 3.3 describes our data collection and analysis approaches. Section 3.4 discusses the results of the two research questions. Section 3.5 discloses the threats to the validity of our study, and Section 3.6 summarizes this chapter.

3.1 The Mozilla Crash Collecting System and Code Review Process

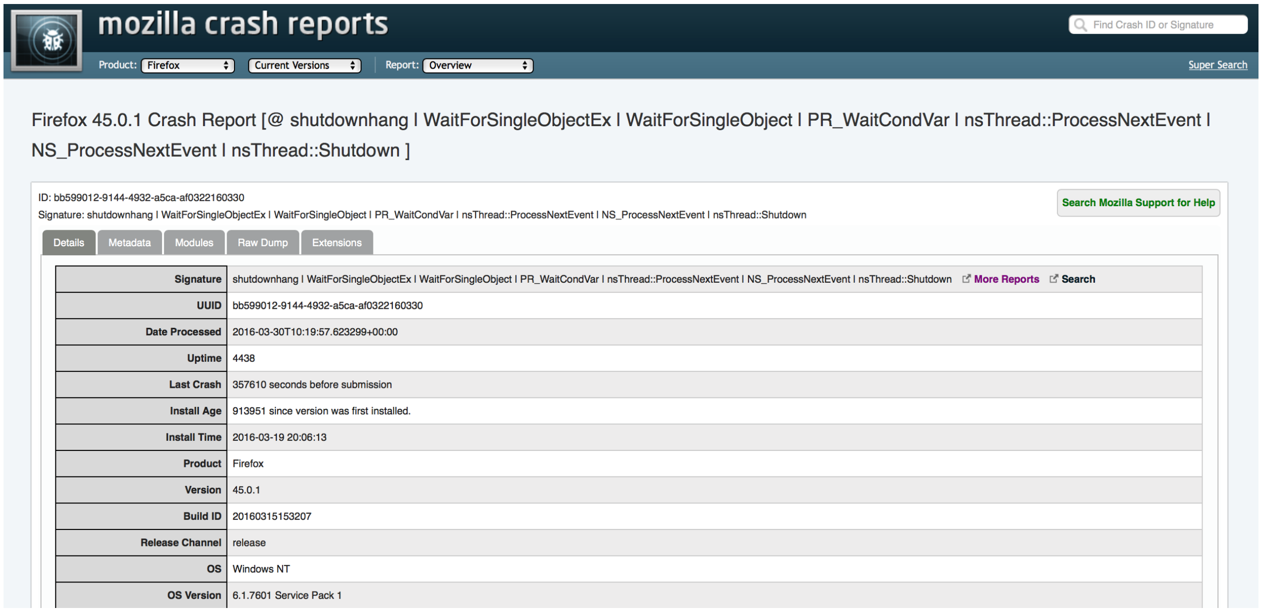
In this section, we describe approaches of Mozilla on crash report collection and code review.

3.1.1 The Mozilla Crash Collection System

Mozilla integrates the Mozilla Crash Reporter, a crash report collection tool, into its software applications. Once a Mozilla application, such as the Firefox browser, unexpectedly halts, the Mozilla Crash Reporter will generate a detailed crash report and send it to the *Socorro* crash report server [20]. Each crash report includes a stack trace of the failing thread and the details of the execution environment of the user. Figure 3.1 shows an example Socorro crash report. These crash reports are a rich source of information, which provide developers and quality assurance personnel with information that can help them to reproduce the crash in a testing environment.

The Socorro server automatically clusters the collected crash reports into *crash-types* according to the similarity of the top method invocations of their stack traces. Figure 3.2 shows an example Mozilla crash-type. The Socorro server ranks crash-types according to their frequency, *e.g.*, Socorro publishes a daily top 50 crash-types, *i.e.*, the crash-types with the maximum number of crash reports, for each of the recent releases of Firefox.

Socorro operators file top-ranked crash-types as issue reports in the *Bugzilla* issue tracking



Product: Firefox | Current Versions | Report: Overview

Firefox 45.0.1 Crash Report [@ shutdownhang | WaitForSingleObjectEx | WaitForSingleObject | PR_WaitCondVar | nsThread::ProcessNextEvent | NS_ProcessNextEvent | nsThread::Shutdown]

ID: bb599012-9144-4932-a5ca-af0322160330
Signature: shutdownhang | WaitForSingleObjectEx | WaitForSingleObject | PR_WaitCondVar | nsThread::ProcessNextEvent | NS_ProcessNextEvent | nsThread::Shutdown

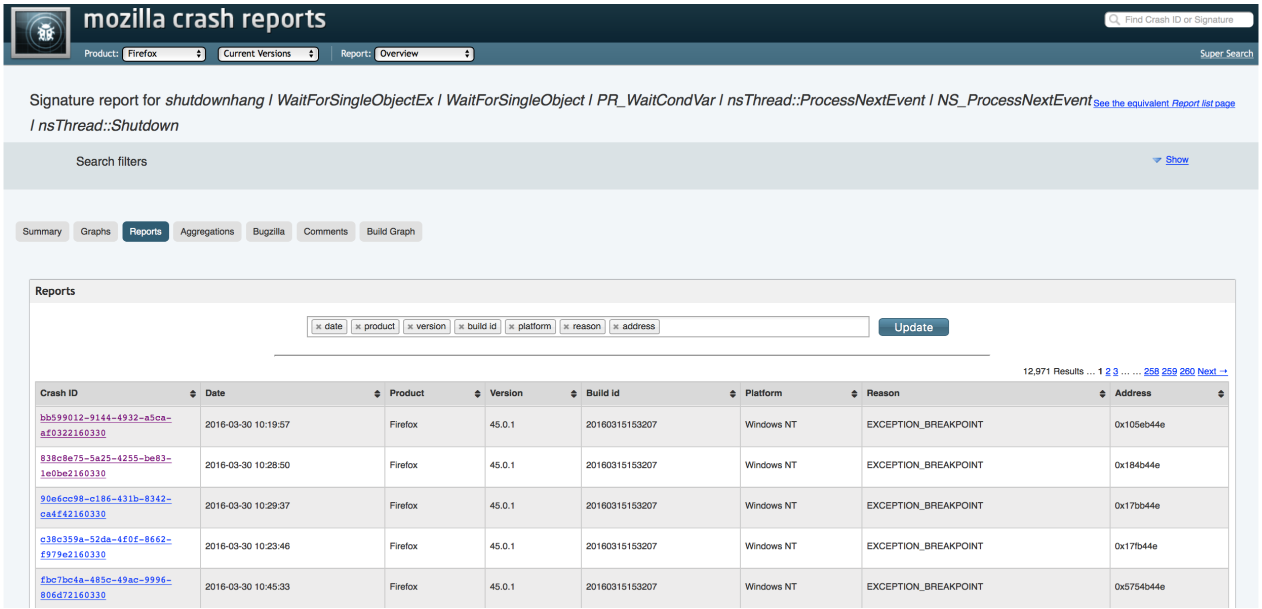
Signature	shutdownhang | WaitForSingleObjectEx | WaitForSingleObject | PR_WaitCondVar | nsThread::ProcessNextEvent | NS_ProcessNextEvent | nsThread::Shutdown More Reports Search
UUID	bb599012-9144-4932-a5ca-af0322160330
Date Processed	2016-03-30T10:19:57.623299+00:00
Uptime	4438
Last Crash	357610 seconds before submission
Install Age	913951 since version was first installed.
Install Time	2016-03-19 20:06:13
Product	Firefox
Version	45.0.1
Build ID	20160315153207
Release Channel	release
OS	Windows NT
OS Version	6.1.7601 Service Pack 1

Figure 3.1 An example of crash report in Socorro.

system. Quality assurance teams use Socorro to triage these crash-related issue reports and assign severity levels to them [47]. For traceability purposes, Socorro crash reports provide a list of the identifiers of the issues that have been filed for each crash-type. This link is initiated from Bugzilla. If a bug is opened from a Socorro crash, it is automatically linked. Otherwise, developers can add Socorro signatures to the bug reports. By using these traceability links, software practitioners can directly navigate to the corresponding issues (in Bugzilla) from the summary of a crash-type in the web interface of Socorro. Note that different crash-types can be linked to the same issue, while different issues can also be linked to the same crash-type [17].

3.1.2 The Mozilla Code Review Process

Mozilla manages its code review process using issue reports in Bugzilla. After writing a patch for an issue, the developer can request peer reviews by setting the `review?` flag on the patch. At Mozilla, the reviewers are often chosen by the patch author herself [48]. If the patch author does not know who should review her patch, they can consult a list of module owners and peers. Senior developers can also often recommend good reviewers. The designated reviewers need to inspect a patch from various aspects [49], such as correctness, style, security, performance, and compatibility. Once a developer has reviewed the patch, they can record comments with a review flag, which also indicates their vote, *i.e.*, in support



Signature report for *shutdownhang | WaitForSingleObjectEx | WaitForSingleObject | PR_WaitCondVar | nsThread::ProcessNextEvent | NS_ProcessNextEvent | nsThread::Shutdown* See the equivalent *Report list* page

Reports

12,971 Results … 1 2 3 … … 258 259 260 Next →

Crash ID	Date	Product	Version	Build id	Platform	Reason	Address
bb599012-9144-4932-a5ca-af0322160330	2016-03-30 10:19:57	Firefox	45.0.1	20160315153207	Windows NT	EXCEPTION_BREAKPOINT	0x105eb44e
838c8e75-5a25-4255-be83-1e0be2160330	2016-03-30 10:28:50	Firefox	45.0.1	20160315153207	Windows NT	EXCEPTION_BREAKPOINT	0x184b44e
90e6cc98-c186-431b-8342-ca4f42160330	2016-03-30 10:29:37	Firefox	45.0.1	20160315153207	Windows NT	EXCEPTION_BREAKPOINT	0x17bb44e
c38c359a-52da-4f0f-8662-f979e2160330	2016-03-30 10:23:46	Firefox	45.0.1	20160315153207	Windows NT	EXCEPTION_BREAKPOINT	0x17fb44e
fbc7bc4a-485c-49ac-9996-806d72160330	2016-03-30 10:45:33	Firefox	45.0.1	20160315153207	Windows NT	EXCEPTION_BREAKPOINT	0x5754b44e

Figure 3.2 An example of crash-type in Socorro.

of (+) or in opposition to (-) the patch. Mozilla applies a two-tiered code review process, *i.e.*, *review* and *superreview*. A *review* is performed by the owner of the module or peer who has expertise in a specific aspect of the code of the module [50]; while a *superreview* is required for certain types of changes, such as significant architectural refactoring, API or pseudo-API changes, or changes that affect the interactions of modules [51]. Therefore, to evaluate patches, there are four possible voting combinations on a reviewed patch: `review+`, `review-`, `superreview+`, and `superreview-`.

A code review may have several iterations. Unless the patch receives only positive review flags (`review+` or `superreview+`), it cannot be integrated into the version control system (VCS) of Mozilla. In this case, the patch author needs to provide a revised patch for reviewers to consider. Some Mozilla issues are resolved by a series of patches. Since the patches are used to address the same issue, reviewers need to inspect the entire series of patches before providing a review decision. In the trial review platform of Mozilla, ReviewBoard, the patches of an issue are automatically grouped together [52]. Thus, in this study, we examine the review characteristics at the issue level. Finally, the Tree Sheriffs [53] (*i.e.*, engineers who support developers in committing patches, ensuring that the automated tests are not broken after commits, and monitoring intermittent failures, and reverting problematic patches) or the patch authors themselves will commit the reviewed patches to the VCS.

3.2 Identifying Reviewed Code that Crashes

In this section, we describe our approach to identify reviewed code that is implicated in a crash report. Our approach consists of three steps: identifying crash-related issues, identifying commits that are implicated in future crash-related issues, and linking code reviews to commits. Below, we elaborate on each of these steps.

3.2.1 Identifying Crash-related Issues

Mozilla receives 2.5 million crash reports on the peak day of each week. In other words, the Socorro server needs to process around 50GB of data every day [54]. For storage capacity and privacy reasons, Socorro only retains those crash reports that occurred within the last six months. Historical crash reports are stored in a crash analysis archive[1]. We mine this archive to extract the *issue list*, which contains issues that are linked to a crash, from each crash event. These issues are referred as to *crash-related issues* in the rest of this chapter.

3.2.2 Identifying Commits that are Implicated in Future Crash-related Issues

We apply the SZZ algorithm [22] to identify commits that introduce crash-related issues. First of all, we use Fischer et al.'s heuristic [55] to find commits that fixed a crash-related issue I by using regular expressions to identify issue IDs from commit messages. Then, we extract the modified files of each crash-fixing commit with the following Mercurial command:

```
hg log --template {node},{file_mods}
```

By using the CLOC tool [56], we find that 51% of the Firefox codebase is written in C/C++. Although JavaScript and HTML (accounts for respectively 20% and 14% in the code base) are the second and third most used languages. Code implemented by these languages cannot directly cause crashes because it does not have direct hardware access. Crash-prone Javascript/HTML changes are often due to the fault of parsers, which are written in C/C++. Therefore, in this research, we focus our analysis on C/C++ code. Given a file F of a crash-fixing commit C, we extract C's parent commit C', and use the `diff` command of Mercurial to extract F's deleted line numbers in C', henceforth referred to as *rm_lines*. Next, we use the `annotate` command of Mercurial to identify the commits that introduced the *rm_lines* of F'. We filter these potential crash-introducing candidates by removing those commits that were submitted after I's first crash report. The remaining commits are referred to as *crash-inducing commits*.

[1] `https://crash-stats.mozilla.com/api/`

As mentioned in Section 3.1.2, Mozilla reviewers and release managers consider all patches together in an issue report during the review process. If an issue contains multiple patches, we bundle its patches together. Among the studied issues whose patches have been approved by reviewers, we identify those containing committed patches that induce crashes. We refer to those issues as *crash-inducing issues*.

3.3 Case Study Design

In this section, we present the selection of our studied system, the collection of data, and the analysis approaches that we use to address our research questions.

3.3.1 Studied System

We use Mozilla Firefox as the subject system because at the time of writing of this thesis, only the Mozilla Foundation has opened its crash data to the public [21]. It is also the reason why in most previous empirical studies of software crashes (*e.g.*, [17, 18]), researchers analyzed data from the Mozilla Socorro crash reporting system [20]. Though Wang et al. [21] studied another system, Eclipse, they could obtain crash information from the issue reports (instead of crash reports). However, the exact crash date are not provided in these issue reports, which hampers our ability to apply the SZZ algorithm. Dang et al. [23] proposed a method, ReBucket, to improve the current crash report clustering technique based on call stack matching. The studied collection of crash reports from the Microsoft Windows Error Reporting (WER) system is not accessible for the public.

3.3.2 Data Collection

We analyze the Mozilla crash report archive. We collect crash reports that occurred between February 2010 (the first crash recorded date) until September 2015. We collect issue reports that were created during the same period. We only take closed issues into account. We filter out the issues that do not contain any successfully reviewed patch (*i.e.*, patch with a review flag `review+` or `superreview+`). To select an appropriate study period, we analyze the rate of crash-inducing commits throughout the collected timeframe (March 2007 until September 2015). Figure 4.1 shows the rate of crash-inducing commits over time. In this figure, each time point represents one quarter (three months) of data. We observe that the rate of crash-inducing commits increases from January 2007 to April 2010 before stabilizing between April 2010 and April 2015. After April 2015, the rate suddenly drops. Since the last issue report is collected in September 2015, there is not enough related information to identify crash-

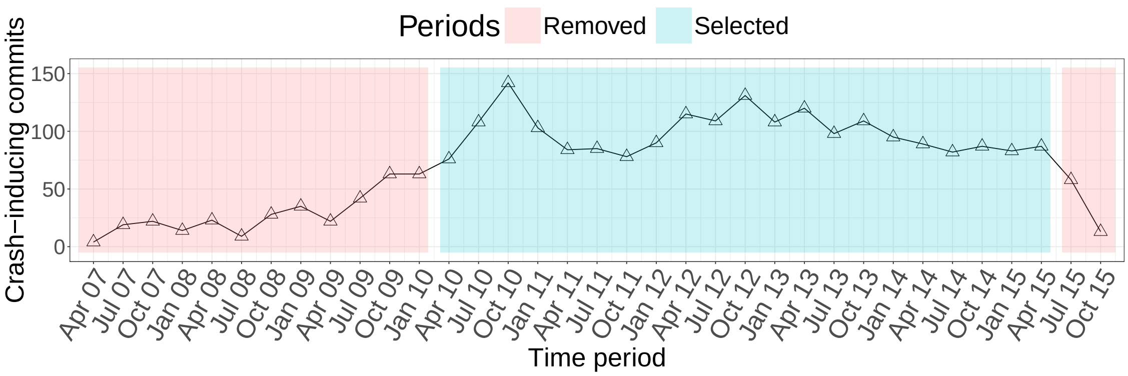


Figure 3.3 Number of crash-inducing commits during each three months from March 2007 to September 2015. Periods with low number of crash-inducing commits are removed.

inducing commits during the last months. Using Figure 4.1, we select the periods between April 2010 and April 2015 as our study period and focus our analysis on the crash reports, commits, and issue reports during this period. In total, we analyze 9,761,248 crash-types (from which 11,421 issue IDs are identified), 41,890 issue reports, and 97,840 commits. By applying the SZZ algorithm from Section 4.2.2, we find 1,202 (2.9%) issue reports containing reviewed patches that are implicated in crashes.

3.3.3 Data Extraction

We compute metrics for reviewed patches and the source code of the studied system. Figure 3.4 provides an overview of our data extraction steps. To aid in the replication of our study, our data and scripts are available online.[2]

Review Metrics

For each reviewed patch, we extract the names of the author and reviewer(s), as well as its creation date, reviewed date, patch size, and the votes from each of the review activities. We also extract the list of modified files from the content of the patch. Although main review activities of Mozilla are organized in Bugzilla attachments, we can also extract additional review-related information from Bugzilla comments and transaction logs. If a comment is concerned with an attachment like a patch, Bugzilla provides a link to the attachment in the comment. We can use this to measure the review discussion length of a patch. Bugzilla attachments only contain votes on review decisions, such as `review+` and `review-`. To obtain the date when a review request for a patch was created, we search for the `review?` activity

[2] `https://github.com/swatlab/crash_review`

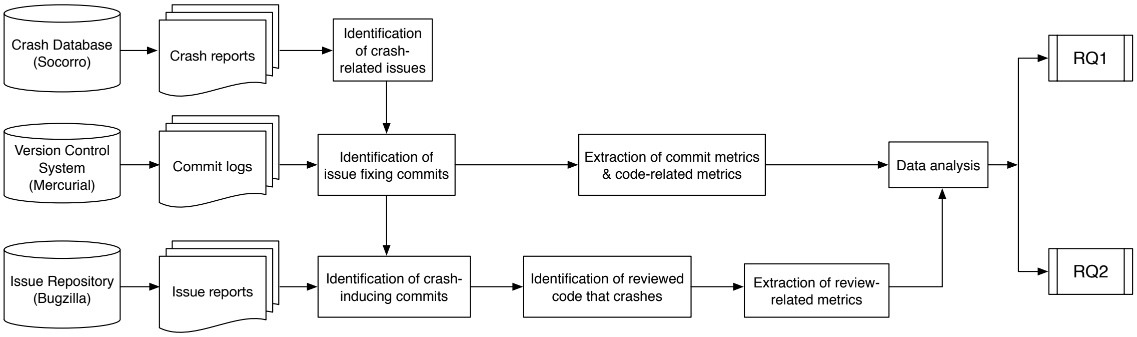

Crash Database (Socorro)
Crash reports
Identification of crash-related issues
Version Control System (Mercurial)
Commit logs
Identification of issue fixing commits
Extraction of commit metrics & code-related metrics
Data analysis
RQ1
RQ2
Issue Repository (Bugzilla)
Issue reports
Identification of crash-inducing commits
Identification of reviewed code that crashes
Extraction of review-related metrics

Figure 3.4 Overview of our approach to identify and analyze reviewed code that crashed in the field.

date in the issue discussion history. As we consider all of the patches of an issue together, we use the mean to aggregate patch-specific values to the issue-level. Unlike other systems, such as Qt [12], Mozilla does not allow self-review, *i.e.*, the author of a patch cannot act as a reviewer of that patch. However, Mozilla patch authors may set the `review+` score themselves, from time to time, when reviewers are generally satisfied with the patch with the exception of minor changes. Thus in this work, we remove the patch author from the reviewer list of each of the studied issues. More details on our review metrics are provided in Section 3.4.

Code Complexity Metrics

To analyze whether reviewed code that crashed in the field is correlated with code complexity, we compute code complexity metrics using the *Understand* static code analysis tool [57]. We wrote a script to compute five code complexity metrics for each C/C++ file using Understand, *i.e.*, Lines Of Code (LOC), average cyclomatic complexity, number of functions, maximum nesting level, and the proportion of comment lines in a file. More details on our complexity metrics are provided in Section 3.4.

Social Network Analysis Metrics

To measure the relationship among classes, we apply Social Network Analysis (SNA) [58] to measure the centrality [59] of each C++ class (or C file), *i.e.*, the degree to which other classes depend on a certain class. A high centrality value indicates that a class is important to a large portion of the system, and any change to the class may impact a large amount of functionality. We compute centrality using the class-to-class dependencies that are provided by Understand. We combine each `.c` or `.cpp` file with its related `.h` file into a *class node.*

We use a pair of vertices to represent the dependency relationship between any two mutually exclusive class nodes. Then, we build an adjacency matrix [60] with these vertex pairs. By using the *igraph* network analysis tool [61], we convert the adjacency matrix into a call graph, based on which we compute the PageRank, betweenness, closeness, indegree, and outdegree SNA metrics.

3.4 Case Study Results

In this section, we present the results of our case study. For each research question, we present the motivation, our data processing and analysis approaches, and the results.

RQ1: What are the characteristics of reviewed code that is implicated in a crash?

Motivation. We intend to compare the characteristics of the reviewed patches that lead to crashes (Crash) with those that did not lead to crashes (Clean). Particularly, we want to know whether patch complexity, centrality, and developer participation in the code review process are correlated with the crash proneness of a reviewed patch. The result of this research question can help software organizations improve their code review strategy; focusing review efforts on the most crash-prone code.

Approach. We extract information from the source code to compute code complexity and SNA metrics and from issue reports to compute review metrics. Tables 3.1 to 3.3 provide descriptions of each of the studied metrics.

We assume that changes to complex classes are likely to lead to crashes because complex classes are usually more difficult to maintain. Inappropriate changes to complex classes may result in defects or even crashes. The SNA metrics are used to estimate the degree of centrality (see Section 3.3.3) of a class. Inappropriate changes to a class with high centrality may impact dependent classes; thus causing defects or even crashes. For each SNA metric, we compute the mean of all class values for the commits that fix an issue. Regarding the review metrics, we assume that patches with longer review duration and more review comments have higher risk of crash proneness. Since these patches may be more difficult to understand, although developers may have spent more time and effort to review and comment on them. We use the review activity metrics that were proposed by Thongtanunam et al. [12]. In addition, we also take *obsolete* patches into account because these patches were not approved by reviewers. The percentage of the obsolete patches that fix an issue can help to estimate the quality and the difficulty of the patches on an issue, as well as developer participation.

We apply the two-tailed *Mann-Whitney U test* [66] to compare the differences in metric values

Table 3.1 Code complexity metrics used to compare the characteristics of crash-inducing patches and clean patches.

Metric	**Description**	**Rationale**
Patch size	Mean number of lines of the patch(s) of an issue. We include context lines and comment lines because reviewers need to read all these lines to inspect a patch.	The larger the code changes, the easier it is for reviewers to miss defects [13].
Changed file number	Mean number of changed C/C++ files in the issue fixing commit(s).	If a change spreads across multiple files, it is difficult for reviewers to detect defects [13].
LOC	Mean number of the lines of code in the changed classes to fix an issue.	Large classes are more likely to crash [18].
McCabe	Mean value of McCabe cyclomatic complexity [62] in all classes of the issue fixing commit(s).	Classes with high cyclomatic complexity are more likely to lead to crashes [18].
Function number	Mean number of functions in all classes in the issue fixing commit(s).	High number of functions indicates high code complexity [63], which makes it difficult for reviewers to notice defects.
Maximum nesting	Mean of maximum level of nested functions in all classes in the issue fixing commit(s).	Code with deep nesting level is more likely to cause crashes [18].
Comment ratio	Mean ratio of the lines of comments over the lines of code in all classes of the issue fixing commit(s)	Reviewers may have difficulty to understand code with low ratio of comment [64], thus miss crash-prone code.

between crash-inducing patches and clean patches. We choose to use the Mann-Whitney U test because it is *non-parametric*, *i.e.*, it does not assume that metrics must follow a normal distribution. For the statistical test of each metric, we use a 95% confidence level (*i.e.*, $\alpha = 0.05$) to decide whether there is a significant difference among the two categories of patches. Since we will investigate characteristics on multiple metrics, we use the Bonferroni correction [67] to control the familywise error rate of the tests. In this work, we compute the adjusted p-value, which is multiplied by the number of comparisons.

For the metrics that have a significant difference between the crash-inducing and clean patches, we estimate the magnitude of the difference using *Cliff's Delta* [68]. Effect size measures report on the magnitude of the difference while controlling for the confounding factor of sample size [69].

To further understand the relationship between crash proneness and reviewer origin, we calculate the percentage of crash-inducing patches that were reviewed by Mozilla developers, external developers, and by both Mozilla and external developers. Previous work, such as [70], used the suffix of an email address to determine the affiliation of a developer. However, many Mozilla employees use an email address other than `mozilla.com` in Bugzilla, when they review code. To make our results more accurate, our collaborator, who is working at

Table 3.2 Social network analysis (SNA) metrics used to compare the characteristics of crash-inducing patches and clean patches. We compute the mean of each metric across the classes of the fixing patch(es) within an issue. *Rationale*: An inappropriate change to a class with high centrality value [59] can lead to malfunctions in the dependent classes; even cause crashes [18].

Metric	**Description**
PageRank	Time fraction spent to “visit” a class in a random walk in the call graph. If an SNA metric of a class is high, this class may be triggered through multiple paths.
Betweenness	Number of classes passing through a class among all shortest paths.
Closeness	Sum of lengths of the shortest call paths between a class and all other classes.
Indegree	Numbers of callers of a class.
Outdegree	Numbers of callees of a class.

Mozilla, used a private API to examine whether a reviewer is a Mozilla employee.

Results. Table 3.4 compares the reviewed patches that lead to crash (Crash) to those that do not crash (Clean). Statistically significant p-values and non-negligible effect size values are shown in bold. Figure 3.5 visually compares crash-inducing and clean patches on the metrics (after removing outliers because they can bury the median values), where there is a statistically significant difference and the effect size is not negligible. In this figure, the red bold line indicates the median value on the crash-inducing patches (or clean patches) for a metric. The dashed line indicates the overall median value of a metric. The width variation in each plot shows the variation of the data density.

For the code complexity metrics, crash-inducing patches have a significantly larger patch size, higher number of changed files, and higher comment ratio than clean patches. The magnitude of the differences on patch size and changed files is large; while the magnitude of the differences on comment ratio is small. This result implies that the related files of the reviewed patches that crash tend to contain complex code. These files have higher comment ratio because developers may have to leave more comments to describe a complicated or difficult problem. Our finding suggests that reviewers need to double check the patches that change complex classes before approving them. Investigators also need to carefully approve patches with intensive discussions because developers may not be certain about the potential impact of these patches.

In addition, crash-inducing patches have significantly higher centrality values than clean patches on all of the social network analysis metrics. The magnitude of closeness and outdegree is negligible; while the magnitude of PageRank, betweenness, and indegree is small. This result suggests that the reviewed patches that have many other classes depending on

Table 3.3 Review metrics used to compare the characteristics of crash-inducing patches and clean patches. We compute the mean metric value across the patches within an issue.

Metric	**Description**	**Rationale**
Review iterations	Number of review flags on a reviewed patch.	Multiple rounds of review may help to better identify defective code than a single review round [12].
Number of comments	Number of comments related with a reviewed patch.	Review with a long discussion may help developers to discover more defects [12].
Comment words	Number of words in the message of a reviewed patch.	
Number of reviewers	Number of unique reviewers involved for a patch.	Patches inspected by multiple reviewers are less likely to cause defects [65].
Proportion of reviewers writing comments	Number of reviewers writing comments over all reviewers.	Reviews without comments have higher likelihood of defect proneness [9, 12].
Negative review rate	Number of disagreement review flags over all review flags.	High negative review rate may indicate a low quality of a patch.
Response delay	Time period in days from the review request to the first review flag.	Patches that are promptly reviewed after their submission are less likely to cause defects [65].
Review duration	Time period in days from the review request until the review approval.	Long review duration may indicate the complexity of a patch and the uncertainty of reviewers on it, which may result in a crash-prone patch.
Obsolete patch rate	Number of obsolete patches over all patches in an issue.	High proportion of obsolete patch indicates the difficulty to address an issue, and may imply a high crash proneness for the landed patch.
Amount of feedback	Quantity of feedback given from developers. When a developer does not have enough confidence on the resolution of a patch, she would request for feedback prior to the code review.	The higher the amount of feedback, the higher the uncertainty of the patch author, which can imply a higher crash proneness.
Negative feedback rate	Quantity of negative feedback over all feedback.	High negative feedback rate may imply high crash proneness for a patch.

them are more likely to lead to crashes. Reviewers need to carefully inspect the patches with high centrality.

Regarding the review metrics, compared to clean patches, crash-inducing patches have significantly higher number of comments and comment words. This finding is in line with the results in [13], where the authors also found that the number of comments have a negative impact on code review quality. The response time and review duration on crash-inducing patches tend to be longer than clean patches. These results are expected because we assume that crash-inducing patches are harder to understand. Although developers spend a longer time and comment more on them, these patches are still more prone to crashes. In terms of the magnitude of the statistical differences, crash-inducing and clean patches that have been

Table 3.4 Median metric value of crash-inducing patches (Crash) and clean (Clean) patches, adjusted p-value of Mann-Whitney U test, and Cliff's Delta effect size.

Metric	**Crash**	**Clean**	***p*-value**	**effect size**
		Code complexity metrics		
Patch size	406	111	<**0.001**	**0.53 (large)**
Changed files	4.8	2.0	<**0.001**	**0.49 (large)**
LOC	1259.3	1124.5	0.2	–
McCabe	3.0	3.0	0.5	–
Function number	45.8	43.0	0.3	–
Maximum nesting	3.0	3.0	1	–
Comment ratio	0.3	0.2	<**0.001**	**0.24 (small)**
		Social network analysis metrics		
PageRank	4.4	3.2	<**0.001**	**0.17 (small)**
Betweenness	50,743.5	22,011.3	<**0.001**	**0.16 (small)**
Closeness	2.2	2.1	<**0.001**	0.12 (negligible)
Indegree	12.0	7.5	<**0.001**	**0.15 (small)**
Outdegree	27.3	26.0	**0.02**	0.05 (negligible)
		Review metrics		
Review iterations	1.0	1.0	**0.001**	0.03 (negligible)
Number of comments	0.5	0	<**0.001**	**0.15 (small)**
Comment words	2.5	0	<**0.001**	**0.16 (small)**
Number of reviewers	1.0	1.0	1	–
Proportion of reviewers writing comments	1	1	<**0.001**	0.10 (negligible)
Negative review rate	0	0	**0.03**	0.01 (negligible)
Response delay	14.2	8.1	<**0.001**	0.14 (negligible)
Review duration	15.2	8.2	<**0.001**	**0.15 (small)**
Obsolete patch rate	0	0	1	–
Amount of feedback	0	0	**0.03**	0.02 (negligible)
Negative feedback rate	0	0	1	–

reviewed only have a small effect size on number of comments, comment words, and review duration; while the effect sizes of other statistical differences are negligible.

Table 3.5 shows the percentage of the patches that were reviewed by Mozilla developers, external developers, and by both Mozilla and external developers. Regarding the crash-inducing rate of the studied patches, the patches reviewed by both Mozilla and external developers lead to the highest rate of crashes (5.9%). On the one hand, there are few patches that were reviewed by both Mozilla and external developers, this result may not be representative. One the other hand, Mozilla internal members and external community members do not have the same familiarity on a specific problem, such collaborations may miss some crash-prone changes. We suggest patch authors to choose reviewers with the same level of familiarity on the changed module(s) and the whole system. In the future, we plan to further investigate the relationship between crash proneness and the institution that the reviewers represent.

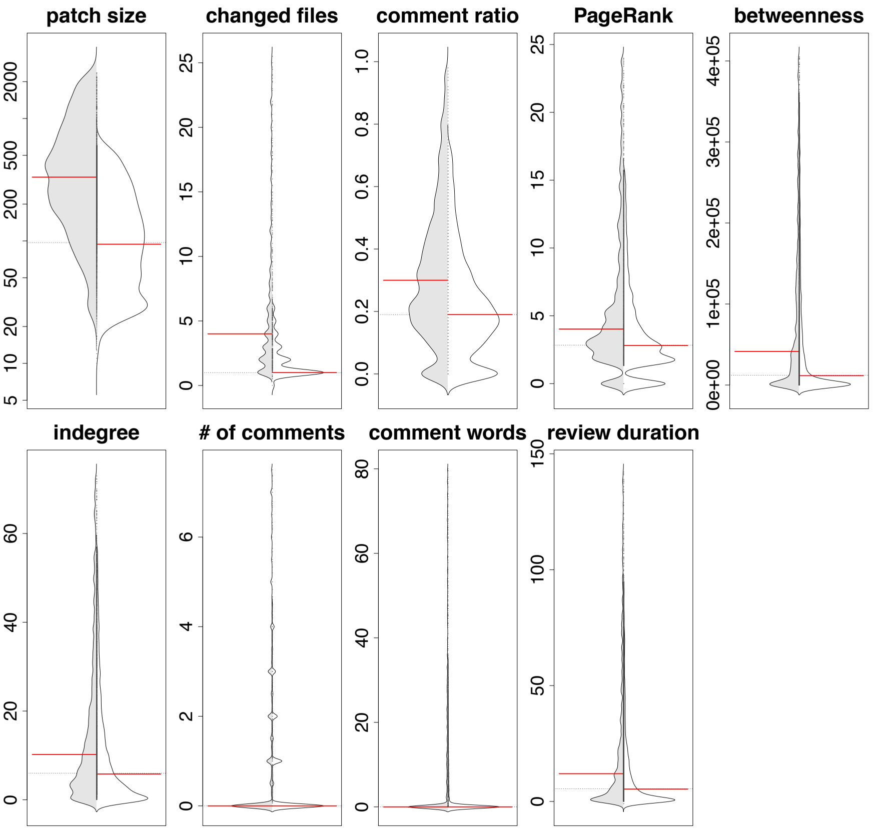


Figure 3.5 Comparison between crash-inducing patches (left part, grey) *vs.* clean patches (right part, white). Since we removed outliers from the plots, the median values may not correspond to the values in Table 3.4, which includes the outliers.

RQ2: Why did reviewed patches crash?

Motivation. In **RQ1**, we compared the characteristics of reviewed code that crashes with reviewed code that does not crash. To more deeply understand why reviewed patches can still lead to crashes, we perform a qualitative analysis on the purposes of the reviewed patches that crash and the root causes of their induced crashes.

Approach. To understand why developers missed the crash-inducing patches, we randomly sample 100 out of the 1,202 issues that contain reviewed patches that crash. If we use a confidence level of 95%, our sample size corresponds to a confidence interval of 9%. Inspired by Tan et al.'s work [71], we classify the purposes of patches (patch reasons) into 13 categories based on their (potential) impact on users and detected fault types. The "incorrect functionality" category defined by Tan et al. is too broad, so we break it into more detailed patch reasons: "incorrect rendering", "(other) wrong functionality", and "incompatibility".

Table 3.5 Origin of the developers who reviewed clean patches and crash-inducing patches.

Origin	**Total**	**Crash**	**Crash rate**
Mozilla	38,481	1,094	2.8%
External	2,512	55	2.2%
Both	897	53	5.9%
Total	41,890	1,202	2.9%

Table 3.6 Patch reasons and descriptions (abbreviation are shown in parentheses).

Reason	**Description**
Security	Security vulnerability exists in the code.
Crash	Program unexpectedly stops running.
Hang	Program keeps running but without response.
Performance degradation (perf)	Functionalities are correct but response is slow or delayed.
Incorrect rendering (rendering)	Components or video cannot be correctly rendered.
Wrong functionality (func)	Incorrect functionalities besides rendering issues.
Incompatibility (incompt)	Program does not work correctly for a major website or for a major add-on/plug-in due to incompatible APIs or libraries, or a functionality, which was removed on purpose, but is still used in the wild.
Compile	Compilation errors.
Feature	Introduce or remove features.
Refactoring (refactor)	Non-functional improvement by restructuring existing code without changing its external behaviour.
Improvement (improve)	Minor functional or aesthetical improvement.
Test-only problem (test)	Errors that only break tests.
Other	Other patch reasons, *e.g.*, data corruption and adding logging.

In addition, since we do not only study defect-related issues as Tan et al. [71], we add more categories about the reason of patches, such as “refactoring”, “improvement”, and “test-only problem”. Table 3.6 shows the patch reasons used in our classification. We conduct a card sorting on the sampled issues with the following steps: 1) examine the issue report (the title, description, keywords, comments of developers, and the patches). Two graduate students individually classified each issue into one or more categories; 2) created an online document to compare categories and resolved conflicts through discussions; 3) discussed each conflict until a consensus was reached.

Then, from the results of the SZZ algorithm, we find the crash-related issues caused by the patches of the sampled issues. Following the same card sorting steps, we classify the root causes of these crash-related issues into five categories, as shown in Table 3.7.

Table 3.7 Crash root causes and descriptions.

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

Results. Figure 3.6 shows the distribution of patch reasons obtained from our manual classification. Among the reviewed patches that lead to crashes, we find that most patches are used for improving Firefox' performance, refactoring code, fixing previous crashes, and implementing new features. These results imply that: 1) improving performance is the most important purpose of the reviewed patches that crash; 2) some "seemingly simple" changes, such as refactoring, may lead to crashes; 3) fixing crash-related issues can introduce new crashes; 4) many crashes were caused by new feature implementations. The classification suggests that reviewers need to scrutinize patches due to the above reasons, and software managers can ask a *super review* inspection for these types of patches.

Figure 3.7 shows the distribution of our manually classified root causes. According to the results, most crashes are due to memory and semantic errors. To further understand the detailed causes of the memory errors, we found that 61% of these errors are as a result of null pointer dereferences. By studying the issue reports of the null pointer crashes, we found that most of them were eventually fixed by adding check for NULL values, *e.g.*, the issue #1121661.[3] This finding is interesting because some memory faults can be avoided by static analysis. Mozilla has planned to use static analysis tools, such as *Coverity* [72] and *Clang-tidy* [73], to enhance its quality assurance. We suggest that software organizations can perform static analysis on a series of memory faults, such as null pointer dereference and memory leaks, prior to their code review process. Our results suggest that static code analysis can not only help to mitigate crashes but also certain security faults. Even though the accuracy of the static analysis cannot reach 100%, it can help reviewers to focus their inspection efforts on suspicious patches. In addition, semantic errors are also an important root cause of crashes. Many of these crashes are eventually resolved by modifying the `if` conditions of the faulty code. Semantic errors are relatively hidden in the code, we suggest

[3] `https://bugzilla.mozilla.org/show_bug.cgi?id=1121661#c1`

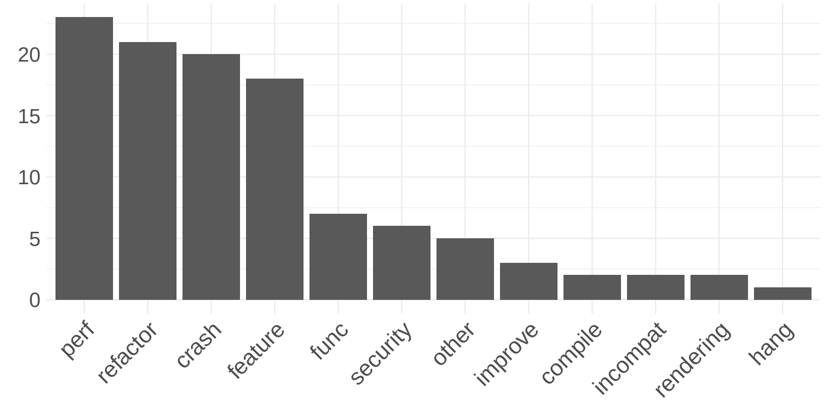


Figure 3.6 Distribution of the purposes of the reviewed issues that lead to crashes.

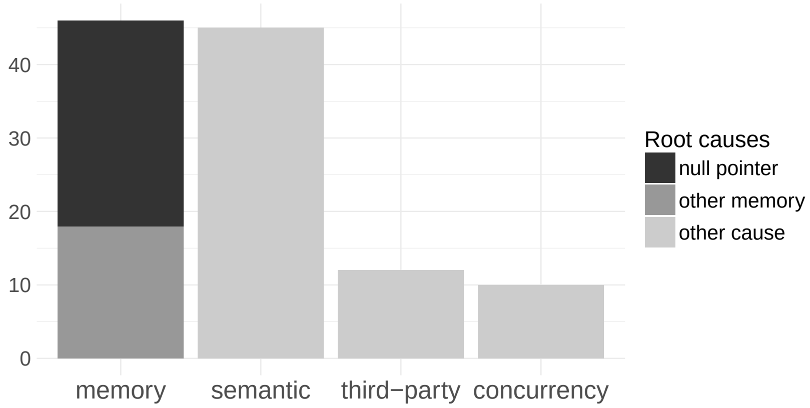


Figure 3.7 Distribution of the root causes of the reviewed issues that lead to crashes.

reviewers to focus their inspections on changes of control flow, corner cases, and exception handling to prevent potential crashes. Software organizations should also enhance their testing effort on semantic code changes.

3.5 Threats to Validity

Internal validity threats are concerned with factors that may affect a dependent variable and were not considered in the study. We choose steady periods for the studied commits by analyzing the distribution of crash-inducing commit numbers. We eliminate the periods where the numbers of crash-inducing commits are relatively low because some crash-inducing code has not been filed into issues at the beginning and at the end of our collected data.

The SZZ algorithm is a heuristic to identify commits that induce subsequent fixes. To mitigate the noise introduced by this heuristic, we removed all candidates of crash-introducing commits that only change comments or whitespace. We validate the accuracy of the algorithm by comparing changed files of a crash-inducing commit with the information in its corresponding crash-related issue report. As a result, 68.1% of our detected crash-inducing commits changed at least one file mentioned in the crashing stack trace or comments of their corresponding issues. The remaining commits might change a dependent class of the code in the stack trace, or developers do not provide any stack trace in their corresponding issue reports. Therefore, we believe that the SZZ algorithm can provide a reasonable starting point for identifying crash-prone changes.

Finally, in **RQ1**, we use some time-related metrics (*e.g.*, review duration), which measures the period since a review for a patch was requested until the patch was approved. Although a review duration of two months does not mean that developers really spent two months to review a patch, it can reflect the treatment time of a development team (including pending time, understanding time, and evaluation time) to the patch. For example, when the review queue of a reviewer is long, her assigned patches may be pending for a long time before she begins to inspect them [74].

Conclusion validity threats are concerned with the relationship between the treatment and the outcome. We paid attention not to violate the assumptions of our statistical analyses. In **RQ1**, we apply the non-parametric test, the Mann-Whitney U test, which does not require that our data be normally distributed.

In our manual classifications of root causes of the reviewed patches that crashes, we randomly sampled 100 reviewed issues and the crashes that were induced. Though a larger sample size might yield more nuanced results, our results clearly show the most crash-prone types of patches, and the major root causes of the reviewed patches that crash.

Reliability validity threats are concerned with the replicability of the study. To aid in future replication studies, we share our analytic data and scripts online: `https://github.com/swatlab/crash_review`.

External validity threats are concerned with the generalizability of our results. In this work, we study only one subject system, mainly due to the lack of available crash reports and code review data. Thus, our findings may not generalize beyond this studied system. However, the goal of this study is not to build a theory that applies to all systems, but rather to empirically study the relationship between review activities and crash proneness. Nonetheless, additional replication studies are needed to arrive at more general conclusions.

3.6 Summary

The code review process helps software organizations to improve their code quality, reduce post-release defects, and collaborate more effectively. However, some high-impact defects, such as crash-related defects, can still pass through this process and negatively affect end users. In this chapter, we compare the characteristics of reviewed code that induces crashes and clean reviewed code in Mozilla Firefox. We observed that crash-prone reviewed code often has higher complexity and centrality, *i.e.*, the code has many other classes depending on it. Compared to clean code, developers tend to spend a longer time on and have longer discussions about the crash-prone code; suggesting that developers may be uncertain about such patches (RQ1). Through a qualitative analysis, we found that the crash-prone reviewed code is often used to improve performance of a system, refactor source code, fix previous crashes, and introduce new functionalities. Moreover, the root causes of the crashes are mainly due to memory and semantic errors. Some of the memory errors, such as null pointer dereferences, could be likely prevented by adopting a stricter organizational policy with respect to static code analysis (RQ2). In the future, we plan to investigate to which extent static analysis can help to mitigate software crashes. We are also contacting other software organizations in order to study their crash reports to validate the results obtained in this work.

CHAPTER 4 AN EMPIRICAL STUDY OF PATCH UPLIFT IN RAPID RELEASE DEVELOPMENT PIPELINES[*]

Patch uplift is risky because the time allowed for the stabilization of uplifted patches is reduced by six weeks for each skipped channel.

The advent of continuous delivery and rapid release practices have significantly reduced the amount of stabilization time available for new features, forcing companies to resort to innovative techniques to ensure that important features are released to the public, in a timely manner and with a good quality. To cope with short release cycles, Mozilla has re-organized its release process around four channels: a development channel named *Nightly*, two stabilization channels (*Aurora* and *Beta*), and a main *Release* channel. Features corresponding to a new release are developed on the Nightly channel over a period of six weeks. After that, the code is transferred to Aurora, where it is tested by Mozilla developers and contributors, for a period of six weeks, and then to Beta where it is tested by a selected group of external users. Finally, mature Beta features are imported into the main Release channel and delivered to end users. This pipelined process allows Mozilla to avoid mixing the development of new features with the stabilization process, which is particularly important given that integration operations are unpredictable [75], and can significantly delay a release process, if not enough time is allowed for stabilization. However, this well organized release process is frequently subverted by urgent patches, implementing high-value features or critical fixes, that cannot wait for the next release train. These features and fixes are directly promoted from the development channel to stable channels (*i.e.*, Aurora, Beta, and main Release), a practice called *patch uplift*. Patch uplift is risky because the time allowed for the stabilization of uplifted patches is reduced by six weeks for each skipped channel. Therefore, it is important to carefully pick the patches that are uplifted and ensure that developers scrutinize them properly, to reduce the risk of regressions. There are a set of rules in place at Mozilla to govern this uplift process. One of these rules is that patches uplifted to the Beta channel should be *(1) ideally reproducible by the QA team, so that they can be verified; (2) should have been verified on Aurora/Nightly first; and (3) should not contain string changes (i.e., changes in the text which is visible to users).* However, despite these rules, multiple uplifted patches still introduce regressions in the code. Hence, it is unclear if–and–how the rules are

[*] Part of the content of this chapter is published in "An Empirical Study of Patch Uplift in Rapid Release Development Pipelines", Marco Castelluccio, Le An, and Foutse Khomh, *Empirical Software Engineering (EMSE)*, DOI=`10.1007/s10664-018-9665-y`.

enforced at Mozilla and why certain uplifted patches introduce post-release bugs.

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

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

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

RQ2: *How effective are uplift operations?*

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

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

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

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

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

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

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

Chapter Overview

Section 4.1 provides background information about patch uplift. Section 4.2 describes the design of our case study. Section 4.3 presents the results of the case study. Section 4.4 discusses threats to the validity of this study, and Section 4.5 summarizes the chapter.

4.1 Mozilla Patch Uplift Process

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

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

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

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

The more a channel is stable, the higher is the bar for approval of uplift requests. Below we present an excerpt of the rules in place at Mozilla on the different channels.

- *Aurora*: Uplifts to the Aurora channel are less critical, as they still have considerable time for stabilization. The rules are not strict in this case: no new features are accepted; no disruptive refactorings; no massive code changes; no string changes, unless the localization team is aware and has approved; they must be accompanied, if possible, by automated tests.

- *Beta*: Uplifts to the Beta channel are more critical, as they have less time for stabilization. In addition to the rules outlined for Aurora, the changes uplifted to the Beta channel should be (1) ideally reproducible by QA, so that they can be verified; (2) they should have been verified on Aurora/Nightly first; and should not contain (3) changes to the user-visible strings in the application (as those require a very high effort and time to be localized, since Mozilla relies on volunteer contributors). The uplifted changes can be proven performance improvements, fixes to important crashes, fixes for recent regressions. The closer to the release date, the stricter the release managers should be in enforcing the rules.

- *Release*: Uplifts to the Release channel are generally discouraged, as they require a new version to be built and released to users. Possible uplifts are fixes for major top crashes, security issues, functional regressions with a very broad impact.

Once a patch is accepted for uplift, Tree Sheriffs [53] (*i.e.*, engineers responsible for supporting developers in committing patches and ensuring that the automated tests are not broken after commits, monitoring intermittent failures and backing out patches in case of test failures) or the developers themselves can commit it to the stabilization channel(s) for which the patch was approved.

4.2 Case Study Design

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

4.2.1 Data Collection

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

Mozilla developers use customized Bugzilla flags to request for patch uplifts. These flags have the form `approval-mozilla-CHANNEL`, where `CHANNEL` can be Aurora, Beta, or Release. The postfix of the flag is set to a question mark (`?`) when a developer asks for an uplift, to a minus sign (`-`) if the release manager rejects the uplift, and to a plus sign (`+`) if the release manager approves the uplift. We relied on these flags to identify uplifted patches. At Mozilla, release managers usually inspect all patches in an issue report before deciding whether they can be uplifted together. Thus, in this work, we considered uplift characteristics at the issue level. If an issue contains multiple patches, we bundled the patches together. To study the patch uplift process, we need to consider a period of time during which the practice was well established at Mozilla. To decide on this period, we computed the amount of patches that were uplifted each month, over our initial period of July 2014 to August 2016. Figure 4.1 shows the distribution of the number of uplifts in three Firefox's release channels during this period. We did not consider uplifts that concern the "Pocket" component, as the inclusion of Pocket (which is a third-party add-on) in Firefox, a one-time event, might introduce noise in our data. In Figure 4.1, each time point represents a period of one month (we can see that the Release channel did not receive any uplift in May and November 2015). Figure 4.1 shows that the number of uplifted patches increased from July 2014 to August 2014 and then became stable from September 2014 to August 2016. Based on this distribution, we selected the period between September 2014 and August 2016, for our study. In other words, we limited our dataset to only issue reports and commits that occurred within this period. Between September 2014 and August 2016, we study in total 33,664 issue reports, in which there are 7,267 uplift requests: 285 to Release, 2,614 to Beta, and 4,368 to Aurora.

4.2.2 Data processing

Figure 4.2 shows a general overview of our approach. We describe each step of the approach below. The corresponding data and scripts are available online at: `https://github.com/swatlab/uplift-analysis`.

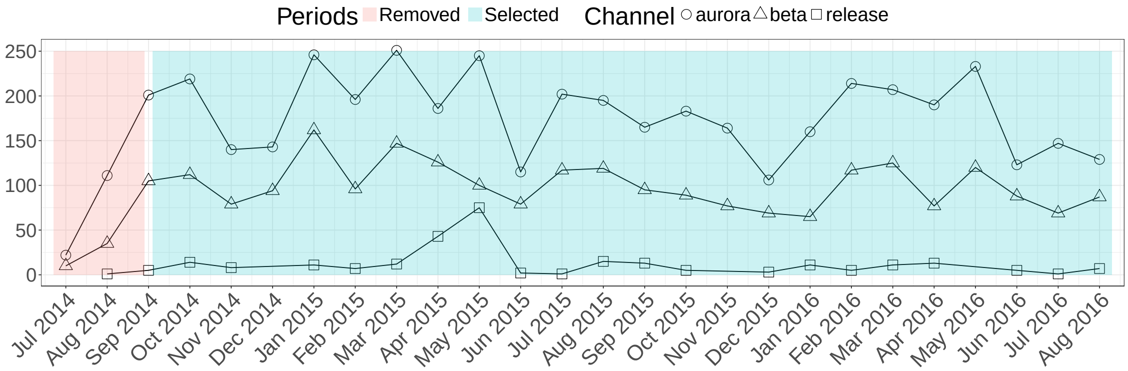


Figure 4.1 Number of uplifts during each month from July 2014 to August 2016. Periods with low number of uplifts or not covering all the three channels are removed.

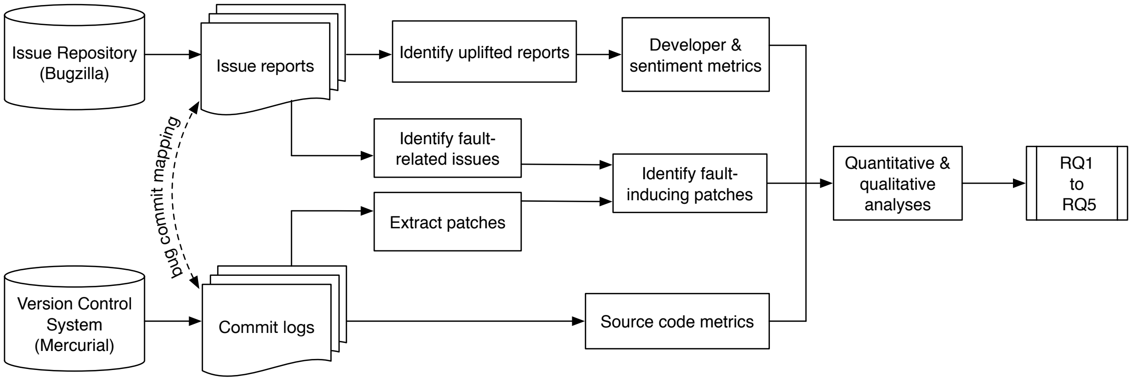

Issue Repository (Bugzilla)
Issue reports
Identify uplifted reports
Developer & sentiment metrics
Identify fault-related issues
Identify fault-inducing patches
Extract patches
Quantitative & qualitative analyses
RQ1 to RQ5
bug commit mapping
Version Control System (Mercurial)
Commit logs
Source code metrics

Figure 4.2 Overview of our data processing approach.

Identification of Fault-related Issues

Mozilla uses Bugzilla to manage and track its issues. All types of issues, whether they are faults or new features, are managed in this system. Unlike JIRA [76], which offers the possibility to distinguish between issues using a tag, Bugzilla does not provide issue type information. Therefore, our first processing task is to differentiate issues that are related to faults, from new feature requests or improvements. To automatically identify fault-related issues, we used a keyword-based heuristic to search information in the title, description, flags, and user comments of each issue report. Our list of keywords includes: crash, regression, failure, leak, steps to reproduce (STR), and hang. The full list is available at: `https://github.com/swatlab/uplift-analysis`.

To ensure the accuracy of our detection on fault-related issues, we manually validated a sample of our results. From a total of 33,664 issue reports, we randomly selected a sample of 380 issue reports, which corresponds to a confidence level of 95% and a confidence interval of

5%. The first and the second authors read each of the 380 issue reports independently and classified them into *fault-related* and *other* categories. We then compared their classification results and observed that 41 issue reports were classified into different categories by the two authors. To resolve these discrepancies, we created an online document for the 41 issues; allowing all of the authors to comment and discuss the issues. After this round, a consensus was reached for 35 out of the 41 issues. For the remaining 6 issues, we organized a meeting and discussed the classification of each of them until a consensus was found. The result of our manual classification shows that our keyword-based heuristic achieves a precision of 87.3% and a recall of 78.2%, when classifying issues into *fault-related* (the true class) and *other* (the false class) categories.

Identification of Fault-inducing Patches

We used the SZZ algorithm [22] to identify patches (these patches could be fault-fixing patches or patches related to features or improvements) that introduced faults in the system. First, we used Fischer et al.'s heuristics [55] to map each studied issue to its corresponding patch(es) (*i.e.*, commits). This heuristic consists in looking for issue IDs in commit messages using regular expressions. Next, for each fault-related issue, we used the following Mercurial command to extract the list of files that were changed to fix the issue:

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

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

```
hg annotate commitˆ -r f_fix -c -l -w -b -B
```

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

Identification of Duplicate Issues

There has not been an approach that can identify duplicate issues[1] with 100% accuracy. Two general threads of approaches were proposed in previous works. The first thread of approaches ranks the similarity between one given issue and other issues in a dataset, such as [77, 78, 79]. The other thread predicts whether two given issue reports are duplicate or not, such as [80, 81, 82]. Inspired by these works, we designed the following approach, which is customized for our dataset.

1. For each subject issue report, we extracted its short description (*i.e.*, title) and long description (*i.e.*, first comment). We performed stemming and stop word removal against these raw texts.

2. As [81, 82], we used *Okapi BM25* algorithm [83] (referred as BM25 in the rest of the chapter) to rank of the similarity between any pair of issues: $\{(issue_i,\ issue_j) \mid i \neq j,\ issue_i \in uplift\ bugs,\ issue_j \in all\ bugs\}$. In a given pair of $(issue_i,\ issue_j)$, we respectively calculated the similarity between their titles and their descriptions. As there are in total 33,664 studied issues and 4,958 unique uplifted issues[2], we should perform $(33664 - 4958) \times 4958 + 4958 \times (4958 - 1) \approx 167M$ comparisons (for titles and descriptions respectively). In each of these comparisons, the BM25 algorithm yields a score of similarity, the higher the score the closer the two pieces of information (*i.e.*, titles or descriptions).

3. We ranked the BM25 scores for all pairs of issues by descending order. We removed the pairs where the BM25 scores is 0. The rest of the results were considered as duplicate issue candidates. We intended to manually examine the correctness of each title (respectively description) pair by carefully analyzing the whole issue reports. There are 8.1 million pairs of duplicate issues candidates, our manual validation cannot cover all these but can only focus on the most likely candidates. First, we narrowed down our manual analysis scope to the top 1,000 candidates because correct duplicate cases can hardly be observed beyond the top 1,000 candidates (in which the highest BM25 value is 97.5, and the lowest value is 29.1) through a preliminary analysis. Second, we designed a heuristic to further filter out the pairs in which the two issue reports are not linked to each other: if Issue A is never mentioned in Issue B (either in one of the comments, or in the “Blocking”, “Depends On”, “See Also” fields), we considered the two issues to be “not linked” (meaning that, in practice, developers did not notice any

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

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

relationship between the issues). To calculate the false positive rate of this heuristic, we manually examined the top 50 and 100 other randomly selected candidates, and found that only two correct duplicate pairs were misclassified by the "unlinked" heuristic. As a result, 137 candidates survived this step. Our manual validation was then performed on these candidates.

4. Since we separately performed Steps 2 and 3 on the issue titles and descriptions, we combined the results and removed redundancies. We also removed the pairs where an issue is a clone of another one. From the obtained results, we only keep the duplicate pairs where the duplicate issue were opened or resolved after the original patch had been uplifted.

Compared to any fully automated approach, our approach can achieve a very high precision because all of the reported duplicate issue pairs have been carefully examined by two graduate students (by understanding the whole context of the issue reports). Although we cannot guarantee a 100% recall, we believe that our reported results covers all possible cases where the titles (respectively descriptions) of a pair of issues are textually similar to each other. In fact, text processing is the base of most aforementioned approaches. BM25 is considered as an advanced measure of ranking similarities, which has a higher performance than the traditional TF-IDF algorithm [82]. Some approaches, such as [81, 82], used additional information (*e.g.*, priority, product, and version fields from the analyzed issue reports), but such information cannot help to retrieve more possible candidate (*i.e.*, it cannot increase the recall). In this work, we only ignored the issue pairs where the titles or descriptions have no relevance (*i.e.*, BM25 value is 0) or have little relevance (*i.e.*, the two issues are not linked and the BM25 value is weak).

Mining Issue Reports

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

Computing Metrics

To capture the characteristics of patches that were uplifted, we computed the 22 metrics described in Tables 4.1 to 4.5. These metrics correspond to the following five dimensions:

Table 4.1 Developer experience and participation metrics (m_1 - m_5).

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

Table 4.2 Uplift process metrics (m_6 - m_8).

Metric	m_i	**Description**	**Type and range**
Landing delta	6	Time elapsed (in days) between when the patch was applied to the Nightly version and when the developer asked for approval of an uplift. The value can be negative, as sometimes developers request uplift before their patch is applied to Nightly.	Float, from -41.59 to around 153.73.
Response delta	7	Time elapsed (in days) between when the developer asked for approval for the uplift and when the release manager decided (approved or rejected).	Float, from 0.0 to around 31.23.
Release delta	8	Time elapsed (in days) between when the developer asked for approval for the uplift and the date of the following release.	Float, from 0.0 to around 42.76.

Developer experience and participation metrics Our rationale for computing these metrics is that patches written or reviewed by experienced developers may have a higher chance to be accepted for uplift, and may be less fault-prone. Long comments and long review durations may indicate the complexity of an issue and developers' uncertainty about it, which may explain its rejection or fault-proneness.

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

Table 4.3 Sentiment metrics (m_9 - m_{10}).

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

Table 4.4 Code complexity metrics (m_{11} - m_{19}).

Metric	m_i	Description	Type and range
Patch size	11	Number of lines in a patch (excluding test patches).	Integer, from 0 to 301114.
Test patch size	12	Number of lines in a test patch.	Integer, from 0 to 127155.
Prior changed times	13	Number of previous commits that modified the same files that the patch is modifying.	Integer, from 0 to 114051.
LOC	14	Average lines of code in all files in a patch.	Integer, from 0 to 27727.
Average cyclomatic	15	Average cyclomatic complexity of the functions in a file.	Integer, from 0 to 128.
Number of functions	16	Average number of files' functions in a patch.	Integer, from 0 to 3878.
Maximum nesting	17	Average maximum level of nested functions in all files in a patch.	Integer, from 0 to 13.
Comment ratio	18	Average ratio of the lines of comments over the total lines of code in all files in a patch.	Integer, from 0 to 99.
Module number	19	Number of modules (as defined by Mozilla in [84]) involved by a patch.	Integer, from 0 to 76.

Sentiments We computed sentiment metrics because we believe that sentiments can affect uplift decisions and their success rate. From each studied issue, we extract developers' comments to compute their sentiments. We leverage the sentiment mining tool, *SentiStrength* [85], to estimate the extent of developers' positive and negative sentiments toward a specific issue. As one of the state-of-the-art sentiment mining tool, SentiStrength is easy to apply, and it has achieved a reasonable performance in prior works [85, 86]. To adapt this tool to the software engineering context, we ignored a group of words that have negative meanings in general but do not represent any negative sentiment in Bugzilla discussions, *e.g.*, *bug*, *error*, *issue*, *regression*, *failure*, *fail*, *leak*, *crash*[3]. To further filter out irrelevant information from the comments, we used regular expressions to ignore hyperlinks and referred texts (*i.e.*, lines starting with ">"). In addition to developers' sentiments, we also computed module owners' sentiments.

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

Table 4.5 Code centrality (SNA) metrics (m_{20} - m_{22}).

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

Code Complexity Previous works, such as [18], have shown that complex code is likely to introduce faults. We calculated code complexity metrics to understand how uplifting decisions and their success are affected by the complexity of the uplifted patches. We extracted the files changed in each patch and use the static code analysis tool *Understand* [57] to calculate the following complexity metrics on the files: lines of code (LOC), average cyclomatic complexity, number of functions, maximum nesting, and ratio of the comment lines over the total code lines.

Code centrality (SNA) metrics Kim et al. [18] observed that functions close to the centre of a call graph are likely to experience more faults. Hence, we computed metrics capturing the centrality of functions involved in uplifted patches and uplifted patch candidates. We used the network analysis tool, *igraph* [61], in combination to *Understand* [57], as in [87], to compute the following *Social Network Analysis* (SNA) metrics: PageRank, betweenness, and closeness. When computing complexity and SNA metrics, we only considered the C/C++ code since Firefox contains 86% of C/C++ code. Computing code complexity and SNA metrics is a very time-consuming task. Instead of computing the metrics for each patch, we computed metrics by releases and map a given patch to its latest major release as in our previous work [87]. To make the metric results as precise as possible, we considered all major releases from Firefox 32.0 until Firefox 48.0, which cover the system's history from September 2014 until August 2016.

4.3 Case Study Results

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

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

Motivation. This question aims to understand the characteristics of patches that are uplifted. We are particularly interested in understanding what differentiates patch uplifts among different channels. Although Mozilla has published rules to guide the patch uplift process [88], it is unclear if and how these rules are enforced in practice. The answer to this research question can help discover hidden factors that affect the uplift process, and help software practitioners make this process more predictable.

1) Quantitative Analysis

Approach. Using the metrics from Tables 4.1 to 4.5, we statistically compared 22 numerical characteristics of patch uplift candidates that were accepted and those that were rejected. As Mozilla release managers take a whole issue report into account during the uplift process (see Section 4.2.1), we calculated the average values of the code complexity and SNA metrics for all patches in a subject issue report.

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

We used the Mann-Whitney U test [66] to accept or reject these hypotheses. The Mann-Whitney U test is a non-parametric statistical test that measures whether two independent distributions have equally large values. We used a 95% confidence level (*i.e.*, $\alpha = 0.05$) to accept or reject the hypotheses. Since we performed more than one comparison on the same dataset, to reduce the chances of obtaining false-positive results, we used Bonferroni correction [67] to control the familywise error rate. Concretely, we calculated the adjusted p-value, which is multiplied by the number of comparisons. Whenever we obtained statistically significant differences between metric values, we computed the Cliff's Delta effect size [89] to measure the magnitude of the difference. Given a result of the Cliff's Delta, d, we use the following thresholds to decide its magnitude: $|d| < 0.147$ "negligible", $|d| < 0.33$ "small", $|d| < 0.474$ "medium", otherwise "large" [90]. In the following, we report only the metrics for which there is a statistically significant difference between accepted and rejected patch uplift candidates.

Results. Table 4.6 summarizes differences between the characteristics of patches that were accepted for an uplift and those that were rejected. We show the median value of accepted and rejected uplifts for each metric, as well as the p-value of the Mann Whitney U test and the effect size. For all three channels, rejected uplifts have longer response delta (m_7) than accepted uplifts. We attribute this outcome to the high complexity of the rejected patches,

Table 4.6 Accepted vs. rejected patch uplift candidates.

Channel	Metric	Accepted	Rejected	*p*-value	Effect size
Aurora	Comment ratio	0.1	0.2	**0.03**	small
	Landing delta	0.4	3.0	**0.02**	small
	Response delta	0.9	2.4	**<0.001**	medium
Beta	LOC	529.0	1,046.8	**<0.001**	small
	Cyclomatic	2.0	3.0	**0.04**	negligible
	# of functions	20.0	35.2	**<0.001**	small
	Comment ratio	0.1	0.2	**<0.001**	small
	Betweenness	2,789.0	20,586.3	**0.01**	negligible
	PageRank	1.4	1.7	**0.01**	negligible
	Max. nesting	2.3	3.0	**7.72e-03**	negligible
	Module number	1.0	1.0	**7.13e-03**	negligible
	Response delta	0.7	1.0	**<0.001**	small
Release	Response delta	0.02	3.1	**<0.001**	large

which required longer time for risk assessment. We summarize the different results among the channels as follows:

- *Aurora*: We observed that rejected uplift requests have significantly higher landing delta; this might imply that the rejected patches are landing at the end of the Aurora cycle, and so have less time for stabilization. Also, rejected uplift requests have higher ratio of comment in the source code, although we expected that a higher comment ratio might help release managers understand the code. A high comment ratio could also indicate a high code complexity. Release managers may hesitate to release patches with complex code ahead of schedule.

- *Beta*: Compared to accepted patches, rejected patches tend to have higher code complexity in terms of LOC and number of functions, as well as higher SNA values in terms of PageRank. This result is expected, because we assume that complex code and code connected with many other classes is less likely to be accepted for urgent releases. As in the Aurora channel, rejected patches also contain code with higher ratio of comment. Although accepted and rejected patches have significant differences on some other metrics such as cyclomatic complexity, the magnitude of these differences is negligible.

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

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

2) Qualitative Analysis

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

Approach. From 2,384 uplifted issues in the Beta channel and 231 uplifted issues in the Release channel, we randomly chose respectively 459 and 154 issues as our samples (which correspond to a confidence level of 95% and a confidence interval of 5%). Inspired by Tan et al.'s work [71], we classified the uplift reasons into 14 categories based on their (potential) impact and detected fault types. Some of Tan et al.'s categories are too broad, such as incorrect functionality. We broke them into more detailed uplift reasons, *e.g.*, incorrect functionality is split to incorrect rendering and (other) wrong functionality. Some of Tan et al.'s categories, such as data corruption, are with too few occurrences. We combined them into the "other" category. Table 4.7 shows the uplift reasons used in our classification. We performed a card sorting on each of the sampled issues. By studying the issue report, two graduate students individually classified each issue into one or multiple uplift reasons (some uplift may be due to multiple reasons). Then we compared their classifications and resolved conflicts through discussions. We discussed each conflict until an agreement was reached.

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

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

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

After this first more structured interview with the questions above, we performed a semi-

Table 4.7 Uplift reasons and descriptions (abbreviations are shown in parentheses).

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

structured one, showing the results of our quantitative analysis to the release managers and asking them for their feedback.

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

Results. Figures 4.3 and 4.4 show the distribution of the uplift reasons, as well as the distribution of fault-inducing uplifts and clean uplifts for each reason. We observed that, in the Beta channel, most patches are uplifted because of a wrong functionality, crash, security vulnerability, incompatibility with some major websites, or to introduce/remove a feature. Most regressions are introduced by the uplifts that resolved wrong functionalities, crash, and security issues. For some uplift reasons, including improvement, resolving add-on/plug-in incompatibility and compiling errors, few patches lead to faults in our studied sample. However, a high percentage of patches resolving performance and rendering problems introduced new regressions.

In the Release channel, we observed the same top five uplift reasons. Compared to the Beta

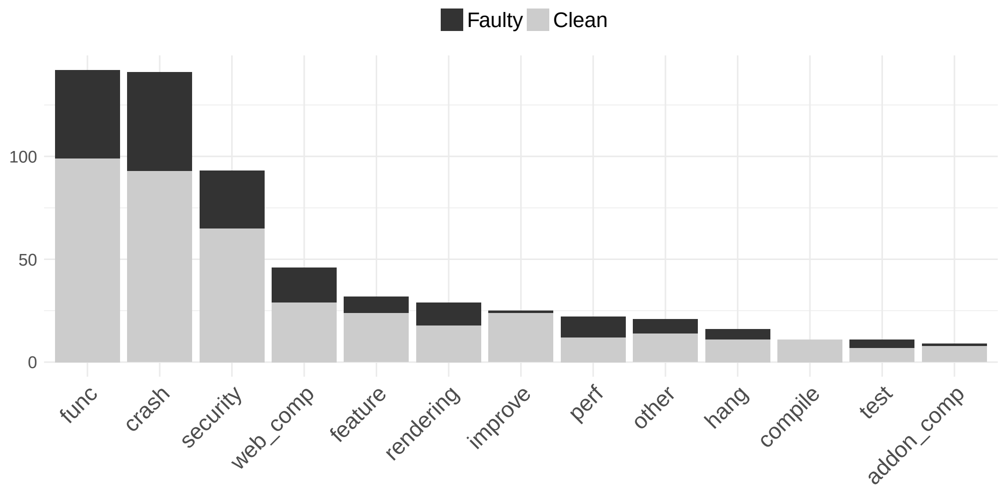


Figure 4.3 Distribution of uplift reasons in Beta.

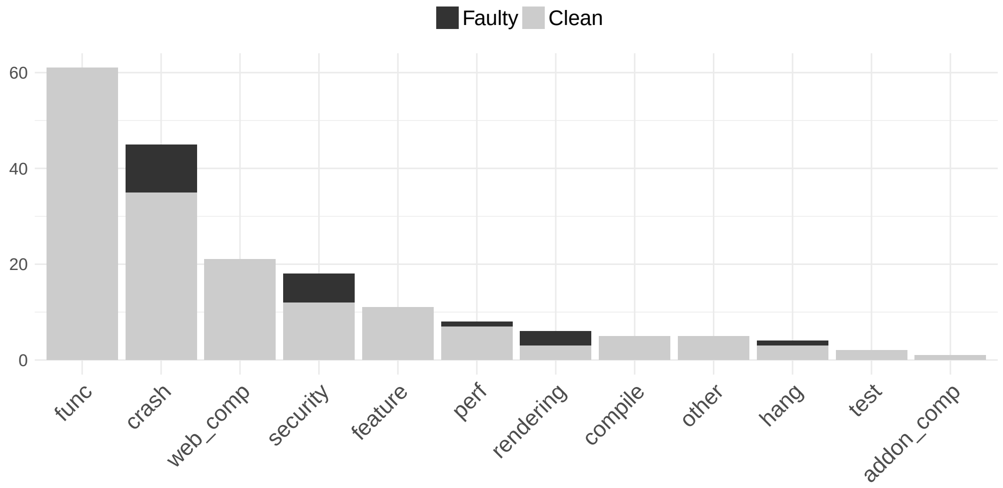


Figure 4.4 Distribution of uplift reasons in Release.

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

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

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

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

We presented the release managers with the results of our quantitative and qualitative analysis and collected the following observations.
They found that the response delta information is interesting. After thinking about it, they all gave us similar replies. When they are evaluating a complex issue and are still undecided, they will not make the call immediately. One release manager said that *"when I reject something, I won't make the call immediately. I will think about it before doing it, in case I change my mind or new facts are coming in the equation"*.
Regarding the landing delta, they were surprised, as they thought they were more likely to accept patches with a higher landing delta (that is, patches that have been in Nightly for longer). They have also said that they are almost always accepting patches during the first four weeks of the Aurora cycle, which would explain this discrepancy (as those patches have a small landing delta).

The interviewed release managers also told us that they take into account the fault-proneness of components when making uplift decisions; which is in line with what we found (some components have a smaller acceptance rate). One release manager told us that *"some components* *always come out as causing the most regressions, e.g., graphics layers, DOM".* Regarding the trust in developers, they all mentioned the assessment of risk as one of the first factors. One release manager explained that *“when they seem really overconfident or aren't telling me the whole story I lose some trust"*, another one stated that *“some developers are taking a lot of risks, some other less and are super reactive to fix potential fallout".* This finding is consistent with the uplift criteria followed at Facebook [91], where release managers tend to trust developers who introduced less regressions in the past.

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

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

RQ2: How effective are uplift operations?

Motivation. Previous studies showed that some issues cannot be effectively fixed by one patch, but need additional fixing efforts. These issues can be detected by seeking **reopened** [92], **cloned** [93], **duplicate**, or **resolved by multiple patches** [94] (which also includes backouts made by tree sheriffs, [95]) issues. In this research question, we want to examine whether it happens that patch uplift operations require multiple attempts (we refer to such uplifts as “ineffective uplifts”). Since such outcome is not desirable, it would be useful to help developers identify the characteristics of such patch uplifts, so that they can take the necessary steps to avoid reoccurrences of issues addressed by uplift operations.

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

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

To identify issues that were fixed by more than one uplift, we used regular expressions to detect uplifts in issue reports (see Section 4.2.1), and initially marked issues where at least two uplifts occurred (at a distance of at least three days between them). We chose three days because the distance between two beta builds is three days. A shorter time would likely have caught simple follow-up fixes that we are not interested in. A longer time would likely have missed some cases of multiple uplifts.

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

To identify issues duplicate of a previous issue fixed by patch uplift, we used the approach described in Section 4.2.2.

For each identified and verified issue that was not effectively fixed by an uplift, two of the authors independently card sorted the root causes of the ineffective uplift into one or multiple categories. They first defined categories separately, and then merged similar categories into one. Next, they standardized the category names as shown in Table 4.8. Finally, they used these standardized categories to compare their classification differences and resolve conflicts until reaching an agreement for each of the issues.

Results. Table 4.9 shows the number of ineffective uplifts detected from the three development channels. Since some patches were uplifted into multiple channels, the table also shows the unique number of the ineffectively uplifted patches in a specific manner (*e.g.*, reopened, cloned, or duplicate). Figure 4.5 depicts the root causes of the ineffective uplifts and shows the prevalence of each root cause. In this figure, if the patch of an issue was uplifted to multiple channels, we only counted it once. **In general, 196 out of the 4,958 (4%) studied issues were not effectively fixed by one patch uplift and required additional efforts.** In previous studies, Park et al. [94] and An et al. [96] respectively detected 32.8% and 23.8% general Mozilla issues (in different time periods) that were resolved by multiple patches. Shihab et al. [97] detected 6.5% to 26% reopened issues from Eclipse, Apache HTTP, and OpenOffice. Compared to these results, uplifted patches are more likely to fix a problem in one shot than other patches, even though we analyzed ineffective uplifted patches from different angles, including reopened, cloned, duplicate issues, and issues fixed by multiple uplifts. This implies that uplifted patches have a better general quality than other patches.

"The original uplifted patches did not completely fix the problem" is the most

Table 4.8 Root causes of the ineffective uplifts.

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

Table 4.9 Number of ineffective uplifts in the three channels.

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

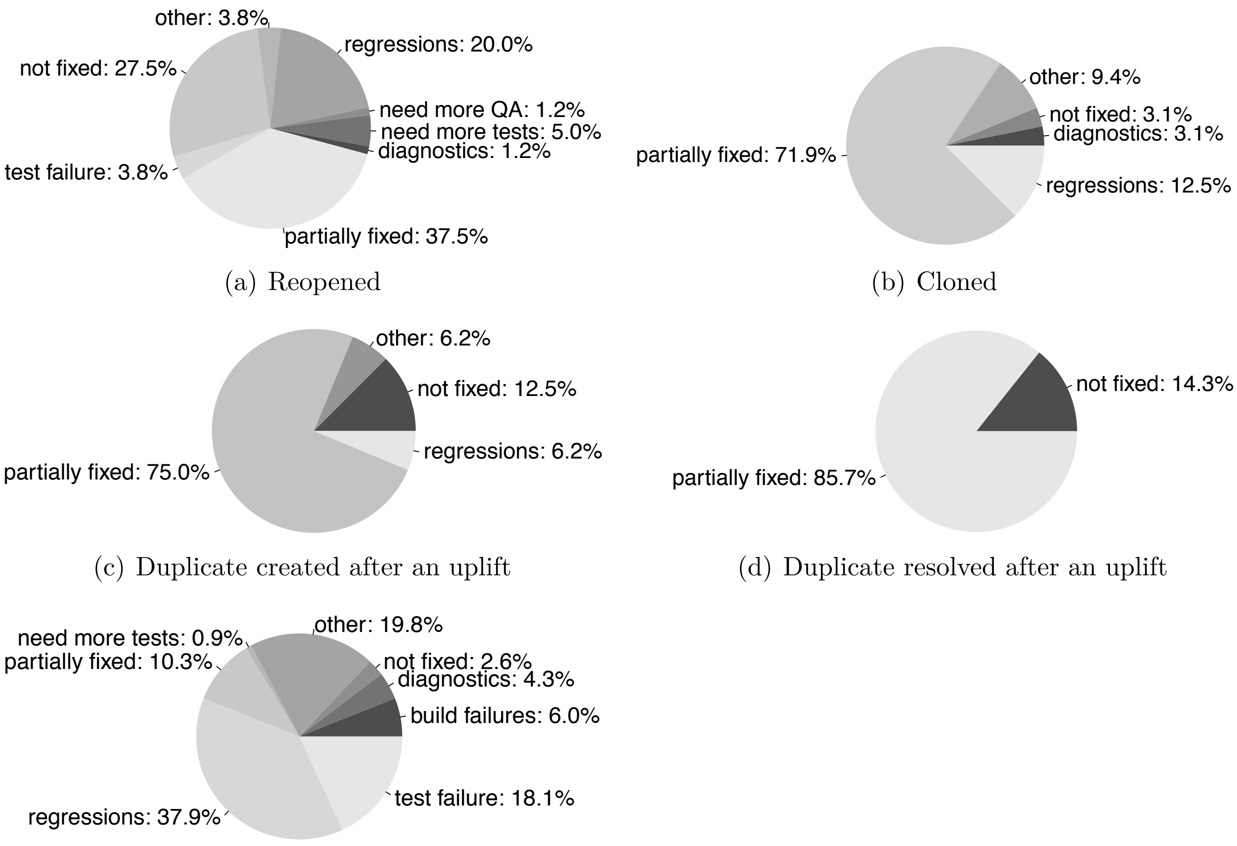
frequent root cause behind the issues that were ineffectively fixed and were later reopened, cloned, or duplicate. An example of such case is issue #1156182; the original uplifted patch of issue #1156182[4] only fixed the crash problem on Windows. The issue was reopened to further fix crashes on Linux.

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

In addition, among the ineffective uplifts, 27.5% of the issues were reopened after patch uplifts because these patches did not resolve the issues at all. 18.1% of the issues were resolved by multiple uplifts because their first uplifted patch did not pass a test case. Test and build

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

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



(a) Reopened

(b) Cloned

(c) Duplicate created after an uplift

(d) Duplicate resolved after an uplift

(e) Resolved by multiple uplifts

Figure 4.5 Root causes of the ineffective uplifts.

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

Moreover, we observed that 9 out of the 77 reopened issues did not completely get resolved, which were further filed as cloned or duplicate issues. For example, issue #1154003[6] was created due to crashes in the drawing method `DrawingContext::FillRectangle`. After uplifting a patch to the Aurora and Release channels, developers still observed a high volume of crashes with the same signature. To address the missing edge cases of these crashes, developers cloned the issue into issue #1162520[7]. This finding inspired us to investigate whether the cloned and duplicate issues were resolved in the same version as their original issues or resolved in a later version. We found that 23 out of the 54 (32+15+7) cloned or duplicate issues were resolved in the same version as their original issues, and the other 32 issues were resolved in a later version.

In this study, we only target for closed issues, but during our manual analysis, we observed that some issues fixed by uplifted patches have not been eventually closed. For example, issue #1297390[8] was created as a follow-up to the crashes fixed in issue #1280110[9]. Issue #1297390 has not been closed because the crash volume decreased again to a relatively low level. The priority of this issue were adjusted to P3, *i.e.*, would like to fix, but waiting for resources [98]. Although it would be interesting to investigate how many issues fixed by ineffective uplifts have been "completely and eventually" resolved, we can hardly get an exact answer because first, our subject dataset is dated from September 2014 to August 2016. Answering this question is beyond the scope of our study. Second, developers and testers can hardly know whether the most recent patch has covered all possible aspects to fix a certain issue, in other words, a "fixed" problem may come back again in the future. A lesson from this finding is that some issues are more difficult to get fixed than others. If an issue has recurred in the field, a proper follow-up is required even after the issue has been closed.

Regarding the differences of the ineffective uplift among channels, we observed that 153 out of the 4,368 (3.5%) Aurora uplifts, 112 out of the 2,614 (4.3%) Beta uplifts, and 16 out of the 285 (5.6%) Release uplifts were ineffective. Although the strictness of the uplift rules increases from Aurora, Beta, and to Release, the prevalence of ineffective uplifts does not decrease accordingly in these channels. The percentages vary among different kinds of ineffective uplifts, in particular "not fixed" uplifts account for 0.5% in Aurora, 0.9% in Beta, and 2.5% in Release. A possible reason could be that patches uplifted to the Release channel are aimed at more critical problems, which might be harder to fix. We looked in more detail at the "not fixed" cases in Release. It turns out that these uplifts indeed often fix very hard issues that occur in not-easily reproducible scenarios (even though they affect many users),

[6] `https://bugzilla.mozilla.org/show_bug.cgi?id=1154003`
[7] `https://bugzilla.mozilla.org/show_bug.cgi?id=1162520`
[8] `https://bugzilla.mozilla.org/show_bug.cgi?id=1297390`
[9] `https://bugzilla.mozilla.org/show_bug.cgi?id=1280110`

thus developers are forced to fumble around in the dark, attempting tentative fixes that sometimes do not work at all. However, we still suggest that release managers enhance the review effort on the Release uplifts, because these patches are targeted to the most stabilized version and most users of the product. Releasing updates to them without fixing the issues might be counterproductive.

According to our results, we suggest that developers and testers should carefully inspect whether a patch has completely resolved an issue and verify whether the patch has covered all possible scenarios of the issue. They also need to examine whether the patch would lead to new problems (*i.e.*, regressions) before requesting for uplift. Some ineffective uplifts (such as those due to test and build failures) can be prevented by performing uplift simulations.

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

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

Motivation. In **RQ2**, we studied ineffective uplifts, *i.e.*, uplifted patches that need additional fixing efforts. We observed that leading to regressions is one of the reasons of these ineffective uplifts. In this research question, we focus on the uplifted patches that introduced new regressions. These patches not only decrease the users-perceived software quality, but also increase development costs, since developers, testers and release managers have to rework the faulty patches. In Firefox' Aurora, Beta and Release channels, we found respectively 8.8%, 8.3%, and 7.9% of uplifted patches that introduced regressions in the system. Understanding the characteristics of these "fault-inducing uplifts" can help software organizations focus their QA and code review efforts on specific kinds of uplifts to prevent users' frustration.

1) Quantitative Analysis

Approach. To discover all possible fault-inducing uplifts, we applied the SZZ algorithm (described in Section 4.2.2) on all fault-fixing changes to identify uplifted patches that introduced a fault in the system. Next, we classified the uplifted patches into two groups: fault-inducing uplifts and clean uplifts. We used the 22 metrics listed in Tables 4.1 to 4.5 to assess the differences between these two groups. For each (m_i) metric, we tested the following hypothesis:

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

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

Results. Table 4.10 summarizes differences between the characteristics of uplifted patches that introduced a fault in the system and those that did not. We observed that fault-inducing uplifts have significantly larger patch size (m_{11}) than clean ones, across all three channels. The effect size of the difference is large. This implies that patches with larger modifications are more likely to introduce a regression if uplifted. We observed the following on the different channels:

- On Aurora and Beta channels, fault-inducing uplifts tend to have more complex code in terms of LOC, cyclomatic complexity, number of functions, and number of modules. These patches often contain classes that are connected to many other classes, in terms of closeness, betweenness and PageRank. Fault-inducing uplifts also tend to have higher comment ratios and tend to change files that were changed more frequently. Interestingly, fault-inducing uplifts are frequently submitted by developers or reviewers with high experience. Fault-inducing uplifts also have a larger amount of comments than clean uplifts. A large number of comments may be a sign that developers are struggling with the patch, which may explain the high fault-proneness. Although fault-inducing uplifts and clean uplifts also display other significant differences (as shown in Table 4.10), the magnitude of these differences is negligible.
- For the Release channel, we do not observe a significant difference between fault-inducing uplifts and clean uplifts for the above metrics.

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

Similar to **RQ1**, we examined patch uplifts per component, and observed that patch uplifts affecting certain components (*e.g.*, *Graphics* component) are more likely to cause regressions than others. Some of the components with the highest fault-inducing rates also have a low approval rate; probably because the release managers were acting based on their previous

Table 4.10 Fault-inducing Uplifts vs. Clean uplifts.

Channel	Metric	Faulty	Clean	*p*-value	Effect size
Aurora	Patch size	155.0	34.0	**<0.001**	large
	Prior changes	362.5	164.0	**<0.001**	small
	LOC	903.6	457.4	**<0.001**	small
	Cyclomatic	2.5	2.0	**<0.001**	small
	# of functions	34.3	17.0	**<0.001**	small
	Max. nesting	2.7	2.0	**<0.001**	negligible
	Comment ratio	0.2	0.1	**<0.001**	small
	Module number	2.0	1.0	**<0.001**	small
	Closeness	1.5	1.2	**<0.001**	small
	Betweenness	45,221.9	880.7	**<0.001**	small
	PageRank	1.7	1.4	**<0.001**	small
	# of comments	26.0	20.0	**<0.001**	small
	Developer exp.	28.5	10.0	**<0.001**	small
	Reviewer exp.	9.0	2.0	**<0.001**	small
	Comment words	10.0	2.0	**<0.001**	small
	Developer senti.	-3	-3	**<0.001**	negligible
	Owner sentiment	-2	-1	**<0.001**	negligible
Beta	Patch size	141.0	32.0	**<0.001**	large
	Prior changes	268.0	156.5	**1.02e-03**	small
	LOC	895.5	476.3	**1.66e-03**	small
	Cyclomatic	2.5	2.0	**3.69e-03**	small
	# of functions	37.0	18.0	**3.13e-03**	small
	Max. nesting	2.7	2.2	**0.01**	negligible
	Comment ratio	0.2	0.1	**<0.001**	small
	Module number	2.0	1.0	**<0.001**	small
	Closeness	1.6	1.2	**<0.001**	small
	Betweenness	35,661.7	1,327.8	**<0.001**	small
	PageRank	1.7	1.4	**<0.001**	small
	# of comments	28.0	22.0	**<0.001**	small
	Comment words	8.0	3.0	**0.04**	negligible
	Developer exp.	29.0	10.0	**<0.001**	small
	Reviewer exp.	10.0	2.0	**<0.001**	small
	Owner sentiment	-2	-1	**4.14e-03**	small
Release	Patch size	108.0	27.0	**2.07e-03**	large

experiences with those components (for example, the *Web Audio* component). Components like the *Audio/Video*, which are involved in multiple patch uplift operations, also have the highest fault-inducing rates; these components would be inherently more prone to faults because of their complexity, or technical debt.

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

2) Qualitative Analysis

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

Approach. We manually examined uplifted patches (from the samples selected in **RQ1**) that introduced faults, and classified the reasons behind the faults. Inspired by the work of Tan et al [71], we defined seven possible root causes for uplift faults (as shown in Table 4.11). We identified respectively 132 and 17 fault-inducing uplifts from the Beta and Release samples chosen in **RQ1**, and performed a card sorting to classify each of the faults into one or multiple causes. As in **RQ1**, the first and the second authors individually read the issue reports and their fault-fixing patches to understand the root causes of the faults (*i.e.*, the reason why their corresponding uplifted patches caused the faults) and classified these root causes along our seven categories. Similar to **RQ1**, disagreements were resolved through discussions.

We also interviewed release managers, asking them the following question: *What are the characteristics of fault-inducing patches that you are not currently taking enough into account but could be considered in the future?*

Results. Figure 4.6 depicts the distribution of the reasons why fault-inducing uplift introduced regressions. In both channels, semantic and memory-related errors are dominant root causes of the uplift regressions. With a detailed check on the patches, we found that many memory errors are due to null pointer dereference and memory leak. In addition, incompatibility of plug-ins and drivers also cause uplift regressions in both channels. Concurrency issues are ranked as a popular cause for Beta's uplift regressions, but we did not find any example of this category in the Release channel. In general, our results suggest that, when uplifting a patch, **release managers need to carefully check for potential faults on the program's semantic meaning, memory operations, synchronization, and third-party extension's compatibility**.

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

The fault categorization was also interesting for the release managers, who told us that Mozilla is about to employ more static analysis tools (*e.g.*, Coverity [72]) and to move some of their code from C++ to a safer language (*e.g.*, Rust). It is promising for them to see how many memory and concurrency faults can be avoided by using these techniques, and how many semantic and third-party faults can be reduced by enhancing code review or testing efforts.

Table 4.11 Fault reasons and descriptions.

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

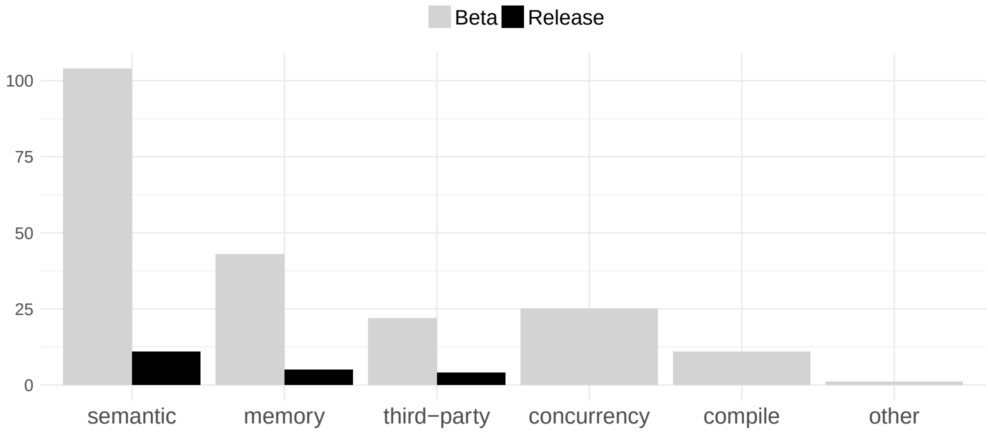


Figure 4.6 Reasons of fault-inducing uplifts.

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

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

Approach. We performed a manual analysis on the uplifted patches that were examined in **RQ3**. For each of these patches, two of the authors independently identified: 1) the problem the patch aims to address (noted as “original problem”), and 2) the impact of the regression the patch caused (noted as “regression problem”). To facilitate the comparison on the severity level between the original problem and the regression problem, we merged some of the categories (which have the same severity) defined in Table 4.7 as in Table 4.12. We also ranked the severity among different uplifted reasons (or regressions).

In some cases, the uplift and regression problems belong to the same category, but they affect users to a different extent. For example, issue #1059797[10] (which was uplifted to address a hang problem) caused a regression as issue #1239789[11] (which is a crash problem). Although crash and hang are considered to have the same level of severity, the first issue only happened during test runs, whereas the second one can be reproduced by users. To reduce any biases in the above rule, we also carefully examined the severity of the issues that belong to different categories. For example, issue #1075199[12] (which was uplifted to add a mock GMP plugin for testing) caused issue #1160914[13] (which is a crash). Although the latter is a crash, it only affects the plugin used for testing, *i.e.*, it has no impact on end users. Thus, we considered that the former is more important.

Results. Figure 4.7 depicts the proportion of uplifted patches that caused a more, same, or less severe regression. Tables 4.13 and 4.14 show the frequency and probability of a regression that an uplift on the Beta or Release channel can lead to.

In the Beta channel, more than one third (37.5%) of the manually examined uplifted patches led to a regression that is more severe than the problem they intended to address. Most of these patches were used to introduce improvements or new features (but caused crashes/hangs and broken functionalities), to fix broken functionalities (but caused crashes/hangs), or to fix performance degradation (but caused crashes/hangs and broken functionalities). In addition, we observed that crash/hang and broken functionality are the most frequent and the most probable regressions, which ranked as the top regression for each type of the analyzed uplifts. Especially, 50% of the patches uplifted to fix a crash caused other crashes, and 50% of the patches uplifted to fix a broken functionality broke other functionalities. Regarding the patches uplifted for security vulnerabilities (which have the worst impact on users), 21% of them caused other severity vulnerabilities and 29% of them caused crashes/hangs.

In the Release channel, none of the examined uplifted patches led to a regression that is more severe than the problem the patches intended to address. This result is expected because patches uplifted for the Release channel should have been more strictly reviewed and approved. The examined patches are only used to fix security vulnerabilities,

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

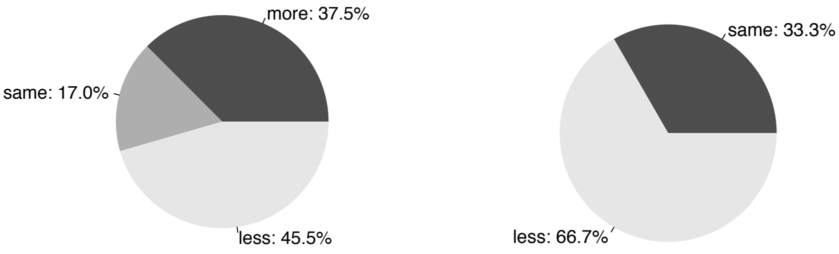
[11] `https://bugzilla.mozilla.org/show_bug.cgi?id=1239789`

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

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

Table 4.12 Categories of uplift reasons and regression impact. The severity is ranked by descending order (1 represents the most severe reason; while 6 represents the least severe reason)

Reason	**Description**	**Severity**
Security	Same as *security* in Table 4.7.	1
Crash	*crash + hang.*	2
Broken functionality (func)	*func + web compat + addon compat + rendering.*	3
Performance degradation (perf)	Same as *perf* in Table 4.7.	4
Improvement or new feature (improve)	*improve + feature.*	5
Compile or test problem (compile)	*compile + test.*	6
Other	Same as *other* in Table 4.7.	6



(a) Beta channel

(b) Release channel

Figure 4.7 Whether the regression an uplift caused is more severe than the problem the uplift aims to address.

crashes/hangs, and broken functionalities, which respected the uplift rules for the Release channel. 33.3% of these patches led to a regression as the same type of problem they intended to address. All these patches have a high probability to cause a new broken functionality.

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

Table 4.13 The frequency and probability of a regression that an uplift in the Beta channel can lead to (rows in *italic* indicates that the regression is more severe than the problem the uplift intended to address).

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

users.

Release managers were, as one might have predicted, happy to see our results regarding the release channel, but were not surprised because, compared to other channels, release uplifts are inspected with more QA efforts and are more carefully approved. When using the metrics listed in Tables 4.1 to 4.5 to compare the differences between Beta uplifts that caused more severe regressions than they fixed and other manually analyzed Beta uplifts[14], we observed that the former uplifts tended to happen closer to the release date and tended to have a shorter review duration (but these results are not statistically significant as the sample we analyzed is probably small). Release managers thought that these patches might have been uplifted in a rush and under pressure, which would explain both the closeness to the release

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

Table 4.14 The frequency and probability that an uplift in the Release channel can lead to.

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

date and the short review duration.

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

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

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

We then manually analyzed the identified regressions, categorizing both whether an issue was reproducible and how the issue was found. We have analyzed all Release regressions, and a representative sample of 152 Beta regressions (which corresponds to a confidence level of 95% and a confidence interval of 5%).

Table 4.15 and Table 4.16 show and describe how an uplift regression is reproducible and how it was found. We considered the regressions as *possibly preventable* by additional testing if they were not only reproducible by the issue reporter and were found either on a widely used feature/website/config or via telemetry. If they were reproducible only by the issue reporter, additional testing would not help. The regressions found via telemetry could be prevented if the data (crash reports and measurements) were analyzed in a timely manner (for example if there was an alerting system in place). We considered the regressions as *not easily preventable*, if they were reproducible but found on a rarely used feature/website/configuration, or found via telemetry but not reproducible, since manual testing is likely going to focus on widely used features/websites/configurations rather than seldom used ones, and issues noticed via telemetry are harder to fix if they cannot be reproduced. We consider the remaining regressions as *hardly preventable*: the regressions found by tooling could hardly be prevented, as the specific tooling was not available at the time the uplift was made (they could be prevented now that it is available); the regressions found by developers (*e.g.*, by code inspection) could hardly be prevented by additional testing. They could, in some cases, be mitigated by more detailed code reviews.

Results. Figure 4.8 shows the proportion of reproducibility on the regressions. On Beta, 58 out of 73 regression issues were reproducible by all or by some developers, 9 were not reproducible or reproducible only by the reporter. The reproducibility of the remaining 6 regressions cannot be identified. On Release, 10 out of 12 were reproducible by all or by some developers, 2 were not reproducible or reproducible only by the reporter. To summarize, **79.5% of the regressions caused by Beta uplifts and 83.3% of the regressions caused by Release uplifts were reproducible**.

Figure 4.9 shows the distribution of ways through which the regressions were found by Mozilla. In Beta, 20 regressions were found by developers, 14 were found by tooling, 13 were found via telemetry, 17 were found by users on widely used features/websites/configurations, 9 were found on rarely used features/websites/configurations. In Release, 4 were found by developers, 1 was found by tooling, 2 were found via telemetry, 3 were found by users on widely used features/websites/configurations, 2 were found on rarely used features/websites/configurations.

Between the two channels, both the reproducibility and how the issues were found have

Table 4.15 How an uplift regression is reproducible.

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

Table 4.16 How a regression was found.

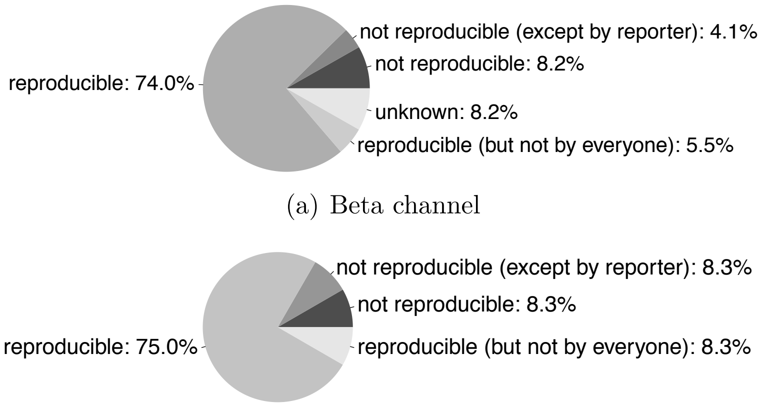
Found	**Description**
By tooling	The issue was found by fuzzing or static analysis.
By developers	The issue was found by Mozilla developers (by code inspection, by running tests that were not included in Firefox' test suites, or by running special tools such as Valgrind or ASan) or by an external developer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

similar characteristics (*i.e.*, the proportions are very similar), as can be seen from the figures mentioned above.

In order to understand the share of regressions that could have possibly been prevented, we compare the numbers of the possibly preventable, not easily preventable, and hardly preventable regressions in each channel. **In Beta, 20 regressions (around 30%) could have been possibly prevented according to our definition**; 13 regressions (around 20%) could not be prevented easily; 34 regressions (around 50%) could hardly be prevented. **In Release, 3 regressions (around 25%) could have been possibly prevented according to our definition**; 3 regressions (around 25%) could not be prevented easily; 6 regressions (around 50%) could hardly be prevented. We notice that the proportions are similar between the two channels; meaning that our discussion applies to both channels.

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

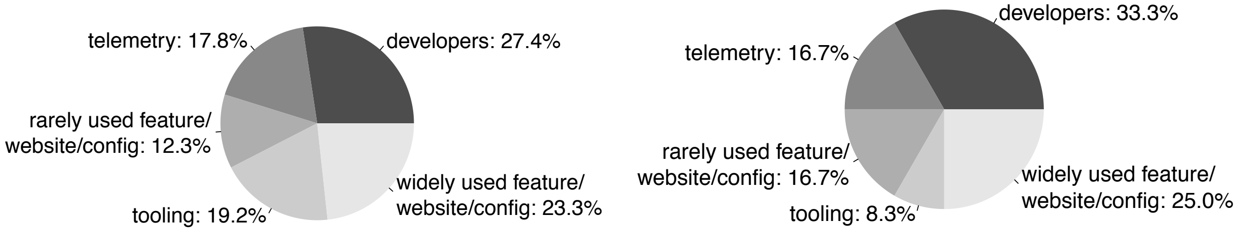
1. Try to detect issues via telemetry as early as possible (*e.g.*, using alerting systems), so that they can also be fixed in time;
2. Perform more QA on the stabilization channels, *e.g.*, trying more diverse configurations,



(a) Beta channel

(b) Release channel

Figure 4.8 Whether the regressions caused by an uplift were reproducible.



(a) Beta channel

(b) Release channel

Figure 4.9 How the regressions caused by uplifts were found.

as around 24% of the issues were reproducible and found on widely used features.

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

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

4.4 Threats to Validity

In this section, we discuss the threats to validity of our study following the guidelines for case study research [99].

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

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

Conclusion validity threats concern the relationship between the treatments and the outcome. We paid attention not to violate the assumptions of the statistical tests that are performed in this study. Specifically, in **RQ1** and **RQ3**, we applied non-parametric tests that do not require making assumptions on the distribution of our dataset. We used SentiStrength as the sentiment detection tool. We compared the performance of this tool with

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

SentiStrengthSE [100], the version tailored for software engineering, and obtained the same results, *i.e.*, no significant differences between accepted and rejected uplifts in any channel, and only a small effect size of the differences on the module owners' sentiment between clean and fault-inducing uplifts. Another reason why we prefer SentiStrength over SentiStrengthSE is that the former tool can be used from the command line and can be easily integrated into our automated scripts. On the contrary, the latter tool can currently only be executed from a user interface. In addition, when ingesting a large dataset such as the one we used in this study, the latter tool cannot be as easily deployed into a distributed environment. Before conducting the case study, we limited our studied dataset within a duration that covers consecutive series of relatively stable periods on all the three uplift channels. In addition, we used a keyword matching heuristic to identify fault-related issues. We manually validated a random sample of 380 issues. Three researchers participated in the validation. Whenever there were diverging opinions, we set up a meeting and discussed the issue until a consensus was reached. As a result, we found that our heuristic can achieve a precision of 87.3% and a recall of 78.2%, when identifying fault-related issues. Moreover, we performed manual classifications on the uplift reasons, the root causes of uplift regressions and reoccurrences, the reproducibility of the uplift regressions, and the way by which developers were discovered the regressions. We also manually compared the severity of the issues that the uplifts intended to address with the severity of the regressions that they led to. To mitigate potential bias that may result from our subjective opinions, we also discussed on each of our classification conflicts until reaching a consensus. However, as any other taxonomic study, we cannot guarantee a 100% of accuracy on our classification results. Future replications are welcomed to validate our work. Last, we used a heuristic to detect issues that duplicate a previous issue fixed by uplifted patches, which was inspired by Tian et al.'s approach [82]. Besides the automated detection, we manually confirmed every case used in our analyses to answer **RQ2**. Although some true positive cases might have been missed, the goal of **RQ2** is not to find all duplicate cases, but to understand why some uplifted patches did not completely resolve a problem and re-occurred in the field.

External validity threats are concerned with the generalizability of our results. In this work, we only studied Mozilla Firefox. First, Mozilla Firefox is the most studied system for issues related to rapid releases; moreover, the system's data are publicly available. We also have the opportunity to perform both quantitative and qualitative analyses (including the interviews with release managers) on this system. However, we should recognize that our findings may not be generalizable to other systems. In the future, we plan to collaborate with other software organizations, to validate and extend the results of this work. In addition, more studies on other systems with other programming languages are desirable to further validate our results. To facilitate future replication studies, we share our datasets and scripts at: `https://github.com/swatlab/uplift-analysis`. Another issue is that, in the manual classification, although we randomly chose our samples by applying a confidence level of 95% and a confidence interval of 5%, our samples might not precisely reflect the distributions of the uplift reasons and–or root causes of uplift regressions on the whole Firefox dataset. Further investigations on larger data sets are desirable.

4.5 Summary

Mozilla follows a rapid release model, which uses 18 weeks to deliver fault fixes and new features to users. Frequently, certain patches that fix critical issues, or implement high-value features are promoted directly from the development channel to a stabilization channel, because they are too urgent and cannot wait for the next release train. This practice, known as *patch uplift*, is risky because the time allowed for the stabilization of the uplifted patches is short. In average, 8% of uplifted patches introduced a regression in the code of Firefox. In this chapter, we investigated the decision making process of patch uplift at Mozilla and observed that release managers are more inclined to accept patch uplift requests that concern certain specific components, and–or that are submitted by certain specific developers (RQ1). We found that 4% of the issues fixed by patch uplift were not effectively resolved but were later reopened, cloned, duplicated, or fixed by additional uplifts. Two frequent root causes were identified from our manual analysis, *i.e.*, the original uplifts only partially fixed the issues or caused regressions (RQ2). We examined the characteristics of uplifted patches that introduced regressions in the code and found that they are more complex than clean uplifts, and they tend to change a higher number of lines of code. Most regressions are caused by patch uplifts aimed at fixing wrong functionalities and crashes. The most common root causes of faults in uplifted patches are semantic and memory errors (RQ3). In addition, through a manual analysis on a sample of the uplifts that introduced regressions, we found that more than one third of the fault-inducing Beta uplifts led to a regression that is more severe than the problem they aimed to address (RQ4). Last but not least, we observed that 25% to 30% of the regressions due to Beta and Release uplifts could be possibly prevented because they can be reproduced not only by the issue reporter but also by developers and were found on widely used feature/website/configuration or via the Mozilla telemetry (RQ5). We hope that software organizations take our findings and suggestions as a reference to improve their uplift (or urgent patch approval) strategy.

CHAPTER 5 AN EMPIRICAL STUDY OF DLL INJECTION BUGS IN THE FIREFOX ECOSYSTEM*

While Firefox and other equivalent browsers provide public APIs for extending functionality, a lot of third-party software still employs DLL injection techniques.

Since its inception, Firefox has always provided APIs to extend the functionality of the browser. There has been an evolution of methods to extend the functionality towards safer and more stable methods (starting from plugins such as Flash, moving to XUL/XPCOM extensions, then ending with JavaScript/HTML WebExtensions). While Firefox and other equivalent browsers provide public APIs for extending functionality, a lot of *third-party software* (*i.e.*, software that adds code into another software) still employs DLL injection techniques, *i.e.*, techniques that forces *host software* (*i.e.*, software that allows other software to extend its functionality) to run arbitrary code by making it load a dynamic-link library (DLL). By injecting arbitrary code, third-party software can extend the functionality of the host software without limits. However, injecting arbitrary code, while it is a very powerful technique, can easily cause severe bugs, such as crashes, in the host software. As can be seen in [101], bugs arising from injection can be indeed severe and widespread as to delay or cause revisions of entire software releases.

To the best of our knowledge, there has not been an empirical study towards understanding the DLL injection landscape, why third-party software vendors still employ these techniques despite the availability of safer alternatives, the root causes of DLL injection bugs, and proposing solutions to reduce them. This motivated us to conduct this work, in which we analyzed DLL injection bugs that occurred from July 2015 to August 2017 in the Firefox ecosystem. In particular, our study aims to answer the following three research questions:

RQ1: *What are the characteristics of the bugs caused by DLL injections?*

We observed that most of the DLL injection bugs led to severe problems. Out of the 103 studied bugs, 93 bugs (90.3%) caused crashes (among them, 47 bugs (45.6%) crashed Firefox while the browser was starting) and four bugs (3.9%) made the browser hang (*i.e.*, losing responses from users' requests). By analyzing the types of the third-party software,

*Part of the content of this chapter is published in "An Empirical Study of DLL Injection Bugs in the Firefox Ecosystem", Le An, Marco Castelluccio, and Foutse Khomh, *Empirical Software Engineering (EMSE)*, DOI=10.1007/s10664-018-9677-7.

we found that 57 bugs (55.3%) derive from antivirus software, 19 from hardware vendor drivers, and 10 from malware.

RQ2: *Which factors triggered the DLL injection bugs?*

To further understand the root causes of DLL injection bugs, we surveyed third-party vendors who caused the bugs. From their responses, we learnt that third-party software uses a variety of techniques (including standard Windows DLL injection techniques and proprietary techniques) to inject DLLs into the host software. DLL injection bugs can be triggered by injection engine errors, compiler/runtime incompatibility, or version incompatibility between the host and third-party software.

RQ3: *What would be the potential solutions to reduce such DLL injection bugs?*

In the survey, we also asked questions about the potential solutions that could reduce DLL injection bugs. From the answers, we realized that DLL injection should not be outright blocked from the ecosystem because it could be useful under certain circumstances, *e.g.*, when antivirus software intercepts suspicious processes. Host and third-party software vendors should strengthen their collaboration. Host software vendors should extend the features of the extension API (as a safer alternative to DLL injection) and can build a publicly accessible validation test framework.

Chapter Overview

Section 5.1 provides background knowledge on the Firefox ecosystem as well as the risks and countermeasures of DLL injection in the system. Section 5.2 describes the design of the case study. Section 5.3 shows and analyzes the results of the case study. Section 5.4 discusses the implications of our findings. Section 5.5 discloses the threats to the validity. Section 5.6 summarizes this chapter.

5.1 Background

5.1.1 Firefox Ecosystem

There are several ways third-party developers have been able to extend the functionality of Firefox: a) themes; b) plugins; c) extensions; d) DLL injection.

Themes are only allowed to change UI elements of the browser, thus they are very limited.

The API used to build plugins, NPAPI (Netscape Plugin Application Programming Interface), has been introduced by Netscape in 1995, and later adopted by most major browsers. NPAPI plugins declared content types that they could handle. When the browser was not natively able to handle that content type, it would load the appropriate plugin and let it run. NPAPI plugins are binary plugins, and they have been slowly deprecated for security reasons (*e.g.*, Chrome dropped NPAPI plugins in September 2015, Firefox dropped all NPAPI plugins except Flash in March 2017 and will drop Flash too in 2019).

Since its inception, Firefox has also allowed third-party developers to extend the functionality of the browser through JavaScript/HTML APIs by writing extensions. Extensions are either self-hosted, or hosted on a Mozilla website called AMO (addons.mozilla.org). When hosted on AMO, they undergo code review by Mozilla employees and/or volunteers. Since Firefox 44 (released in January 2016), Mozilla introduced a signing requirement where all extensions (either self-hosted or hosted on AMO) must be signed by Mozilla in order to be installable in Firefox (with the objective of reducing malware). This means that all extensions since Firefox 44 undergo code review.

Initially, extensions had access to browser internals (using XUL/XPCOM APIs); meaning that they could introduce technical debt into Firefox itself, as Mozilla developers could not easily modify Firefox internal code that was being used by extensions.

To ease development and to make extensions higher level (which would allow Mozilla to change their internal APIs without breaking existing extensions), Mozilla later introduced an extension SDK (JetPack). Behind the hood, JetPack extensions were still using XUL/XPCOM APIs.

A new set of APIs, the WebExtensions API [102], was later introduced in alpha state in November 2015, then in stable state since August 2016. Since November 2017, following a major rewrite of the browser which would have made many extensions incompatible, all extensions are required to use the WebExtensions API, which is an API supported by many major browsers (Firefox, Edge, and Chromium-based browsers). The advantage of such a common API is that developers only need to write a single extension and it will (modulo implementation differences) work on multiple browsers seamlessly, much like the web. The WebExtensions API is more restrictive than the old APIs, but also more secure and stable, and with better performance characteristics [103] [104]. Moreover, since these extensions are not allowed to use Firefox internal APIs, they cannot introduce technical debt as the old extension APIs used to do.

Another way that third-party developers use to extend the functionality of the browser (and of other software) is DLL injection.

5.1.2 Risks of DLL Injection and Countermeasures

By employing DLL injection, third-party developers are able to inject in the Firefox process any type of code, whose behaviour was not intended nor anticipated by Mozilla developers.

DLL injection is a powerful technique as it allows third-party developers to extend the functionality of the host software however they want, but it can be very risky. The injected code can, for example, use internal functions of the host software, without the knowledge of the host software developers, thus causing crashes or other problems when the host software removes or changes the behaviour of those functions. In order to use internal functions of the host software, some injected code depends on the binary layout of the host software, which changes for every specific build. If there are no mitigations in place, the injected code can cause crashes for every new release of the host software.

Figure 5.1 shows an excerpt of some buggy code injected in Firefox by a software using an open source library, EasyHook[1]. This is one of the few examples that can be shown, as usually the injection techniques are proprietary. In this example, Firefox is the host software (whose functionality is extended) and the software using the EasyHook library is the third-party software (which injects its code into Firefox). The process of the third-party software used the `CreateRemoteThread` function[2] to create a thread that runs in the Firefox process address space. The thread would call the `Injection_ASM_x86` function, which first loads the library to inject (line 11), then tries to find the entry point of the library using the `GetProcAddress` function (`AcLayers!NS_Armadillo::APIHook_GetProcAddress()`, from the Windows DLL: `AcLayers.dll`) (line 19). This is where the crash occurs: the address to the `GetProcAddress` function was retrieved by the third-party software in its process, but then called in the Firefox process, expecting it to have the same function and at the same address. Since Firefox does not load `AcLayers.dll`, this function does not exist in its process. EasyHook later fixed the bug by retrieving the address of the function from the remote process, rather than the process doing the injection.

Other software employed a very similar technique to the one used by EasyHook, but using `apphelp!StubGetProcAddress()` instead (from the Windows DLL `apphelp.dll`. Again, the technique is not used by Firefox). `AcLayers.dll` and `apphelp.dll` are both part of Windows, providing fixes for backward compatibility. `GetProcAddress` is usually part of `kernel32.dll` (which is loaded in every process), but for such software, Windows was probably shimming the API for compatibility, redirecting to `apphelp.dll` or `AcLayers.dll`.

[1] https://github.com/EasyHook/EasyHook

[2] https://docs.microsoft.com/en-us/windows/desktop/api/processthreadsapi/nf-processthreadsapi-createremotethread

```
public Injection_ASM_x86@0
  Injection_ASM_x86@0 PROC
  ; no registers to save, because this is the thread main function
  ; save first param (address of hook injection information)

        mov esi, dword ptr [esp + 4]

  ; call LoadLibraryW(Inject->EasyHookPath);
        push dword ptr [esi + 8]

        call dword ptr [esi + 40] ; LoadLibraryW@4
        mov ebp, eax
        test eax, eax
        je HookInject_FAILURE_A

  ; call GetProcAddress(eax, Inject->EasyHookEntry);
        push dword ptr [esi + 24]
        push ebp
        call dword ptr [esi + 56] ; GetProcAddress@8
        test eax, eax
        je HookInject_FAILURE_B
```

Figure 5.1 An example of DLL injection performed by RoboSizer

Mozilla later totally blocked this kind of injection mechanism which uses `CreateRemoteThread` (ironically, the code blocking this kind of injection mechanism triggered a bug in another third-party software, an antivirus, which was later fixed by the vendor).

Using public APIs rather than DLL injection is preferable. Besides the aforementioned examples, there are other reasons:

1. Since the WebExtensions API is supported by multiple browsers, the extension code only needs to be written once but can be deployed to different major browsers;
2. The public API is controlled by the browser vendor, who has information on the API's usage and can decide when to deprecate it (and when not to);
3. The extensions are written in JavaScript and HTML, just like normal web pages, which implies a very reduced chance of crashing the browser compared to the binary code that is injected with DLL injection;
4. Should an extension cause a problem, the browser can easily recover (*e.g.*, by reloading the extension). Instead, when an injected DLL causes a problem, it will likely lead to an unrecoverable situation.

Mozilla has been applying a blocklisting policy to react to bugs caused by third-party DLLs [105]. If a DLL causes a severe and–or widespread bug (such as an easily reproducible startup crash), Mozilla will, in parallel: a) try to contact the vendor of the third-party DLL and ask them to solve the problem; b) start preparing a blocklisting addition to block the DLL; c) attempt to reproduce the problem with its own quality assurance (QA) resources, if the third-party software is publicly available.

In order to solve the problem, third-party vendors usually request crash dumps from Mozilla, which often cannot be shared with external people for privacy reasons (the dumps might contain personal information of Firefox users). Mozilla may share crash dumps with third-party vendors only in the two following situations: 1) when Mozilla's QA manages to reproduce the crash; 2) when Mozilla manages to get in contact with users who can reproduce the crash (users can optionally leave their contact details when they submit a crash via Socorro, *i.e.*, Mozilla's automated crash reporting system) and the users agree to the sharing of crash dumps.

If the third-party software is publicly available, Mozilla will prepare modified Firefox builds that block the offending DLLs. Sometimes blocking a DLL is not easily feasible, as some DLL injection techniques operate at the kernel level. Sometimes blocking DLLs can cause more severe problems than the ones caused by the DLL itself. Hence, the blocklisting addition has to be tested first. If blocklisting works and does not cause regressions, Mozilla will apply the blocklisting patch, uplift it (*i.e.*, publish the patch ahead of the normal release cycle [106]), and, if the problem is widespread enough, generate a new release build to ship to users.

5.2 Case Study Design

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

5.2.1 Data Collection

From the Mozilla bug tracking system, Bugzilla [107], we searched bug reports that were created between July 2015 and August 2017. We chose this time window because the WebExtensions API was introduced in September 2015, and our study started in August 2017. In this work, we did not limit the analysis on already resolved bugs, because some bugs were closed as `WONTFIX` or `WORKSFORME`, for example, if a DLL injection bug was deemed too hard to fix for very little benefit or if the influence of a DLL injection bug drastically decreased after the opening of the bug. From all the bugs in the studied time period, we selected the ones that matched at least one of the following rules:

- the Bugzilla component of the bug is the one Mozilla uses to track bugs caused by third-party software ("External Software Affecting Firefox::Other");
- the title of the bug contains one of the keywords: ".dll", "virus", "malware" or "adware";

- the whiteboard of the bug contains the text “AV”, which Mozilla uses to mark some bugs caused by antiviruses.

We then manually analyzed the results of the search to filter out false positives, obtaining 103 bugs caused by external software through DLL injection.

The AV- and malware-specific rules only helped increasing our dataset slightly (5 out of 103 bugs), so our results should not be biased towards those kinds of software. Within the results from the other generic rules, we also found AV- and malware-specific bugs.

5.2.2 Data Processing

We manually identified a series of characteristics from the 103 bugs obtained in Section 5.2.1. Table 5.1 shows the names and the descriptions of the characteristics. To reduce biases in the manual identification, two of the authors separately collected the characteristics before comparing their results together. They created an online document to discuss any divergence until reaching an unanimous decision. In addition, we wrote scripts to automatically extract some other characteristics as shown in the bottom of Table 5.1.

5.2.3 Survey

To further understand the root cause of the DLL injection bugs and how the bugs were resolved, we designed a survey intended for the 58 vendors who caused these bugs. However, we could not find the contact information of 14 vendors (including the malware producers) from Bugzilla or through an online search. Hence, we ended up contacting only 44 vendors. Among them, 12 vendors answered all or part of our questions, which corresponds to a response rate of 27%. As we aim to propose potential solutions to reduce this kind of bugs, we also asked these software vendors questions on improving the reliability when adding their code into Firefox.

In our survey, we only used open questions. Participants could choose all or a part of the questions to answer. Our questions were designed to better understand the DLL injection landscape: what techniques are used, what kinds of bugs can arise, why DLL injection is still used as an extension mechanism despite the presence of safer techniques. Here are the questions we used in the survey:

Q1. What is the injection mechanism that you used?

Q2. Do you know the root cause of this bug?

Q3. If the bug is resolved from your part, do you remember the way by which you resolved

Table 5.1 Characteristics of the bugs caused by third-party software.

Characteristic	**Description**
Manually collected characteristics	
Bug impact	Whether a bug broke the functionality of the browser, caused a crash (or startup crash), or caused a hang.
Software name	Name of the software that caused a bug. If no software name is mentioned in a bug report, we marked as “unknown”.
Software type	Type of the external software, *e.g.*, antivirus, malware, and hardware vendor driver.
How resolved	How a bug is resolved, *e.g.*, fixed by the vendor, or blocked by Mozilla.
Reproducibility	Whether a bug can be reproduced by the QA of Mozilla or third-party vendors.
Automatically collected characteristics	
Percentage of DLL users	Percentage of Firefox users who also have the third-party software.
Fixing time	How many days it took for a bug to be fixed since its first occurrence. We cannot retrieve the first occurrence date for some bugs, we have to use the time period from the creation date until the fixed date to estimated these bugs’ fixing time.
Tracked or blocking	Whether a bug was ever tracked for a release or was blocking a release. More information about Mozilla tracking flags and how they are used in the release management process can be found in [108].

this bug?

Q4. Since Mozilla is encouraging other organizations to produce their software as an extension, is there any specific reason why you are still using the way of DLL injection to add functionalities into Firefox?

Q5. Would you be open to switching to an extension-based solution if Mozilla gave you the API you needed?

Q6. Do you run QA with pre-release versions of Firefox (*e.g.*, Firefox Beta)?

Q7. Do you have any suggestions to improve the Mozilla API extension?

A possible approach to mitigate the DLL injection issues is to adopt a whitelist solution. Instead of reacting to DLL injection issues by blocklisting misbehaving DLLs, Mozilla could proactively block all DLLs except “good" ones. The vendors in the whitelist would need to be more careful and perform QA in order to be in the whitelist. Once a whitelisted DLL causes a problem, it will be removed from the whitelist. Also, developers using the WebExtensions API would effectively be exempt and would always be in the whitelist. Besides reducing bugs, Mozilla expects that this mechanism can push third-party software vendors to use the WebExtensions API, which can also avoid crashes in the third-party code taking down Firefox [101].

To evaluate how this solution would be received by third-party vendors, we asked additional questions to the vendors who have answered our initial questions. During this work, we consulted some Mozilla developers by email and added these follow-up questions based on their suggestions.

Q8. In your opinion, what would be a solution to allow for an effective integration of third-party code into software like Firefox?

Q9. Some software vendors are moving to instruct users to uninstall third-party software after a crash, what do you think of such practice?

Q10. When Firefox rolls out new content security features, it often runs into compatibility issues with third-party suites that leverage injection. What steps do you think Firefox should take to prevent these issues with your product(s) in the future?

Q11. What support might you be willing to provide to avoid these issues in the future?

Q12. If Firefox blocks third-party injection associated with your product, what side effects do you anticipate? Would this potentially break your software product(s)? Could this break Firefox?

Q13. Some vendors are considering introducing a whitelist that only allows "reliable" DLLs to be installed. Would the whitelist be an incentive to adopt the cross-browser WebExtensions API? (products using the extension API are always whitelisted)

Q14. Would the existence of a whitelist be an incentive for your company to do more QA with Firefox?

Q15. Would your company try to circumvent the whitelist? If yes, how would you do it?

5.3 Case Study Results

We present the results of our case study and discuss the implications of these results.

RQ1: What are the characteristics of the bugs caused by DLL injections?

According to Mozilla telemetry[3], large shares of Firefox users are also users of software employing DLL injection to extend Firefox functionality. Each major third-party software can be installed on between 1% and 15% of Firefox users' machines. Severe bugs affecting a DLL from a third-party software that is installed on 15% of users' machines (or even 1%) can be very concerning for Mozilla.

Table 5.2 shows the distribution of the impact of the DLL injection bugs. Out of the 103 studied bugs, 93 bugs (90.3%) caused browser crashes, *i.e.*, the browser unexpectedly ter-

[3] `https://wiki.mozilla.org/Telemetry`

Table 5.2 Impact of the DLL injection bugs (some bugs have more than one impact)

Bug impact	**Occurrence**	**Proportion**
startup crash	47	45.6%
crash (unknown)	25	24.3%
crash	21	20.4%
broken functionality	8	7.8%
hang	4	3.9%
plugin crash	2	1.9%

Table 5.3 Types of the DLL injection software

Software type	**Occurrence**	**Proportion**
antivirus	57	55.3%
hardware vendor driver	19	18.4%
malware	10	9.7%
multimedia tool	4	3.9%
screen reader	3	2.9%
other	3	2.9%
IME	2	1.9%
download manager	2	1.9%
desktop customization	1	1.0%
file hosting service	1	1.0%
accessibility	1	1.0%

minates. Among them, 47 bugs (45.6%) caused crash during the browser startup (the most severe type); 21 (20.4%) crashed while the browser was running; we could not deduct the type of crash from the other 25 bugs (24.3%) (*i.e.*, uptime unknown). Besides, two bugs (1.9%) crashed a browser plugin. In addition, four bugs (3.9%) caused hangs, *i.e.*, the browser does not respond to users' requests. Only eight bugs (7.8%) have lower severity. They break the browser's expected functionality. The overall impact of the DLL injection bugs are severe, which can negatively affect users' trustfulness on the quality of the browser. From the side of users, they may not know whether the severe problems (such as crashes) are caused by the host software itself (Firefox in this case) or by its interaction with third-party software (usually they will just assume it is the host software, since that is the one which crashes, even if the crash stems from injected code). If the problems are kept unresolved for a long time, users may switch to other equivalent products. Especially for startup crashes, where users cannot use the browser at all, nor automatically update it to a newer version when a fix is released by Mozilla. The only options for them are to manually reinstall Firefox after a fix is released, wait for an update of the third-party software, or switch to use another browser.

Table 5.3 shows the types of the DLL injection software. More than half of the bugs (57,

Table 5.4 How the DLL injection bugs were fixed (some bugs were fixed by more than one resolution)

Resolution	**Occurrence**	**Proportion**
fixed by the vendor	24	23.3%
worksforme	18	17.5%
not yet resolved	18	17.5%
blocklisted	16	15.5%
duplicate	12	11.7%
wontfix	8	7.8%
workaround	5	4.9%
invalid	2	1.9%
fixed by switching to WebExtension	2	1.9%
fixed bug in firefox	1	1.0%

i.e., 55.3%) are from antivirus software, 19 (18.4%) are from hardware vendor drivers, 10 (9.7%) are from malware, and 17 (16.5%) are from other software, including multimedia tools, screen readers, input method tools (IME), and download managers. Overall, except for a small amount of malware and purpose-unidentified software, most bugs are derived from DLLs that provide useful features to users.

Table 5.4 shows how the DLL injection bugs were resolved (or not resolved). 58 bugs (56.3%) were not actually resolved by the time of this study. Some of the bugs were closed with a label as "WORKSFORME" (bugs can no longer be reproduced), "INVALID" (bugs are in the third-party software and with low enough severity), "WONTFIX" (due to low or decreased volume of impact), or "DUPLICATE" (duplicate of another resolved bug). Unfortunately, the labels are not always used consistently (for example, bugs with very low impact are sometimes resolved as INVALID and sometimes as WONTFIX). Besides, five bugs (4.9%) were fixed by employing workarounds (temporary and ugly solutions). For the bugs that were actually resolved, 16 (15.5%) were fixed by Mozilla by blocklisting the offending DLLs; 24 (23.3%) of them were fixed from the vendor side. Only two bugs (1.9%) were resolved by switching to using Mozilla's WebExtension API as recommended. Merely one bug (1%) was not due to the DLL vendors but due to defects of Firefox. From the result, we observe that a weak percentage of the bugs can be resolved by the host software itself (Firefox). Third-party vendors' efforts and collaboration are important to keep the Firefox ecosystem healthy. Moreover, few third-party vendors have adopted Mozilla's recommendation of using the WebExtensions API.

Figure 5.2 depicts the time period (in six weeks periods) during which the DLL injection bugs were resolved. In this figure, we only considered the 81 bugs that were closed by the time

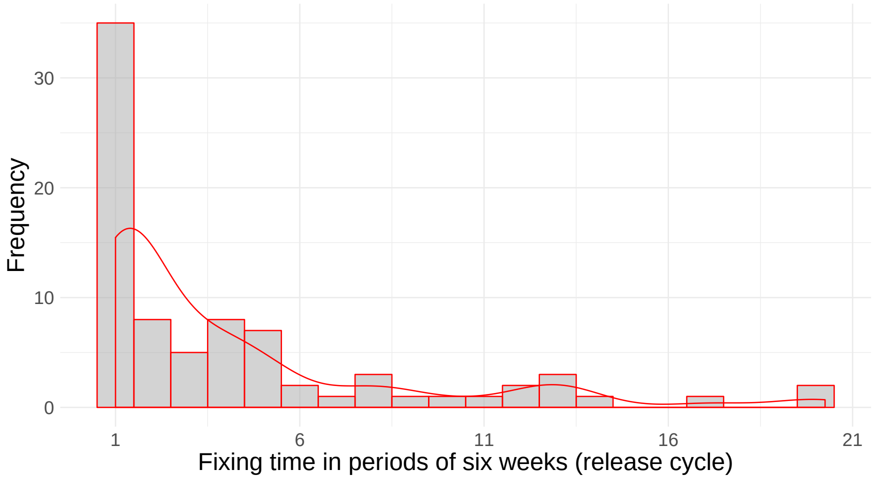


Figure 5.2 Distribution of the bug fixing time. Each bin represents a period of six weeks, *e.g.*, the first bin means bugs fixed within six weeks (*i.e.*, one release cycle).

of this study. 40 bugs were fixed within a period of six weeks; meaning that nearly half of the DLL injection bugs can be fixed before the next release. 55 bugs were resolved within 18 weeks, a full release cycle from Nightly to Release. End users can benefit from the resolution of these bugs within three releases (a new version is released every six weeks). However, we also observed 10 bugs that were not resolved for more than one year. Moreover, 22 other bugs have never been resolved until the writing of this thesis. Long resolution time of DLL injection bugs challenges users' trustfulness not only to the third-party software, but also, and in many cases even more, to the host software. To maintain the health of the ecosystem, both sides of the host and third-party software need to actively and effectively discover and resolve bugs. We found that some bugs, such as Bug #1268470, were resolved late because at the time of reporting the bug, it affected only a small number of users. When the bug started affecting more users, it attracted Mozilla's attention.

Although Bugzilla has priority/importance fields, they are used inconsistently by different developers and different teams, thus cannot be relied upon to infer the importance of a given bug. In order to evaluate the actual severity of the bugs, we analyzed the Bugzilla tracking flags that are used by Release Managers during the release process [108]. We found that 32 bugs (31.1%) were tracked or blocking for a release at least once. These kinds of bugs are particularly important because they either have been closely monitored by release managers for possible resolution in a Firefox release (tracked bugs: 24, 23.3%) or have been marked as blocking (**must be fixed before shipping**) a Firefox release (blocking bugs: 8, 7.8%). To put it into perspective, we can compare these percentages with the overall ones: 3390 tracked bugs (around 0.037%) and 165 blocking bugs (around 0.002%). This means that

DLL injection bugs, even though expectedly rarer than other bugs, are often more severe than other bugs. We also compared the fixing times of DLL-injection blocking/tracked bugs with those of generic blocking/tracked bugs. In addition, we found that the average fixing time is around 3.4 times higher for DLL-injection tracked bugs than generic tracked bugs (for blocking bugs the average is 2.8 times higher). However, the differences are not statistically significant based on the Mann-Whitney U test [66]. One reason is that there are too few samples in our dataset.

Finally, 26 (25.2%) of the DLL injection bugs could be reproduced by Mozilla or third-party vendor's QA, four (3.9%) of the bugs could not be reproduced, and we cannot identify whether the rest 73 bugs (70.9%) could be reproduced or not. For bugs that were reproducible, additional QA performed by either Mozilla or the third-party vendors before a Firefox release could have prevented the bug from hitting users. Among the aforementioned eight blocking bugs (account for 7.8%), five of them could be reproduced by Mozilla or third-party QA, one of them could not be reproduced, and we cannot identify the reproducibility for the remaining two bugs. If more in-depth QA was part of the envisioned whitelist policy of Mozilla, many of these blocking bugs could have been resolved before they became blocking.

RQ2: Which factors triggered the DLL injection bugs?

Firefox is an open source browser. Its crash and bug reports are also open to the public. Developers and researchers can leverage these resources to understand the root causes of most bugs. However, through our manual analysis, none of the DLL injection software that caused bugs in Firefox is open source. Thus, we cannot understand the root causes of these bugs from source code. As we observed in **RQ1**, many subject bugs, which were eventually resolved, were fixed by the software vendors or blocked by Mozilla. In both cases, Mozilla did not know the triggers. Although the third-party vendors knew the triggers of the bugs they resolved, they rarely mentioned them in the bug reports. In other words, bug reports cannot help us to understand the bugs' root causes either. Therefore, to answer this research question, we decided to ask the software vendors themselves. In the rest of this section, we will show the vendors' responses to the corresponding survey questions and discuss these responses. Table 5.5 shows statistics on the participants for each survey question. In this table, we respectively provided the total number of participants who answered a question, types of these participants' software, and number of participants for each type of software. All the reported responses are from closed source software vendors. Due to privacy reasons, we may have hidden some confidential details.

DLL injection mechanisms used by the software vendors (Q1).

Table 5.5 Statistics on the survey participants (all participants are from different vendors)

Question	Participants	Software type (and its frequency)
1	12	antivirus (7) screen reader (1) unknown (1) internet downloader (1) media recorder (1) hardware vendor driver (1)
2	12	antivirus (7) screen reader (1) unknown (1) internet downloader (1) media recorder (1) hardware vendor driver (1)
3	10	antivirus (5) screen reader (1) unknown (1) internet downloader (1) media recorder (1) hardware vendor driver (1)
4	11	antivirus (7) screen reader (1) unknown (1) hardware vendor driver (1)
5	7	antivirus (3) screen reader (1) unknown (1) hardware vendor driver (1)
6	6	antivirus (2) screen reader (1) unknown (1) hardware vendor driver (1)
7	5	screen reader (1) unknown (1) hardware vendor driver (1) media recorder (1)
8	5	antivirus (3) media recorder (1)
9	4	antivirus (3) media recorder (1)
10	4	antivirus (3) media recorder (1)
11	4	antivirus (3) media recorder (1)
12	4	antivirus (3) media recorder (1)
13	4	antivirus (3) media recorder (1)
14	5	antivirus (3) media recorder (1)
15	4	antivirus (3) media recorder (1)

We received 12 responses to the question related to the injection mechanisms used on Firefox. Two general kinds of mechanisms can be identified from the responses: standard Windows techniques and proprietary techniques. Among the eight responses on the standard techniques, seven participants explained the detail of their technique, one participant only mentioned that their DLL injection technique is standard for the Windows OS. Here we quote our participants' answers to this question: *"It's just a standard Shell Extension that runs when folks use the open/save dialogues." "We use SetWinEventHook [109] from user32.dll." "We used a general mechanism (SetWindowsHookEx [110]) to inject other processes in order to be able to influence window creation flags in case the user decides to not be disturbed in Game Mode / Do Not Disturb Mode." "AppInit_dll [111] registry entry." "CreateRemoteThread+LoadLibrary [112, 113]".*

Three participants said that they used proprietary techniques, but none of them revealed details. Two other participants did not directly answer this question but said that the injection mechanism is irrelevant to the bugs. Overall, **third-party software uses a variety of techniques to inject DLLs into the host software**.

Root causes of DLL injection bugs and resolution mechanisms (Q2, Q3).

Our second and third questions concerned the root causes of the bugs and how the bugs were resolved. Nine participants explained the root causes of the bugs caused by their injected software. 10 participants explained the resolution process of the bugs caused by their injected code. Some bugs were caused by the injection engine. The participants said: *"Bug in hook engine. Legacy code not covered by automatic tests."*, *"Problem was internal to the hooked functionality and likely not dependent on Firefox code"*. The DLL vendors resolved the bugs by fixing their injection code.

Compiler or runtime incompatibility is another cause mentioned: *"Our compiler wasn't C++ 11 compliant and therefore introduced a race initialization of a mutex." "(Our DLL) was incompatible with C++ runtime, shipped with Windows 8.0 x64. It is not depend of upgrade or clear installation of FF (Firefox). In addition, it should not depend from browser, for crash it is enough Windows 8.0 x64 C++ runtime and any browser."*. Participants did not provide detailed information about the resolution of this problem. We suppose that upgrading the compiler would address the bugs.

Some other bugs were due to generic programming mistakes, which were later resolved and made the DLL work again. One participant explained: *"It was a mistake regarding 64 and 32 bit values in our code base." "bad_alloc wasn't caught in our code."*

In addition, bugs can also occur when *"users forcibly loaded old extensions to newer versions of Firefox and disabled compatibility checks ... (Old versions of Firefox) missed a check for NULL on one of interface queries. The issue started to persist after significant changes in Mozilla interfaces."* To reduce this kind of bugs, the host software can alert users to upgrade their old version of the third-party software, and warn them of the potential consequences of the incompatibilities on the host/third-party software versions.

Based on our observations, **most bugs are due to injection engine problems, compiler/runtime incompatibility, or version incompatibility between the host and third-party software**. This finding corroborates what we found in RQ1: most bugs are in third-party software's code and thus cannot directly be fixed by Mozilla.

RQ3: What would be the potential solutions to reduce such DLL injection bugs?

Unreliability challenges all software ecosystems. To reduce potential crashes caused by third-party software, from September 2018, Chrome will try to block most third-party software that injects code into it [114] (Chrome developers claim that "users with software that injects code into Windows Chrome are 15% more likely to experience crashes"). The organization hopes third-party software can switch to use the recommended WebExtensions API to run code inside Chrome processes. Mozilla is also trying to reduce bugs caused by third-party software, while avoiding outright blocking, by introducing a whitelist to allow only DLLs, which are proved reliable, to inject code into Firefox. With the same expectation as Chrome, Mozilla hopes that this measure can make third-party software vendors switch from DLL injection to WebExtensions, which is considered as a more reliable way to interact with Firefox. In this study, by analyzing survey participants' answers, we want to discuss whether the whitelist is the best solution to reduce bugs from third-party software, and whether there are better alternatives to it.

Reasons provided for not adopting WebExtensions (Q4).

First, we wanted to know the reasons why many third-party vendors are still using the way of DLL injection, although WebExtensions have been available for a while (in alpha state since Firefox 42, released in 2015-11-03; in a stable state since Firefox 48, released in 2016-08-02). This corresponds to Question #4 in the survey. 11 participants answered this question. Multiple participants mentioned that their DLL is not specifically designed for Firefox but is also being used for other host software, *e.g.*, *"Our software is not just used for FF (Firefox). It is a general purpose audio recorder. Users choose which application they wish to target."* For these vendors, migrating to WebExtensions would not be interesting because it requires extra efforts to refactor the existing code.

Another reason is that some vendors cannot use WebExtensions to achieve their goal, *e.g.*, *"We must be able to gather content from Firefox. The most efficient way being to inject. Extensions are not suitable for Screen Reading software such as ours"*. An antivirus vendor said: *"We provide secure input feature in our product, which means that no one can intercept symbols, which user input in browser fields. The task could not be done on Windows OS without kernel driver and injected dll in browser"*. Another antivirus vendor explained: *"As hackers always inject, while we are reducing to minimize our injections, we cannot totally eliminate them"*. This would partially explain why a big percentage of DLL injection bugs derive from antivirus software. Due to the above two reasons, if a host software banned DLL injections, the vendors will have to find other feasible hosts.

Moreover, some participants indicated the disadvantages of WebExtensions, *e.g.*, *"The main disadvantage we find is that WebExtensions can be easily disabled (for a user with admin-rights, and in a Windows workgroup environment). We had taken this route of injecting a DLL to enforce URL filtering even in such environments"*. Again, DLL injection is currently the most suitable way for such vendors.

Only one participant is willing to accept WebExtensions, but they also said that WebExtensions cannot fulfill some particular purposes, which is inline with the aforementioned observations.

In general, **some DLL vendors do not want to adopt WebExtensions, because they do not target for one specific host software, and the features currently offered by the WebExtensions API are still limited for some purposes**. One participant told us that their organization has thoroughly analyzed the pros and cons about using WebExtensions. However, they still keep using DLL injection because they *"don't see any way how and why to stop injecting there (in order to protect our users, which is our business)"*. We cite their analysis here and hope that host software organizations can take this as a reference to improve the extension API and–or communicate better about their advantages.

"In comparison with injection, extension has much worse deployment possibilities – the installation process is cumbersome (you can't install the extension silently without user interaction which is a major UX problem, you can't protect the extension from uninstalling, you'd need to check for browser reinstalls and install again etc).
Also, it's possible to write the extension, but since the API is limited (everyone saw the 2/3 of extensions being removed from new Firefox because of API problems) and the model is also asynchronous, which kinda gets in a way what would AV product need. And the next point against extensions is a need for three different extensions for three browsers – although they all use WebExtensions, they're quite different. And MSIE is still there, with stronger presence than Edge."

Migration from code injection to WebExtensions (Q5).

Q5 is about whether third-party vendors are open to switch to WebExtensions if Mozilla gave them the needed API. Seven participants answered this question. One participant, who is the one saying that WebExtensions can be easily disabled, simply said Yes. Those vendors targeting multiple hosts answered No, because *"Mozilla doesn't control the surface area we modify"*.

A participant suggested that if different host software organizations can standardize their APIs, third-party vendors will be more willing to migrate. *"It depends on the functionality and if there are general, OS runtime based standard mechanisms already available. It makes no sense to have two different implementations of the same functionality."*

Other participants' attitude is rather open, but they doubt whether Mozilla can provide the specific API they require. For example, *"I doubt that the extension mechanism would be sufficient for our requirements. However, we, Mozilla, and other vendors are actively considering other ways that software such as ours would not have to inject to gather this content."*

"We are combatting malware and exploits though, which work in a low-level way, directly

manipulating Firefox code and interacting with the operating system. It is quite unlikely that a high-level extension (i.e., JavaScript) can be used to detect and mitigate all those threats reliably."

"Actually, we prefer to use 'standard' means whenever possible ... The main concern is, how do you expose the API without any malicious software using it."

Overall, **although some third-party vendors are open to adopt WebExtensions API, they doubt whether the API can fulfill their requirements**.

Quality assurance of injected code (Q6).

Six participants answered whether they run QA with pre-release versions of Firefox. Four participants said Yes, one of them further explained: *"but not as often as we would like"*. The other two said No. In our opinion, running QA against each version of the host software is necessary. The vendors who neglect this process may miss bugs in the ecosystem. In this case, the whitelist would be an effective measure to penalize the vendors who do not test their software well and frequently have bugs.

Suggested improvements to the WebExtensions API (Q7).

Q7 encourages participants to suggest improvements for the WebExtensions API. One participant wished that *"(Mozilla) can provide a mean to get the HWND [115] of a window from within the extension"*. This suggestion is in line with the doubts on the functionality offered by the WebExtensions API.

Another suggestion is about the reliability of the API itself: *"Some of the mechanisms (of WebExtensions) do not work ... We opened a bug (on this problem)"*. Therefore, completely blocking DLL injection may not be the best solution because if a third-party vendor can neither use DLL injection nor program against an available/reliable API, they have to give up the host software and find other platforms. However, if all browsers move to reduce DLL injection, third-party software will be forced to gradually transition to WebExtensions.

To further discuss the solutions of reducing DLL injection bugs, we will analyze the answers on the follow-up questions. Some of the questions are targeted for the upcoming whitelist by Mozilla. Only five participants answered these questions. Their answers may not be representative, but can be used as a reference for host and third-party software to improve the reliability of an ecosystem. In the following of this section, we will cite their answers and discuss the implications.

Allow an effective integration of third-party software into another software (Q8).

Our follow-up questions start by how to allow an effective integration of third-party software into another software. Our participants answered as follows: *"Certainly the most common extensions can and should be handled by a plugin API like WebExtensions. Additionally, having a link to AMSI (Anti-Malware Scan Interface) by Microsoft would make sense. But generally, what Windows supports should be also supported by Firefox, which also includes code injection. For monitoring the process state on a system level, sometimes there are no other options that would come to my mind."*

"Use of extensions is the most effective method. However, in enterprise environment, admin would want to enforce use of certain extensions (without allowing a user to disable it). Browsers allow enforcing certain extension through group policy in domain environment. However, we have a lot of SMB (small and midsize business) customers who don't have domain-network environments. Solving that requirement is tricky."

"There (should be) an extensive QA verification process in place that includes Firefox test scenarios and a working collaboration with Mozilla. One proven approach to improve the code quality of external components is to establish a publicly accessible validation test framework that provides the test scenarios an extension has to pass and where test scenarios are updated, based on observances with field issues."

"If they can provide an API (e.g., callback) that will be available only for registered whitelisted DLLs, we can move to that model instead of our current model and reduce even more compatibilities issues."

Based on their answers, besides the extension API, third-party software vendors believe that DLL injection should also be kept as an option since it is legally supported by the operating system. The collaboration between host and third-party software is necessary to ensure the quality of an ecosystem. Particularly, a publicly accessible validation test framework can help standardize the QA for both parties. Moreover, the upcoming whitelist seems to be a favourable solution for some third-party vendors.

Whether suggest users to uninstall third-party software after a crash (Q9).
We then were curious to know the opinions of third-party vendors on the practice that some host software (*e.g.*, Chrome [114]) will suggest users to uninstall third-party software after a crash. We received a favourable opinion *"If an app crashes on your machine then sure uninstall it. Makes complete sense. Not all machines are created equal."* versus multiple against opinions *"I consider this generally to be a bad practice, especially when a crash can't be clearly attributed to a particular third-party software – which is usually not possible in an automated way."* *"They put their customers at risk, since the legitimate (e.g., antivirus) will be removed ... If I were malware, I will use this functionality to ask users to remove any 3rd*

party mechanisms that prevent me from doing whatever I need." "Uninstalling third-party solution isn't a long term solution."

From the answers, we can see that this is a complex problem. First, such suggestions may become false alarms to users because a host vendor cannot simply decide whether a crash is due to the third-party or the host software itself. Second, in the Mozilla ecosystem, many crashes are caused by antivirus software. If such antivirus software is uninstalled, malware may take advantage of this. **Facing a third-party software related crash, we suggest that host vendors warn users about the potential risks of running the third-party software (*e.g.*, by showing the number of crashes) but also remind them of the risks of removing it.** Besides, host vendors should investigate whether the crash happens with other equivalent host software. Moreover, host vendors should always make efforts to improve the reliability of their platform if necessary, because if users value the importance of the third-party software and find it working well with other hosts, they may uninstall the host software instead.

Incompatibilities between host and third-party software (Q10, Q11).

Q10 and Q11 are about the way to prevent incompatibilities between host and third-party software when the host software rolls out new content security features. Our participant suggested: *"Notify us like they did when there is an issue. Worked well last time. We have a fix rolled out very quickly when we were made aware of the issue."*

"Browser vendors can closely work with security vendors to bring about more stable, secure browser ecosystem."

"A preview of such functionality to test it in our labs will be highly appreciated (with enough leeway and documentation to have the time for the vendors to adapt their code)."

In the meanwhile, the participants told us that they are willing to take the following measures from their part. *"We always try and fix any issues with our software when they are reported to us. We do this as soon as we were alerted to the problem."*

"Regular compatibility testing of latest aurora/beta releases of various browsers from our side along with our product and addresses any issues found."

"We are willing, and already testing, any beta and post beta releases. But if we can get documentation and enough time, we can commit to have our code ready and tested by the release date (or if push comes to shove, temporarily some remove functionality to accommodate browsers releases)."

Overall, we learnt that many third-party vendors are making efforts on compatibility testing and bug fixing for each (pre-) release. **A good communication between host and**

third-party software would help to reduce incompatibilities due to new security features. Mozilla can provide some preview and necessary documentation of the new features to the trusted (*i.e.*, whitelisted) vendors (for compatibility testing) before the features are released to users.

Blocking of third-party DLLs (Q12).

Blocking third-party DLLs is one the of measures host software is using. Let us look at the potential side effects analyzed by third-party vendors.

"Our users would not be able to target FireFox ... and would probably use another browser."

"Practically I wouldn't anticipate any side effects, although theoretically it could affect the stability of Firefox, our software products or even the whole operating system."

"It will break our protections and cause frauds associated with the removed protections, can crash our browser components and probably Firefox as well."

"This will break our ability to scan HTTPS URLs for malware/phishing links."

Again, according to the respondents, blocking DLLs would not be the best way to resolve DLL bugs. Before doing this, host software vendors should be aware of any potential and serious side effects. This is the reason why in Mozilla's blocklisting policy the blocks are always applied after careful consideration and testing, and also why outright blocking might pose problems if not handled well.

Enforcing a whitelist (Q13, Q14).

Some host software vendors are considering to put the DLLs into the whitelist if the DLL software is also using the standard extension API.

On the one hand, some third-party vendors agreed that such whitelist bonus is an incentive for them to adopt the extension API, but these vendors have already considered/started to migrate to the API. *"Yes ... (the whitelist bonus will be) along with the ability to enforce addons in certain scenarios." "We already adapting to the best of our ability the WebExtension API. We also moved to that methods on other browsers."*

On the other hand, some others are not interested in this bonus because *"I am unaware that we can extract audio from a browser using this API"* and *"The WebExtensions API has simply different use cases than the ones we are currently implementing. Therefore I don't think it makes sense to mix that up"*. The benefit of the whitelist bonus still needs to be verified in the future.

Some participants agreed that the existence of a whitelist will be an incentive for them to do more QA. For the two participant who did not agree, one thought that their *"current*

QA processes are sufficient". The other one absolutely denied potential benefits from the whitelist: *"A whitelist approach is inferior as it holds back the extension ecosystem overall, in my opinion. A proactive approach providing extensive and frequently updated test scenario framework support covering known problematic techniques is superior."* Therefore, we also need future evidences to answer this question.

Bypassing the whitelist (Q15).

About our last question, no participant plans to circumvent the whitelist, even for the vendors who insist to use DLL injection.

"No, because it won't be a long term solution."

"We would not for legal reasons. We do not circumnavigate anything." "This question is quite hypothetical right now. Likely we would respect Firefox's policy and not try to actively circumvent anything like this by technical means, but instead we may notify our users about this and suggest to move to another browser. Depending on the exact method of implementation, it's questionable if we'd be affected by such a whitelist though."

"If we will be on the white list, why should we (circumvent it)?"

However, we do not know whether malware producers would try to circumvent the whitelist (our guess is that they probably would), since we are not able to contact any of them. Also, we cannot be sure that the answers to this question are actually honest, given that circumventing the whitelist might be illegal and would be a direct challenge against Mozilla. Clearing out this doubt will be a part of our future work, once we collect enough field data on the whitelist.

5.4 Discussion

In a software ecosystem, pursuing user satisfaction is one of the most important goals for both host and third-party vendors. However, to achieve this goal, some host and guest vendors are taking conflicting measures. In the previous section, we have observed that, on the one hand, some host vendors are (even completely) blocking third-party software added through DLL injection and are suggesting users to uninstall unreliable software. On the other hand, some third-party vendors are not willing to adopt host vendors' advices and new solutions because once their extensions cannot work with the host software, they claim that they will suggest users to migrate to another host. We believe that in an ecosystem, host and third-party vendors should not consider their benefit as a zero-sum game, but a win-win game. To satisfy and hold their common users, host and third-party vendors should strengthen their collaboration along all aspects of the development of the ecosystem, including (but not limited to) testing, bug fixing, feature introducing, and API evolution.

In this work, we choose DLL injection as subject because some host software vendors realize that this technique often caused bugs (even crashes) and can be exploited by attackers. However, besides DLL injection and a standard extension API, there are other ways to add third party code into another software, such as Flash. As a resource consuming and outdated technique, Flash has been made "click-to-play" in both Firefox and Chrome since 2017, and will be completely blocked in all browsers by 2019 (2020 for Firefox ESR), so we do not study it in our work. Comparing the reliability among different extension techniques will be a part of our future work.

5.5 Threats to Validity

Construct validity threats are concerned with the relationship between theory and observation. Studying DLL injection bugs in an ecosystem is a new research topic. As far as we know, there has not been a theory behind this. However, before conducting the empirical study, we learnt some assumptions through our contact with Mozilla developers, but observed opposing results. For example, some Mozilla developers thought that the WebExtensions API can fulfill most of the purposes. They guessed that some third-party vendors are not willing to migrate to the API because the vendors do not want to spend time to modify their existing code. However, multiple of our survey participants indicated that their purposes cannot be satisfied by the current WebExtensions API. Moreover, to reduce DLL injection bugs, host vendors are taking measures, *e.g.*, blocking DLL injection, suggesting users to uninstall "unreliable" extensions. By analyzing feedback from third-party vendors, we realize that many of these measures could be harmful for end users and even the host vendors themselves.

Internal validity threats concern factors that may affect a dependent variable and were not considered in the study. Some of our observations derived from the 12 survey responses. Although these responses cannot represent all third-party vendors' opinions, they provided us valuable information to understand the root causes of the DLL injection bugs and to propose potential solutions to reduce the bugs occurrence. The most important reason is that such information cannot be discovered from any open source repositories, such as Mozilla bug reports, crash reports, or commit logs. Besides, we studied all the 103 DLL injection bugs reported during the past two years. These bugs were caused by 58 different vendors, among which, 44 vendors were contacted. 12 survey participants represent a 21% coverage of all subject third-party vendors and 27% survey response rate (which is higher than the average response rate in questionnaire-based software engineering studies, *i.e.*, 5%, according to Singer et al.'s finding [116]).

Conclusion validity threats concern the relationship between the treatments and the outcome. When investigating the characteristics of the DLL injection bugs, we manually classified DLL bugs into different categories. To reduce any biases during this process, we did not predefine any category. For each characteristic, two of the authors independently made their classification before comparing their results and resolving each of the discrepancies. Despite this, we cannot guarantee a 100% accuracy on our classification result. To help future studies validate our result, we share our dataset online at: `https://github.com/swatlab/dll_injection`. Some of the important observations are based on the survey responses. To reduce any possible biases, besides our discussion and analyses, we cited participants original answers. Readers can use this information to validate our conclusion and discover more insight. When compiling the survey responses, we hid some details due to privacy reasons. For example, we did not make a table showing which participant answered which question because this way may disclose information that participants do not wish to publish. In the survey, we only use open questions, because first, our subject problem has not been empirically studied before, *i.e.*, there is no reference to help us predefine options for the answers. Second, predefined answers may bias and limit participants' judgement. In this work, we are open to receive any unexpected ideas that can lead us to a better understanding of the subject problem.

External validity threats are concerned with the generalizability of our results. In this work, we choose Mozilla Firefox as subject ecosystem because other equivalent ecosystems either lack relevant data or will try to completely block DLL injection soon (*e.g.*, Chrome). We believe that Firefox is a large-scale representative ecosystem, which contains various and diverse DLL software (refer to the software types discussed in **RQ1**). In addition, Firefox possesses some public resources that we cannot benefit from other host vendors, such as bug reports, where we can also often see decision processes in play, and third-party vendors' contacts. Nevertheless, the results and conclusion of our work may not be generalized to other environments. Future studies are required to validate and complement our findings. Researchers can also use our shared dataset to replicate this study: `https://github.com/swatlab/dll_injection`.

5.6 Summary

In a software ecosystem, DLL injection allows third-party software to forcibly load arbitrary code into the host software. This technique may cause severe problems, such as crashes and hangs. In this work, we quantitatively and qualitatively studied DLL injection bugs in the Firefox ecosystem. We found that: most of the subject bugs (93 bugs, *i.e.*, 90.3%)

led to crashes, and 57 (55.3%) of them were caused by antivirus software (**RQ1**). Various DLL injection mechanisms were applied by third-party vendors; the triggers of the bugs can be engine errors, compiler/runtime incompatibility, or version incompatibility between the host and third-party software (**RQ2**). Completely banning DLL injection might not be the best strategy because some software (*e.g.*, antivirus) relies on this technique. Collaboration between host and third-party software vendors could help reduce DLL injection bugs; host software vendors should extend the features of the extension API (as a safer alternative of adding functionalities onto the host software) and build a publicly accessible validation test framework (**RQ3**). In the future, we plan to investigate whether the upcoming whitelist can further help reduce DLL injection bugs.

CHAPTER 6 CONCLUSION

In this chapter, we conclude the thesis and summarize our findings. In addition, we will discuss the limitations of our proposed approaches and the directions for future work.

6.1 Summary

Release engineering is a sub-discipline of software engineering that aims at improving software release strategies; helping software organizations increase user perceived quality and reduce maintenance cost. Although release engineering problems have been extensively studied by previous researchers, some important issues have never been systematically investigated yet. Specifically, software practitioners would be curious to know the reasons why crash-prone code was missed in the code review process; how to effectively release urgent patches; and how to prevent DLL injection bugs. In this thesis, we conducted empirical studies on Mozilla Firefox against the above issues, and proposed solutions to academic and industrial practitioners.

In the rest of this section, we review our subject problems and summarize our findings and implications.

Why was crash-prone code missed in the code review process?

Nowadays, most software organizations are investing time and human resources to conduct code review; expecting to improve the quality of their source code. However, severe bugs, especially crashes, can still elude from this process. In the first part of this thesis (Chapter 3), we identified reviewed code that were implicated into crashes. We compared the characteristics of such crash-prone code with reviewed code that did not crash. We also conducted a manual inspection on a sample of crash-prone patches that have been peer reviewed with the aim of understanding the intensions of these patches and the root causes of their crashes.

As a result, we found that crash-prone code tends to be more complex and be involved with more complicated dependencies. Developers tend to spend longer time to review and discuss on the crash-prone code than other code. Most of the inspected crash-prone code is used to improve performance, refactor code, add functionality, or fix previous crashes. Memory and semantic errors were identified as major root causes of the crashes.

Based on these findings, we suggest that software organizations should more carefully approve complex changes and changes with complicated dependencies. Static analysis tools should be applied to help reviewers find more memory and concurrency related errors.

How can we effectively release urgent patches?

At Mozilla, patches introducing important features or fixing severe faults cannot wait for the standard release process and are promoted directly from the development channel (*i.e.*, Nightly) to one of the stable channels (*i.e.*, Aurora, Beta, or Release). This process is called patch uplift. Patch uplift is risky because the time allowed for the stabilization of uplifted patches is reduced. Such a rushed operation can lead to regressions in the code. In the second part of this thesis (Chapter 4), we studied the uplifted patches that did not address the subject problems and that introduced regressions. We conducted a series of statistical and manual analyses to understand the reasons behind patch uplift decisions, the root causes of ineffective uplifts, the characteristics of uplifted patches that led to regressions, and whether such regressions can be prevented.

As a result, we observed that most uplifted patches were used to fix wrong functionality or a crash. 4% of the subject uplifts were identified as ineffective because they only partially fixed the expected problems or caused regressions. Regression introducing uplifts tend to have larger patch size, and most of their faults are due to semantic or memory errors. We also found that release managers are more inclined to accept uplift requests that concern certain specific components, and–or that are submitted by certain specific developers. In addition, more than 25% of the regressions due to Beta or Release uplifts could have been prevented because they could be reproduced by developers or found in widely used feature/website/configuration or via Mozilla's telemetry.

We suggest that software organizations take our findings and suggestions as a reference to improve their patch uplift strategies.

How can we prevent DLL injection bugs?

As many other browsers, Firefox allows other software to extend its functionalities. DLL injection is a technique that allows third-party software to run its code within the address space of another process by forcing the load of a dynamic-link library. However, this technique can be very risky and is not recommended by Mozilla because it can lead to arbitrary consequences, such as severe crashes and vulnerabilities. In the third part of this thesis (Chapter 5), we investigated bugs of Firefox that were caused by third-party software via DLL injection.

We observed that 90% of the studied DLL injections bugs led to crashes and 55.3% of them were caused by antivirus software. We sent surveys to the software vendors who introduced these bugs and learnt that some vendors did not perform any QA with pre-release versions nor intend to use a recommended technique (the WebExtensions API) but insist on using

DLL injection.

To reduce DLL injection bugs, we suggest that host software vendors strengthen the collaboration with third-party vendors. For example, they can build a publicly accessible validation test framework to prevent from DLL injection bugs. Host software vendors could also maintain a whitelist, which only allows vetted DLLs to inject.

6.2 Limitations of this thesis

- In this thesis, we only used Mozilla Firefox as the subject system. Although Firefox is a representative, large-scale software system, which provides a public crash database and allows us to access its uplift data, we cannot guarantee that our findings are generalizable to other software systems. Future replication studies are required to validate our approaches and results.
- In the first and second parts of this thesis, we used a heuristic (SZZ algorithm) to identify bug- or crash-inducing commits. The basic idea of this heuristic is that the bug-inducing code is fixed in the bug-fixing patches. Although this heuristic is considered as the state-of-the-art approach, it may still yield a lot of noises. For example, some bugs cannot be located from the bug fixing patch. Future researchers can apply some proposed improvement on this heuristic (such as [117]) or perform a manual validation on a sample of the yielded results.
- A part of our conclusions relies on sampled manual analyses or surveys. Future replications with larger sample size or larger number of survey participants are welcomed to make our conclusions more generalizable.

6.3 Future work

In the future, we plan to extend our study in the following directions:

- To validate our findings, we are replicating the study of patch uplift on a closed source software system in the game industry. As in this thesis, we plan to study the uplift decisions, the characteristics of buggy uplifts, and find out the root causes of the uplifted patches. The challenge is that the closed source system is from another industry other than web browsers and its uplift process is different from the one of Mozilla. Adjusting our analytic approaches might be required to obtain sound results. Another challenge is that we need to explain the differences of findings between Firefox and the closed source system.

- We plan to conduct another study on code review: whether similar patches received similar review decisions. The results of this study can help us to understand whether code reviewers make consistent decisions when facing a group of similar patches. The results can also help to build a review recommendation system that prevent reviewers from providing wrong decisions.
- We also plan to follow up the DLL injection work. We are aware that Mozilla is already planning to deploy a whitelist to allow only healthy DLL to inject. In this follow-up study, we want to verify the effectiveness of this whitelist. In addition, as aforementioned, Chrome has completely banned DLL injection from third-party software. We want to investigate the alternatives that third-party software vendors take, *e.g.*, whether they switch their code to the recommended API or migrate to another ecosystem, *e.g.*, from Chrome to Firefox.

REFEFENCES

[1] J. Scott, *Social network analysis.* Sage, 2012.

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

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

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

[5] Y. Jiang, B. Adams, and D. M. German, "Will my patch make it? and how fast?: Case study on the linux kernel," in *Proceedings of the 10th Working Conference on Mining Software Repositories.* IEEE Press, 2013, pp. 101–110.

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

[7] G. K. Hanssen, "A longitudinal case study of an emerging software ecosystem: Implications for practice and theory," *Journal of Systems and Software*, vol. 85, no. 7, pp. 1455–1466, 2012.

[8] F. Khomh, B. Adams, T. Dhaliwal, and Y. Zou, "Understanding the impact of rapid releases on software quality," *Empirical Software Engineering*, pp. 1–38, 2014.

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

[10] B. Adams and S. McIntosh, "Modern release engineering in a nutshell–why researchers should care," in *Software Analysis, Evolution, and Reengineering (SANER), 2016 IEEE 23rd International Conference on*, vol. 5. IEEE, 2016, pp. 78–90.

[11] S. McIntosh, Y. Kamei, B. Adams, and A. E. Hassan, "The impact of code review coverage and code review participation on software quality: A case study of the qt, vtk,

and itk projects," in *Proceedings of the 11th Working Conference on Mining Software Repositories.* ACM, 2014, pp. 192–201.

[12] ——, "An empirical study of the impact of modern code review practices on software quality," *Empirical Software Engineering*, vol. 21, no. 5, pp. 2146–2189, 2016.

[13] R. Morales, S. McIntosh, and F. Khomh, "Do code review practices impact design quality? a case study of the qt, vtk, and itk projects," in *Software Analysis, Evolution and Reengineering (SANER), 2015 IEEE 22nd International Conference on.* IEEE, 2015, pp. 171–180.

[14] P. Thongtanunam, S. McIntosh, A. E. Hassan, and H. Iida, "Investigating code review practices in defective files: An empirical study of the qt system," in *Proc. of the 12th Working Conf. on Mining Software Repositories (MSR)*, 2015, pp. 168–179.

[15] O. Kononenko, O. Baysal, L. Guerrouj, Y. Cao, and M. W. Godfrey, "Investigating code review quality: Do people and participation matter?" in *Software Maintenance and Evolution (ICSME), 2015 IEEE International Conference on.* IEEE, 2015, pp. 111–120.

[16] C. Sadowski, E. Söderberg, L. Church, M. Sipko, and A. Bacchelli, "Modern code review: a case study at google," in *Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice.* ACM, 2018, pp. 181–190.

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

[18] A. Podgurski, D. Leon, P. Francis, W. Masri, M. Minch, J. Sun, and B. Wang, "Automated support for classifying software failure reports," in *Software Engineering, 2003. Proceedings. 25th International Conference on.* IEEE, 2003, pp. 465–475.

[19] 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 *Reverse Engineering (WCRE), 2011 18th Working Conference on.* IEEE, 2011, pp. 261–270.

[20] R. Wu, M. Wen, S.-C. Cheung, and H. Zhang, "Changelocator: locate crash-inducing changes based on crash reports," *Empirical Software Engineering*, pp. 1–35, 2017.

[21] "Socorro: Mozilla's Crash Reporting Server," https://crash-stats.mozilla.com/home/products/Firefox, 2016, online; Accessed March 31st, 2016.

[22] S. Wang, F. Khomh, and Y. Zou, "Improving bug management using correlations in crash reports," *Empirical Software Engineering*, pp. 1–31, 2014.

[23] J. Śliwerski, T. Zimmermann, and A. Zeller, “When do changes induce fixes?” in *ACM sigsoft software engineering notes*, vol. 30. ACM, 2005, pp. 1–5.

[24] Y. Dang, R. Wu, H. Zhang, D. Zhang, and P. Nobel, “Rebucket: a method for clustering duplicate crash reports based on call stack similarity,” in *Proceedings of the 34th International Conference on Software Engineering (ICSE).* IEEE Press, 2012, pp. 1084–1093.

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

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

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

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

[29] J. Bosch, “From software product lines to software ecosystems,” in *Proceedings of the 13th international software product line conference.* Carnegie Mellon University, 2009, pp. 111–119.

[30] S. Jansen, A. Finkelstein, and S. Brinkkemper, “A sense of community: A research agenda for software ecosystems,” in *Software Engineering-Companion Volume, 2009. ICSE-Companion 2009. 31st International Conference on.* IEEE, 2009, pp. 187–190.

[31] I. Van Den Berk, S. Jansen, and L. Luinenburg, “Software ecosystems: a software ecosystem strategy assessment model,” in *Proceedings of the Fourth European Conference on Software Architecture: Companion Volume.* ACM, 2010, pp. 127–134.

[32] Q. Tu *et al.*, “Evolution in open source software: A case study,” in *Software Maintenance, 2000. Proceedings. International Conference on.* IEEE, 2000, pp. 131–142.

[33] D. M. German, J. M. Gonzalez-Barahona, and G. Robles, “A model to understand the building and running inter-dependencies of software,” in *Reverse Engineering, 2007. WCRE 2007. 14th Working Conference on.* IEEE, 2007, pp. 140–149.

[34] J. M. Gonzalez-Barahona, G. Robles, M. Michlmayr, J. J. Amor, and D. M. German, “Macro-level software evolution: a case study of a large software compilation,” *Empirical Software Engineering*, vol. 14, no. 3, pp. 262–285, 2009.

[35] M. Wermelinger and Y. Yu, “Analyzing the evolution of eclipse plugins,” in *Proceedings*

of the 2008 international working conference on Mining software repositories. ACM, 2008, pp. 133–136.

[36] J. Businge, A. Serebrenik, and M. van den Brand, "An empirical study of the evolution of eclipse third-party plug-ins," in *Proceedings of the Joint ERCIM Workshop on Software Evolution (EVOL) and International Workshop on Principles of Software Evolution (IWPSE).* ACM, 2010, pp. 63–72.

[37] D. M. German, B. Adams, and A. E. Hassan, "The evolution of the r software ecosystem," in *Software Maintenance and Reengineering (CSMR), 2013 17th European Conference on.* IEEE, 2013, pp. 243–252.

[38] L. Liu, X. Zhang, G. Yan, S. Chen *et al.*, "Chrome extensions: Threat analysis and countermeasures." in *NDSS*, 2012.

[39] R. Karim, M. Dhawan, V. Ganapathy, and C.-c. Shan, "An analysis of the mozilla jetpack extension framework," in *European Conference on Object-Oriented Programming.* Springer, 2012, pp. 333–355.

[40] S. Andersson, A. Clark, G. Mohay, B. Schatz, and J. Zimmermann, "A framework for detecting network-based code injection attacks targeting windows and unix," in *Computer Security Applications Conference, 21st Annual.* IEEE, 2005, pp. 10–pp.

[41] "Code injection," https://en.wikipedia.org/wiki/Code_injection, 2018, online; Accessed April 12th, 2018.

[42] L.-c. Lam, Y. Yu, and T.-c. Chiueh, "Secure mobile code execution service." in *LISA*, 2006, pp. 53–62.

[43] J. Berdajs and Z. Bosnić, "Extending applications using an advanced approach to dll injection and api hooking," *Software: Practice and Experience*, vol. 40, no. 7, pp. 567–584, 2010.

[44] "API hooking," http://resources.infosecinstitute.com/api-hooking, 2014, online; Accessed April 12th, 2018.

[45] "DLL injection," https://en.wikipedia.org/wiki/DLL_injection, 2018, online; Accessed April 12th, 2018.

[46] M. Jang, H. Kim, and Y. Yun, "Detection of dll inserted by windows malicious code," in *Convergence Information Technology, 2007. International Conference on.* IEEE, 2007, pp. 1059–1064.

[47] S. Fewer, "Reflective dll injection," *Harmony Security, Version*, vol. 1, 2008.

[48] J. Anvik, L. Hiew, and G. C. Murphy, "Who should fix this bug?" in *Proceedings of the 28th international conference on Software engineering*, ser. ICSE '06. New York, NY, USA: ACM, 2006, pp. 361–370. [Online]. Available: http://doi.acm.org/10.1145/1134285.1134336

[49] "How to submit a patch at Mozilla," https://developer.mozilla.org/en-US/docs/Mozilla/Developer_guide/How_to_Submit_a_Patch, 2017, online; Accessed May 31st, 2017.

[50] "Mozilla Reviewer Checklist," https://developer.mozilla.org/en-US/docs/Mozilla/Developer_guide/Reviewer_Checklist, 2017, online; Accessed May 31st, 2017.

[51] "Firefox code review," https://wiki.mozilla.org/Firefox/Code_Review, 2016, online; Accessed March 31st, 2016.

[52] "Super-review policy," https://www.mozilla.org/en-US/about/governance/policies/reviewers/, 2016, online; Accessed March 31st, 2016.

[53] "Creating Commits and Submitting Review Requests with ReviewBoard," http://mozilla-version-control-tools.readthedocs.io/en/latest/mozreview/commits.html, 2017, online; Accessed May 31st, 2017.

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

[55] "Socorro: Mozilla's crash reporting system," https://blog.mozilla.org/webdev/2010/05/19/socorro-mozilla-crash-reports/, 2016, online; Accessed March 31st, 2016.

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

[57] "CLOC," http://cloc.sourceforge.net, 2017, online; Accessed May 22nd, 2017.

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

[59] R. A. Hanneman and M. Riddle, "Introduction to social network methods," 2005.

[60] N. Biggs, *Algebraic graph theory.* Cambridge university press, 1993.

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

[62] T. J. McCabe, "A complexity measure," *Software Engineering, IEEE Transactions on*, no. 4, pp. 308–320, 1976.

[63] C. Bird, N. Nagappan, P. Devanbu, H. Gall, and B. Murphy, "Does distributed devel-

opment affect software quality?: an empirical case study of windows vista," *Communications of the ACM*, vol. 52, no. 8, pp. 85–93, 2009.

[64] H. Hulkko and P. Abrahamsson, "A multiple case study on the impact of pair programming on product quality," in *Proceedings of the 27th International Conference on Software Engineering (ICSM).* IEEE, 2005, pp. 495–504.

[65] P. C. Rigby, D. M. German, and M.-A. Storey, "Open source software peer review practices: a case study of the apache server," in *Proceedings of the 30th International Conference on Software Engineering (ICSE).* ACM, 2008, pp. 541–550.

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

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

[68] N. Cliff, *Ordinal methods for behavioral data analysis.* Psychology Press, 2014.

[69] R. Coe, "It's the effect size, stupid: What effect size is and why it is important," 2002.

[70] M. Pinzger and H. C. Gall, "Dynamic analysis of communication and collaboration in oss projects," in *Collaborative Software Engineering.* Springer, 2010, pp. 265–284.

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

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

[73] "Clang-Tidy tool," http://clang.llvm.org/extra/clang-tidy, 2017, online; Accessed March 31st, 2017.

[74] "Mozilla discussion on speeding up reviews," https://groups.google.com/forum/?hl=en#!msg/mozilla.dev.planning/hGX6vy5k35o/73b3Vw9GmS8J, 2017, online; Accessed May 31st, 2017.

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

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

[77] P. Runeson, M. Alexandersson, and O. Nyholm, "Detection of duplicate defect reports using natural language processing," in *Proceedings of the 29th international conference on Software Engineering.* IEEE Computer Society, 2007, pp. 499–510.

[78] C. Sun, D. Lo, X. Wang, J. Jiang, and S.-C. Khoo, “A discriminative model approach for accurate duplicate bug report retrieval,” in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1*. ACM, 2010, pp. 45–54.

[79] X. Wang, L. Zhang, T. Xie, J. Anvik, and J. Sun, “An approach to detecting duplicate bug reports using natural language and execution information,” in *Software Engineering, 2008. ICSE’08. ACM/IEEE 30th International Conference on*. IEEE, 2008, pp. 461–470.

[80] N. Jalbert and W. Weimer, “Automated duplicate detection for bug tracking systems,” in *Dependable Systems and Networks With FTCS and DCC, 2008. DSN 2008. IEEE International Conference on*. IEEE, 2008, pp. 52–61.

[81] C. Sun, D. Lo, S.-C. Khoo, and J. Jiang, “Towards more accurate retrieval of duplicate bug reports,” in *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*. IEEE Computer Society, 2011, pp. 253–262.

[82] Y. Tian, C. Sun, and D. Lo, “Improved duplicate bug report identification,” in *Software Maintenance and Reengineering (CSMR), 2012 16th European Conference on*. IEEE, 2012, pp. 385–390.

[83] S. E. Robertson, S. Walker, S. Jones, M. M. Hancock-Beaulieu, M. Gatford *et al.*, “Okapi at trec-3,” *Nist Special Publication Sp*, vol. 109, p. 109, 1995.

[84] “Mozilla Modules,” https://wiki.mozilla.org/Modules, 2018, online; Accessed September 22nd, 2018.

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

[86] P. Tourani and B. Adams, “The impact of human discussions on just-in-time quality assurance: An empirical study on openstack and eclipse,” in *Software Analysis, Evolution, and Reengineering (SANER), 2016 IEEE 23rd International Conference on*, vol. 1. IEEE, 2016, pp. 189–200.

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

[88] “Mozilla Release Management Uplift Rules,” https://wiki.mozilla.org/Release_Management/Uplift_rules, 2018, online; Accessed May 20th, 2018.

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

[90] J. Romano, J. D. Kromrey, J. Coraggio, and J. Skowronek, "Appropriate statistics for ordinal level data: Should we really be using t-test and cohen'sd for evaluating group differences on the nsse and other surveys," in *annual meeting of the Florida Association of Institutional Research*, 2006, pp. 1–33.

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

[92] "A Bug's Life," https://developer.mozilla.org/en-US/docs/Mozilla/QA/A_Bugs_Life, 2017, online; Accessed May 20th, 2018.

[93] "The Bugzilla Guide," https://www.bugzilla.org/docs/2.20/html/bugreports.html, 2017, online; Accessed May 20th, 2018.

[94] J. Park, M. Kim, B. Ray, and D.-H. Bae, "An empirical study of supplementary bug fixes," in *Proceedings of the 9th IEEE Working Conference on Mining Software Repositories*. IEEE Press, 2012, pp. 40–49.

[95] "Mozilla Tree Sheriffs - Backouts," https://wiki.mozilla.org/Sheriffing/How_To/Backouts, 2018, online; Accessed September 22nd, 2018.

[96] L. An, F. Khomh, and B. Adams, "Supplementary bug fixes vs. re-opened bugs," in *Source Code Analysis and Manipulation (SCAM), 2014 IEEE 14th International Working Conference on*. IEEE, 2014, pp. 205–214.

[97] E. Shihab, A. Ihara, Y. Kamei, W. M. Ibrahim, M. Ohira, B. Adams, A. E. Hassan, and K.-i. Matsumoto, "Studying re-opened bugs in open source software," *Empirical Software Engineering*, pp. 1–38, 2012.

[98] "Priority Field," https://wiki.mozilla.org/Bugmasters/Projects/Folk_Knowledge/Priority_Field, 2016, online; Accessed May 20th, 2018.

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

[100] M. R. Islam and M. F. Zibran, "Leveraging automated sentiment analysis in software engineering," in *Mining Software Repositories (MSR), 2017 IEEE/ACM 14th International Conference on*. IEEE, 2017, pp. 203–214.

[101] "Preventing Add-ons And Third-party Software From Loading DLLs Into Firefox," https://blog.mozilla.org/addons/2017/01/24/preventing-add-ons-third-party-software-from-loading-dlls-into-firefox/, 2018, online; Accessed November 11th, 2018.

[102] "WebExtensions API," https://wiki.mozilla.org/WebExtensions, 2017, online; Ac-

cessed April 12th, 2018.

[103] "The Future of Developing Firefox Add-ons," https://blog.mozilla.org/addons/2015/08/21/the-future-of-developing-firefox-add-ons/, 2018, online; Accessed April 16th, 2018.

[104] "Advantages of WebExtensions for Developers," https://blog.mozilla.org/addons/2016/03/14/webextensons-whats-in-it-for-developers/, 2018, online; Accessed April 16th, 2018.

[105] "Mozilla's blocklisting policy," https://wiki.mozilla.org/Blocklisting, 2018, online; Accessed April 16th, 2018.

[106] M. Castelluccio, L. An, and F. Khomh, "An empirical study of patch uplift in rapid release development pipelines," *Empirical Software Engineering*, pp. 1–37, 2018.

[107] "Bugzilla@Mozilla," https://bugzilla.mozilla.org, 2017, online; Accessed April 12th, 2018.

[108] "Mozilla Release Management Tracking Rules," https://wiki.mozilla.org/Release_Management/Release_Process, 2018, online; Accessed March 28th, 2018.

[109] "SetWinEventHook function," https://msdn.microsoft.com/en-us/library/windows/desktop/dd373640(v=vs.85).aspx, 2018, online; Accessed April 12th, 2018.

[110] "SetWindowsHookEx function," https://msdn.microsoft.com/en-us/library/windows/desktop/ms644990(v=vs.85).aspx, 2018, online; Accessed April 12th, 2018.

[111] "AppInit_DLLs in Windows 7 and Windows Server 2008 R2," https://msdn.microsoft.com/en-us/library/windows/desktop/dd744762(v=vs.85).aspx, 2018, online; Accessed April 12th, 2018.

[112] "CreateRemoteThread function," https://msdn.microsoft.com/en-us/library/windows/desktop/ms682437(v=vs.85).aspx, 2018, online; Accessed April 12th, 2018.

[113] "LoadLibrary function," https://msdn.microsoft.com/en-us/library/windows/desktop/ms684175(v=vs.85).aspx, 2018, online; Accessed April 12th, 2018.

[114] "Reducing Chrome crashes caused by third-party software," https://web.archive.org/web/20180728201546/https://blog.chromium.org/2017/11/reducing-chrome-crashes-caused-by-third.html, 2017, online; Accessed August 1st, 2018.

[115] "Windows Data Types," https://msdn.microsoft.com/en-us/library/windows/

desktop/aa383751(v=vs.85).aspx, 2018, online; Accessed April 12th, 2018.

[116] J. Singer, S. E. Sim, and T. C. Lethbridge, “Software engineering data collection for field studies,” in *Guide to Advanced Empirical Software Engineering*. Springer, 2008, pp. 9–34.

[117] D. A. da Costa, S. McIntosh, W. Shang, U. Kulesza, R. Coelho, and A. E. Hassan, “A framework for evaluating the results of the szz approach for identifying bug-introducing changes,” *IEEE Transactions on Software Engineering*, vol. 43, no. 7, pp. 641–657, 2017.