
PhD Thesis

DIETI, University of Napoli Federico II

Dottorato in Information Technology and Electrical Engineering

XXXI Ciclo

**IMPROVING SOFTWARE
ENGINEERING PROCESSES USING
MACHINE LEARNING AND DATA
MINING TECHNIQUES**

MARCO CASTELLUCCIO

Tutor

Prof. Carlo Sansone

Co-Tutor

Prof. Luisa Verdoliva

Contents

List of Tables	vii
List of Figures	ix
1 Introduction	1
1.1 Data Mining and Machine Learning in Software Engineering	1
1.2 Thesis Statement	2
1.3 Software Development Process at Mozilla	2
1.3.1 Bug sourcing and monitoring health	2
1.3.2 Bug/Feature tracking and management	3
1.3.3 Code changes development and review	3
1.3.4 Automated testing and continuous integration	4
1.3.5 Release process	4
1.4 Contributions	5
1.5 Outline	6
2 Automatically Analyzing Groups of Crashes for Finding Correlations	9
2.1 Socorro and Crash Reports	12
2.2 Automatic Analysis of Crash Groups	15
2.2.1 The Contrast Set Mining Problem	15
2.2.2 STUCCO	16
2.2.3 Domain-Specific Variations	21
2.3 Validation of Results	23
2.3.1 Deployment on Socorro	24
2.3.2 Feedback from Developers	25
2.4 Threats to Validity	27

2.5	Related Work	28
2.5.1	Automatic Crash Reporting Systems	28
2.5.2	Crash Clustering	28
2.5.3	Visualization of Crash Reports	29
2.5.4	Triaging of Crash Reports	29
2.6	Conclusion	29
2.6.1	Future Work	30
3	Why Did This Reviewed Code Crash? An Empirical Study of Mozilla Firefox	31
3.1	The Mozilla Crash Collecting System and Code Review Process	32
3.1.1	The Mozilla Crash Collection System	32
3.1.2	The Mozilla Code Review Process	34
3.2	Identifying Reviewed Code that Crashes	35
3.2.1	Identifying Crash-related Issues	35
3.2.2	Identifying Commits that are Implicated in Future Crash-related Issues	35
3.3	Case Study Design	36
3.3.1	Studied System	36
3.3.2	Data Collection	37
3.3.3	Data Extraction	38
3.4	Case Study Results	39
3.5	Threats to Validity	49
3.6	Related Work	51
3.6.1	Crash Analysis	51
3.6.2	Code Review & Software Quality	52
3.7	Conclusion	53
4	An Empirical Study of Patch Uplift in Rapid Release Development Pipelines	55
4.1	Mozilla Patch Uplift Process	57
4.2	Case Study Design	59
4.2.1	Data Collection	59
4.2.2	Data processing	60
4.3	Case Study Results	67
4.4	Threats to Validity	93

4.5	Related Work	95
4.6	Conclusion	97
5	An Empirical Study of DLL Injection Bugs in the Firefox Ecosystem	99
5.1	Introduction	99
5.2	Background	101
5.2.1	Firefox Ecosystem	101
5.2.2	Risks of DLL Injection and Countermeasures	102
5.3	Case Study Design	105
5.3.1	Data Collection	105
5.3.2	Data Processing	105
5.3.3	Survey	106
5.4	Case Study Results	108
5.4.1	(RQ1) What are the characteristics of the bugs caused by DLL injections?	108
5.4.2	(RQ2) Which factors triggered the DLL injection bugs?	113
5.4.3	(RQ3) What would be the potential solutions to reduce such DLL injection bugs?	115
5.5	Discussion	122
5.6	Threats to Validity	123
5.7	Related Work	125
5.7.1	Software Ecosystems	125
5.7.2	DLL Injection	126
5.8	Conclusion	127
6	Conclusion	129
6.1	Contributions	129
6.1.1	Automatic analysis of groups of crashes for finding correlations	129
6.1.2	Relation between code review and crashes	130
6.1.3	Patch uplift in rapid release development processes	130
6.1.4	DLL injection bugs in the Firefox ecosystem	131
6.2	Implications	131
6.2.1	Crash handling	131
6.2.2	Code review and crash-related defects	132
6.2.3	Uplift/urgent patches processes	132
6.2.4	Software ecosystems	132

List of Tables

2.1	A subset of the attributes present in a crash report	13
2.2	Example stack trace with group name	14
2.3	Example contingency table	18
2.4	Summary of the results of the validation	24
3.1	Code complexity metrics used to compare the characteristics of crash-inducing patches and clean patches	40
3.2	Social network analysis (SNA) metrics used to compare the characteristics of crash-inducing patches and clean patches	41
3.3	Review metrics used to compare the characteristics of crash-inducing patches and clean patches	42
3.4	Median metric value of crash-inducing patches and clean patches, adjusted p -value of Mann-Whitney U test, and Cliff's Delta effect size	44
3.5	Origin of the developers who reviewed clean patches and crash-inducing patches	45
3.6	Patch reasons and descriptions	47
3.7	Crash root causes and descriptions	48
4.1	Developer experience and participation metrics	64
4.2	Uplift process metrics	64
4.3	Sentiment metrics	65
4.4	Code complexity metrics	65
4.5	Code centrality (SNA) metrics	66
4.6	Accepted <i>vs.</i> rejected patch uplift candidates	68
4.7	Uplift reasons and descriptions	70

4.8	Root causes of the ineffective uplifts	76
4.9	Number of ineffective uplifts in the three channels	77
4.10	Fault-inducing uplifts <i>vs.</i> clean uplifts	81
4.11	Fault reasons and descriptions	83
4.12	Categories of uplift reasons and regression impact	86
4.13	The frequency and probability of a regression that an uplift in the Beta channel can lead to	87
4.14	The frequency and probability that an uplift in the Release channel can lead to	88
4.15	How an uplift regression is reproducible	90
4.16	How a regression was found	90
5.1	Characteristics of the bugs caused by third-party software	106
5.2	Impact of the DLL injection bugs	109
5.3	Types of the DLL injection software	109
5.4	How the DLL injection bugs were fixed	110
5.5	Statistics on the survey participants	113

List of Figures

1.1	High-level overview of the development process	2
2.1	Idealized crash analysis process	11
2.2	Dialog window presented to users when they experience a crash	12
2.3	Overview of the crash reporting system	15
2.4	Root and all possible specializations	16
2.5	Sample run of the algorithm in the context of crash reports	19
2.6	Detail of the dependency graph	22
3.1	An example of crash report in Socorro	33
3.2	An example of crash-type in Socorro	34
3.3	Number of crash-inducing commits during each tree months from March 2007 to September 2015	37
3.4	Overview of our approach to identify and analyze reviewed code that crashed in the field	38
3.5	Comparison between crash-inducing patches <i>vs.</i> clean patches	43
3.6	Distribution of the purposes of the reviewed issues that lead to crashes . .	49
3.7	Distribution of the root causes of the reviewed issues that lead to crashes	50
4.1	Number of uplifts during each month from July 2014 to August 2016 . . .	60
4.2	Overview of our data processing approach	60
4.3	Distribution of uplift reasons in Beta	71
4.4	Distribution of uplift reasons in Release	72
4.5	Root causes of the ineffective uplifts	78
4.6	Reasons of fault-inducing uplifts	84

4.7	Whether the regression an uplift caused is more severe than the problem the uplift aims to address	86
4.8	Whether the regressions caused by an uplift were reproducible	91
4.9	How the regressions caused by uplifts were found	91
5.1	An example of DLL injection performed by RoboSizer	103
5.2	Distribution of the bug fixing time	111

Introduction

1.1 Data Mining and Machine Learning in Software Engineering

The availability of large amounts of data from software development has created an area of research called mining software repositories. Researchers mine data from software repositories both to improve understanding of software development and evolution, and to empirically validate novel ideas and techniques. The first approaches were proposed by Ball *et al.* in 1997 to find clusters of files frequently changed together [12], by Graves *et al.* in 1998 to compute the effort necessary for developers to make changes [52] and by Atkins *et al.* in 1998 to evaluate the impact of tools on software quality [11].

The large amount of data collected from software processes can then be leveraged for machine learning applications. Indeed, machine learning can have a large impact in software engineering, just like it has had in other fields, supporting developers, and other actors involved in the software development process, in automating or improving parts of their work. The automation can not only make some phases of the development process less tedious or cheaper, but also more efficient (increasing the work throughput) and less prone to errors. Moreover, employing machine learning can reduce the complexity of difficult problems, enabling engineers to focus on more interesting problems rather than the basics of development. The possible avenues for usage of data mining and machine learning techniques are many, *e.g.*, they can be used to predict faults [53] or to detect important crashes before release [72].

Our aim in this dissertation is to make another step towards the use of data mining and machine learning for supporting software engineering processes, showing how the

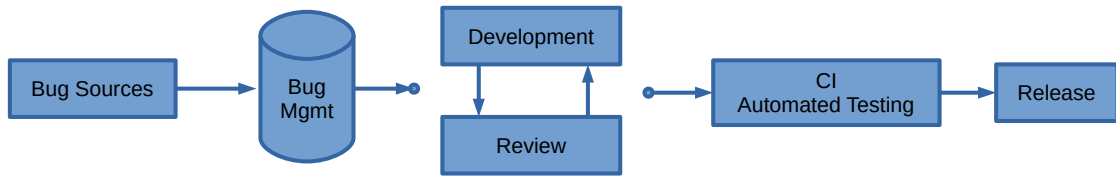


Figure 1.1. High-level overview of the development process.

development and the use of machine learning and data mining techniques can support several software engineering phases, ranging from crash handling, to code review, to patch uplifting, to software ecosystem management.

1.2 Thesis Statement

The thesis we aim to support is:

The development and the use of machine learning and data mining techniques can support several software engineering phases, ranging from crash handling, to code review, to patch uplifting, to software ecosystem management.

To validate our thesis we conducted several studies tackling different problems in an industrial open-source context, focusing on the case of Mozilla.

1.3 Software Development Process at Mozilla

The software development process at Mozilla [101] is composed of several phases, where different actors collaborate with each other to achieve the goal of releasing a new version of the software. The main actors are (i) end users, (ii) volunteers, (iii) bug triagers [100], (iv) QA, (v) developers, (vi) reviewers, (vii) managers, (viii) release managers [104], (ix) tree sheriffs [109]. An overview of the phases of the development process can be seen in Figure 1.1. The phases are not necessarily in order, but they are executed simultaneously.

1.3.1 Bug sourcing and monitoring health

There are several possible ways in which Mozilla is able to know of problems in the wild ("bug sources"):

End-user bug report An end-user could directly open a bug report in Mozilla's bug tracking system.

End-user web compatibility issue report An end-user could open a bug report in a web compatibility issue tracking system.

QA A member of the Quality Assurance team could find bugs by performing manual testing.

Support channels monitoring By monitoring support channels, Mozilla can notice issues affecting multiple users.

External services monitoring By monitoring external services (such as forums, digital distribution services, etc.), Mozilla can notice issues affecting multiple users.

Telemetry By monitoring Telemetry data, Mozilla can detect changes to the worse (*e.g.*, performance regressions).

Automated crash report An end-user can report a crash through automated means.

Not only these are possible sources of bugs, but they can also be used to monitor the health of the software, which can be measured via a set of metrics, such as number of bugs reported, number of regressions, number of crashes, performance characteristics, and so on).

1.3.2 Bug/Feature tracking and management

Once a bug is on file in Mozilla's bug tracking system (Bugzilla), triagers (either volunteers or the developers themselves) are in charge of a) requesting more information about the bug if applicable and if needed; b) resolving potential duplicates; c) moving the bug to the right component, so that it can be seen by the right team. The ultimate responsibility about triaging in a specific component is due to **triage owners** [100]. The priority of a given bug is usually decided by triage owners and developers and project managers together, but the meaning of the priority field on Bugzilla is not consistent between teams.

Bugzilla is also used to track feature work and enhancement requests (both enhancements of the product for the end user and enhancements of the quality of the code). Such bugs can be opened by users, by developers, or by project managers.

1.3.3 Code changes development and review

Once a bug has been triaged, developers might assign themselves or be assigned to work on it. The developer will then submit a patch through one of Mozilla's review

systems (at the time of writing either Phabricator [103] or Splinter [107], an interface integrated directly into Bugzilla which Mozilla has been trying to deprecate for a while). The developer will choose a reviewer for their patch, which should be a reviewer familiar with the code changed (a module owner [102]). In some cases, the developer can choose a group of reviewers, and the person who is available at the time will be the actual reviewer.

1.3.4 Automated testing and continuous integration

Once a patch has been accepted for inclusion, it will be integrated in Mozilla's version control system (Mercurial) and will go through a set of automated tests, plus static and dynamic analysis. In the normal workflow, the patch will land on an integration branch first ("mozilla-inbound" or "autoland"), where it will undergo a set of automated tests (a reduced set, using test selection techniques based on historical failures to reduce the load [65]). If the patch does not cause breakages, it will be merged, together with other patches, in the branch for the Nightly channel (called "mozilla-central"), where it will undergo the full suite of automated tests. This workflow ensures that the channel from which Nightly builds are produced is most of the time in a good state, while also reducing the load on integration branches (as opposed to running all possible tests on every possible commit in isolation). Tree sheriffs [109] are in charge of the process, both merging integration branches to release branches and ensuring the branches are always in a good state by backing out breaking changes. They also, from time to time, have to backfill tests to figure out which change is the culprit of a breakage (it is not always clear, given that on the first integration branches only a subset of the full test suites is run). For more details about landing code, the reader can refer to [108].

1.3.5 Release process

Mozilla's release process is organized around a development channel (Nightly), a set of stabilization channels (Beta and Aurora, which has been recently deprecated in the *Dawn project* [97]) and a main Release channel (each of the channels corresponds to a different branch in the version control system). Nightly is an unstable channel mainly used by early adopters and/or volunteers; Beta is a mostly-stable channel used by users which want to try features before they are released without incurring in the occasional breakage; Release is the stable build which is shipped to the general population. Each build has progressively more users than the previous, and changes in those branches are thus progressively more risky. Normally, at scheduled intervals changes are imported

from a less stable branch (*e.g.*, Nightly) to a more stable one (*e.g.*, Beta). Sometimes, high value patches are allowed to be imported directly outside the normal scheduling. Release managers are the gatekeepers of the process which allows changes to go directly from one channel to another, which is called the "uplift" process.

The work developed in this dissertation focuses in particular on the phases of crash handling, code review, patch uplift, and software ecosystem management, trying to study and, if possible, improve them by using machine learning and data mining techniques.

1.4 Contributions

The main contributions from our work to the state of the art are summarized in the following.

- In [23], we have provided documentation about some of Mozilla software engineering practices, collected during our PhD studies and our work at Mozilla.
- In [118], we conducted a study to obtain an empirical understanding of what makes a code change easier to review.
- In [28], which we presented in Chapter 2, we applied data mining techniques to a crash management problem, to automatically describe groups of crashes.
- In [7], which we presented in Chapter 3, we studied the relation between crashes and code review practices.
- In [24], which we presented in Chapter 4, we examined the uplift process operations, with the aim to characterize successful and unsuccessful uplifts. This study won a IEEE TCSE Distinguished Paper Award.
- In [25], which we presented in Chapter 4, we expanded our study on the uplift process, analyzing additional aspects of it (for example, the severity of the uplifts compared to the regression caused by them).
- In an article currently under submission, which we presented in Chapter 5, we analyzed the effects of DLL injection in the Firefox software ecosystem.

Other than software engineering, during my PhD studies I have also worked on a remote sensing application:

- In [27], we explored the use of convolutional neural networks for the semantic classification of remote sensing images, expanding our previous study [26] with the usage of a new dataset.

1.5 Outline

In **Chapter 2**, presenting our work from [28], we devised an algorithm, inspired by contrast-set mining algorithms such as STUCCO, to automatically find statistically significant properties (correlations) in crash groups. Many earlier works focused on improving the clustering of crashes, but the problem of automatically describing properties of a cluster of crashes is so far unexplored. This means developers currently spend a fair amount of time analyzing the groups themselves, which in turn means that a) they are not spending their time actually developing a fix for the crash; and b) they might miss something in their exploration of the crash data (there is a large number of attributes in crash reports and it is hard and error-prone to manually analyze everything). Our algorithm helps developers and release managers understand crash reports more easily and in an automated way, helping in pinpointing the root cause of the crash. The tool implementing the algorithm has been deployed on Mozilla’s crash reporting service.

In **Chapter 3**, presenting our work from [7], we studied the relation between crashes and reviews, given that some high-impact defects, such as crash-related ones, can elude the inspection of reviewers and escape to the field, affecting user satisfaction and increasing maintenance overhead. We investigated 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 classified a sample of reviewed crash-prone patches according to their purposes and root causes. 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. 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.

In **Chapter 4**, presenting our work from [24] and [25], we analyzed the uplift process. In rapid release development processes, patches that fix critical issues, or implement high-value features are often promoted directly from the development channel to a stabilization channel, potentially skipping one or more stabilization channels. This practice is called *patch uplift*. Patch uplift is risky, because patches that are rushed through the

stabilization phase can end up introducing regressions in the code. This chapter examines patch uplift operations at Mozilla, with the aim to identify the characteristics of the uplifted patches that did not effectively fix the targeted problem and that introduced regressions. Through statistical and manual analyses, we investigated a series of problems, 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 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.

In **Chapter 5**, presenting our work from an article that is currently under submission, we studied the effects of DLL injection in the Firefox 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. We empirically investigated bugs that were caused by third-party DLL injections into the Mozilla Firefox browser. Among the 103 studied DLL injection bugs, we found that 93 bugs (90.3%) led to crashes and 57 bugs (55.3%) 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.

Automatically Analyzing Groups of Crashes for Finding Correlations

Fixing crashes is one of the top priorities for software organizations, as they are one of the main pain points for users and might lead them to leave. Even a single crash can dramatically worsen how users perceive a software, especially if it causes the loss of important data. Acting quickly is thus really important to avoid losing users and keep a high quality software.

Several software organizations have deployed automated crash reporting systems, such as Mozilla's Socorro [4] and Windows Error Reporting [50], which are used to collect reports from users at the time of crash. A report received by Socorro comprises typically more than a hundred attribute-value fields. These reports are then analyzed by dedicated personnel to find out fixes and improve software quality. It should be realized, however, that these systems collect a huge number of crash reports daily, about three hundred thousand reports/day for Socorro, which cannot be processed on an individual basis. Therefore, the typical workflow consists of two key phases

1. crash report clustering;
2. cluster featuring and analysis.

The goal of clustering is to group together similar reports, as they are likely originated by multiple instances of the same software problem. Once the problem is fixed, all these reports can be discarded at once from further analysis. Moreover, clustering allows one to compute precious statistics on the cluster itself, enabling the second phase of the workflow. In fact, the typical features of interest in a cluster concern the frequency of

occurrence of attribute-value pairs, which may provide useful hints for the solution of the problem. As an example, assume that a perfect clustering process succeeds in grouping together all crash reports originated by a given software bug, and assume that all such reports are characterized by a distinctive feature which is never observed in reports of other clusters. While not conclusive, this observation would provide a strong clue for the analyst, and would probably allow a quick fix of the problem. This idealized process is summarized graphically in Figure 2.1.

Real-world operations are very far from this simplistic case. On/off features rarely occur, and the analyst must focus on minor variations in the frequencies of occurrence of attribute-value pairs across groups. Moreover, the most distinctive features concern usually *joint* occurrences. If the number of elementary features is already large, the number of features concerning more complex behaviors, possibly involving tuples of attribute-value pairs, makes brute force analysis simply infeasible. It requires very skilled analysts to navigate effectively through these data and extract useful clues. To further complicate things, the preliminary clustering of crashes is itself far from perfect, which may strongly affect the results of subsequent analyses. When a cluster includes reports that have no relation with one another, the resulting features are averaged together and hardly distinctive anymore. On the contrary, when there are too small groups, since reports for the same crash are divided in multiple clusters, features become unstable, leading to erroneous conclusions.

The above discussion underlines the need of effective automated tools that support the analyst's work in both phases on the process to *i*) perform a reliable clustering of crash reports, and *ii*) single out the most meaningful features. Many previous studies in the literature have focused on the first problem, namely, proposing a number of competing solutions to best cluster crashes in groups. Section 2.5 contains a more detailed explanation of some of them. In this study, instead, we focus on the second problem, and propose an automated tool to support group understanding after the clustering has already taken place. Thus, the research question we aim to address in this study is the following:

RQ: *Could an automated tool to analyze crashes be useful for developers to diagnose or help fix crashes?*

We find that in 19 out of 41 (46.3%) manually analyzed cases, the tool's results directly helped in fixing the crash. This result suggests that software organizations can use these data mining techniques to speed up and simplify the resolution of crashes and to reduce the amount of manual tedious work for developers.

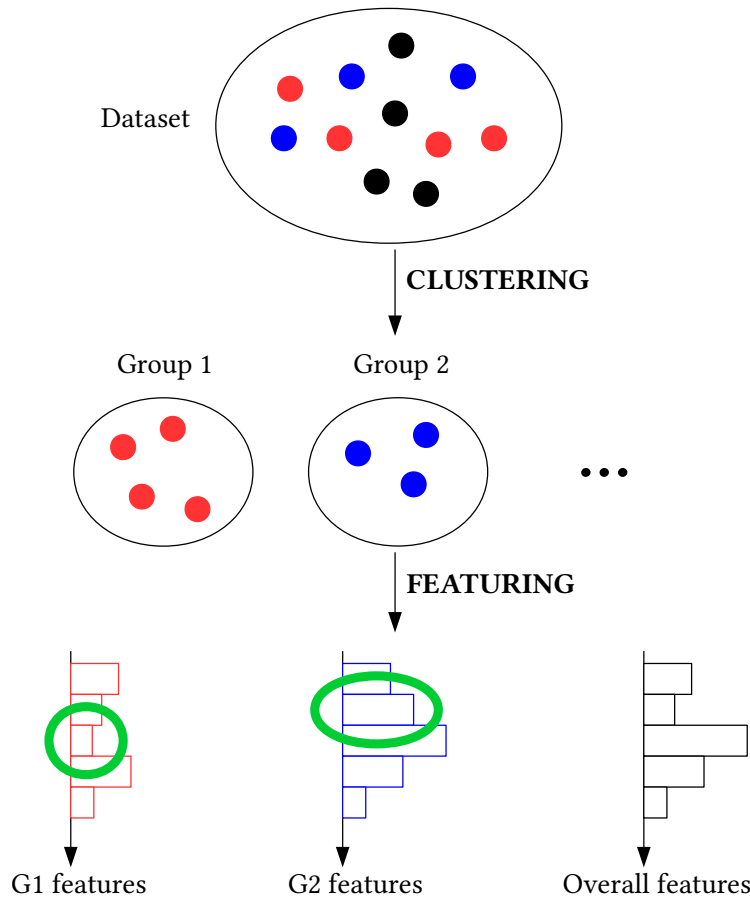


Figure 2.1. Idealized process: with perfect clustering, properties that define the groups are easily found.

The proposed tool finds statistically significant properties in crash groups, sorts them by decreasing importance, and submits them to the analyst. Developers are therefore freed from this tedious preliminary analysis, and can focus on fixing the crash. The manual analysis, given the large number of attributes in crash reports, is not only tedious but also error-prone (also due to the effects of fatigue). The proposed tool may happen to find interesting properties that the analyst could miss. Automatically finding properties of crash groups also allows release managers to quickly act with temporary workarounds, for example by blocking updates to a crashy version for a particular set of users.

Specifically, our approach is based on a data mining technique, contrast-set learning [112], applied successfully to a number of other problems in software engineering [159] and beyond (*e.g.*, [75]). The approach we present in our study can also help with the triage of crash groups, in fact release managers can decide on their importance, after

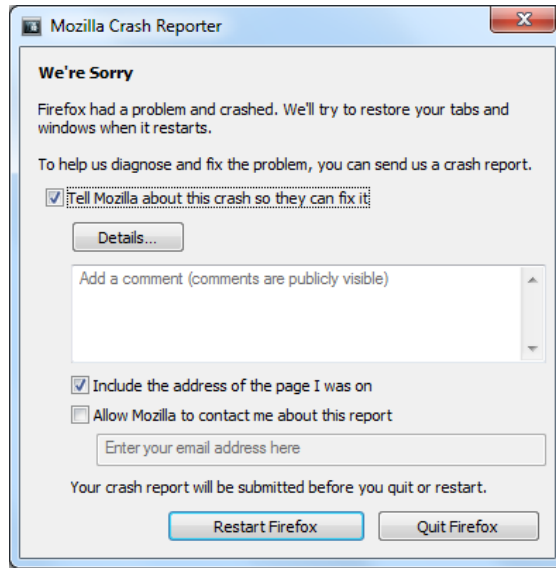


Figure 2.2. Dialog window presented to users when they experience a crash.

understanding the possible causes and properties of a crash. We evaluate the system using crash data collected from the Mozilla crash reporting system and bug tracking system. Although a systematic analysis of performance is not feasible for practical reasons, we collected evidence that the system may actually help understanding a group of crashes and reduce the time needed to solve the problem.

The rest of this chapter is organized as follows. Section 2.1 provides background information about Socorro, the Mozilla crash reporting system used in our study. Section 2.2 describes the proposed algorithm. Section 2.3 presents the validation of the results of our algorithm, applied to real world cases for Mozilla Firefox crashes. Section 2.4 discusses threats to the validity of this study. Section 2.5 summarizes related works.

2.1 Socorro and Crash Reports

Mozilla’s applications are shipped with a built-in automatic crash reporting tool [4]. When end users encounter a crash, they are presented with a dialog window that asks them to submit a report (see Figure 2.2).

Crash reports include stack traces of the threads that were running at the time of the crash and other information about the user’s environment (*e.g.*, operating system, memory-related information, modules loaded in the process). A subset of the fields contained in a crash report is depicted in Table 2.1. The reader may refer to [92] for

Table 2.1. A subset of the attributes present in a crash report.

Name	Description
Platform	The name of the Operating System.
Platform Version	The detailed version of the Operating System (<i>e.g.</i> , <i>uname -a</i> on Linux).
Addons	A list of the addons, with their version, installed in the Firefox profile.
Modules	A list of the modules (DLL files on Windows, SO files on Linux, dylib files on Mac), with their version, loaded in the application's process.
User Comment	A (usually brief) comment left by the user at the time of crashing.
CPU Info	Detailed information (vendor, family, model, stepping, number of cores) about the CPU of the user.
Adapter Vendor ID	The vendor of the graphics card on the user's machine. There are other related attributes such as Adapter Device ID, Adapter Driver Version, etc.
Safe Mode	A boolean variable that indicates whether Firefox was running in safe mode.
User Agent Locale	The language of the user.
...	...

an up-to-date JSON schema of a crash report. Some of the information contained in a crash report might be sensitive, which is why the submission of crash reports is not silent, but requires the user to accept a prompt.

As can be seen from Figure 2.2, the user has a chance to enter a short comment at the time of crash, to specify details about their crash report. For example, what they were doing right before they experienced the crash. Crash reports are then sent to the Socorro server [90], which:

1. assigns a unique ID to each report;
2. performs some post-processing on the reports;
3. groups the reports together using an extremely fast, but not very reliable, algorithm, described below.

Table 2.2. Example stack trace. The group name is in bold.

Frame	Module	Signature
0	xul.dll	mozilla::storage::Service::getSingleton()
1	xul.dll	mozilla::storage::ServiceConstructor
2	xul.dll	nsComponentManagerImpl::CreateInstanceByContractID(char const*, nsISupports*, nsID const&, void**)
3	xul.dll	nsComponentManagerImpl::GetServiceByContractID(char const*, nsID const&, void**)
4	xul.dll	nsCOMPtr_base::assign_from_gs_contractid(nsGetServiceByContractID, nsID const&)
5	xul.dll	nsCOMPtr<mozIStorageService>::nsCOMPtr<mozIStorageService>(nsGetServiceByContractID)
6	xul.dll	nsPermissionManager::OpenDatabase(nsIFile*)
7	xul.dll	nsPermissionManager::InitDB(bool)
8	xul.dll	nsPermissionManager::Init()
9	xul.dll	nsPermissionManager::GetXPCOMSingleton()
10	xul.dll	nsIPermissionManagerConstructor
11	xul.dll	nsComponentManagerImpl::CreateInstanceByContractID(char const*, nsISupports*, nsID const&, void**)
12	xul.dll	nsComponentManagerImpl::GetServiceByContractID(char const*, nsID const&, void**)
13	xul.dll	nsCOMPtr_base::assign_from_gs_contractid(nsGetServiceByContractID, nsID const&)
14	xul.dll	nsCOMPtr<nsIPermissionManager>::nsCOMPtr<nsIPermissionManager>(nsGetServiceByContractID)
15	xul.dll	mozilla::services::GetPermissionManager()
16	xul.dll	mozilla::dom::NotificationTelemetryService::RecordPermissions()
17	xul.dll	NotificationTelemetryServiceConstructor
18	xul.dll	nsComponentManagerImpl::CreateInstanceByContractID(char const*, nsISupports*, nsID const&, void**)
19	xul.dll	nsComponentManagerImpl::GetServiceByContractID(char const*, nsID const&, void**)
20	xul.dll	nsCOMPtr_base::assign_from_gs_contractid(nsGetServiceByContractID, nsID const&)
21	xul.dll	nsCOMPtr<nsISupports>::nsCOMPtr<nsISupports>(nsGetServiceByContractID)
22	xul.dll	NS_CreateServicesFromCategory(char const*, nsISupports*, char const*, char16_t const*)
23	xul.dll	nsXREDirProvider::DoStartup()
24	xul.dll	XREMain::XRE_mainRun()
25	xul.dll	XREMain::XRE_main(int, char** const, nsXREAppData const*)
26	xul.dll	XRE_main
27	firefox.exe	do_main
28	firefox.exe	wmain
29	firefox.exe	__sct_common_main_seh
30	kernel32.dll	BaseThreadInitThunk
31	ntdll.dll	__RtlUserThreadStart
32	ntdll.dll	_RtlUserThreadStart

See Figure 2.3 for an overview of the Socorro architecture.

The reports are clustered based on the top method signature of the stack trace of the crashing thread (or another thread, if the crash is due to the application willingly terminating itself after a hang). Table 2.2 shows an example of a stack trace, with the group name it was assigned by the Socorro algorithm.

There are several rules that allow to skip some methods if they are deemed to be useless for grouping purposes (*e.g.*, a very generic function, a function from an external driver, etc.). Some of the rules are general purpose (*e.g.*, C++ standard library functions), some are specific to the Mozilla applications (*e.g.*, XPCOM [93] functions). This large set of rules has been built over time, manually, by developers.

This algorithm is sometimes ineffective, as two crashes that happen in the same function might be completely different from each other. This is noticeable with crashes related to the JavaScript JIT compiler. However, processing speed is deemed more important than accuracy in this context and new clustering methods should be also very fast to qualify as a viable alternative.

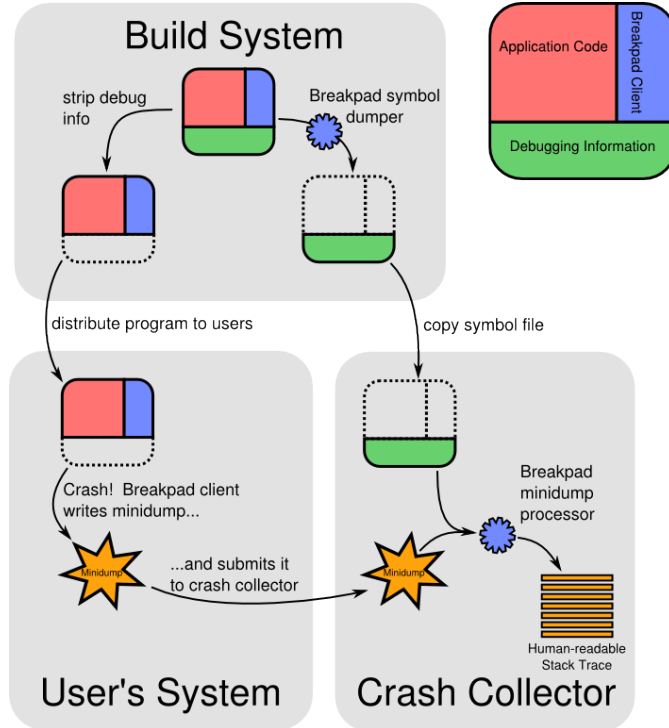


Figure 2.3. Overview of the crash reporting system.

2.2 Automatic Analysis of Crash Groups

The analysis method adopted here is a modified version of the contrast set mining algorithm STUCCO (Searching and Testing for Understandable Consistent COntRasts) proposed originally by Bay and Pazzani [14, 15]. To illustrate the method we will refer to a toy example, with the dataset partitioned in two clusters, or groups, with cardinalities $|G_1| = 700$ and $|G_2| = 300$, and reports including only $n = 2$ attributes, platform (p), and graphics card (g), which can take three and two values respectively, $p \in \{W, L, M\}$ (for Windows, Linux, and Mac) and $g \in \{N, A\}$ (for NVIDIA, and AMD).

2.2.1 The Contrast Set Mining Problem

In the contrast set mining framework, the dataset is a set of n -dimensional vectors, whose components are discrete values. The vectors are partitioned beforehand in mutually exclusive groups, G_1, G_2, \dots , according to external criteria.

A contrast-set is defined as a set of attribute-value pairs. For example, $cset1 = \{p = W\}$ is a contrast set concerning a single attribute-value pair, while $cset2 = \{p = W, g = N\}$ concerns a couple of attribute-value pairs, and is actually a *specialization* of

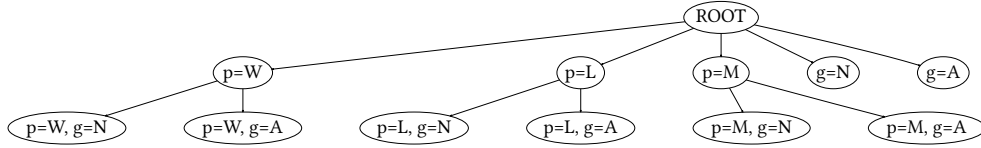


Figure 2.4. Root and all possible specializations.

the former. The support of a contrast-set in a group, $S(cset, G)$, is the percentage of vectors in the group for which the contrast-set is true. Contrast-set supports are the features used to characterize groups. So, for example, having $S(cset1, G_1) = 0.7$ and $S(cset1, G_2) = 0.3$, means that, in Group 1, 70% of crashes occurred on a Windows platform, while in Group 2 the percentage was 30%. Such a large difference seems to indicate that the platform is not irrelevant for these crashes. Accordingly, the goal of contrast-set mining is to find contrast-sets, also called *deviations*, whose support differs meaningfully across groups.

More formally, for a contrast set to be declared a deviation, it must be both *large* and *significant*. The first condition is expressed as

$$\max_{ij} |S(cset, G_i) - S(cset, G_j)| \geq \delta \quad (2.1)$$

where δ is a constant (minimum support difference) defined by the user. Significance, instead, is declared based on the outcome of a statistical test of hypotheses,

$$\begin{cases} H_0 & : P(cset = true|G_i) = P(cset = true|G_j) \\ H_1 & : P(cset = true|G_i) \neq P(cset = true|G_j) \end{cases} \quad (2.2)$$

carried out for all couples of groups, G_i, G_j , with a user-defined false alarm level, α .

2.2.2 STUCCO

In STUCCO, contrast-set mining is cast as a tree search problem. The root node is an empty contrast-set. Then, for each step of the algorithm, existing nodes are specialized by appending new attribute-value pairs to existing ones. A canonical ordering of the attributes is used to avoid visiting the same node twice. With reference to our toy example, Figure 2.4 shows the search tree after two levels of specialization. Note that the nodes $g = N$ and $g = A$ have no children, as g comes after p in our ordered attribute list.

STUCCO performs a breadth-first level-wise search in the tree. We provide a very

high-level description of the algorithm, going into more details in the following subsection. For each node at a given level, the number of occurrences for each group in the dataset is counted. Based on such data, some heuristics are applied to decide on whether the node should be pruned, become a terminal node, or generate new children. The tree grows until no more child node can be generated, or a suitable stopping condition (applied to limit processing time) is met. After the whole tree is grown, each surviving node corresponds to a valid candidate contrast-set. Contrast-sets that are found to be both large and significant (deviations), and also surprising, are eventually kept, and submitted to the analyst as an ordered list, from largest to smallest. Algorithm 1 provides a pseudo-code description of the process. Figure 2.5, instead, shows the first few steps of the algorithm applied to our toy example. In particular:

1. all possible attribute-value pairs (“candidates”) are generated for each attribute in a crash report (figure 2.5a);
2. the number of occurrences for each candidate in each group is counted (figure 2.5b);
3. some nodes are pruned based on suitable heuristics (figure 2.5c);
4. new candidates are generated by merging previous ones which survived pruning, for example $\{p = W\}$ and $\{g = N\}$ give rise to $\{p = W, g = N\}$ (figure 2.5d).

Steps 2-4 are repeated until there are no more candidates or a suitable stopping condition is met, for example, the maximum number of iterations. Eventually, all nodes/contrast-sets are tested, and only those that are large, significant, and surprising are submitted to the analyst.

The following subsections provide the necessary details for a full comprehension of the algorithm, describing the tests on largeness, significance, and surprise, as well as the heuristic rules for tree pruning.

2.2.2.1 Selecting Large Contrast-Sets

This is a straightforward test: For a contrast-set to be large, its support must be larger than the threshold, δ , defined by the user.

2.2.2.2 Selecting Significant Contrast-Sets

To evaluate whether a contrast-set is significant, we rely on the test of hypotheses of Eq.2.2. The null hypothesis is that the support of the contrast-set is equal across all

Algorithm 1: STUCCO algorithm

```

Set of candidates  $C \leftarrow \{\}$ ;
Set of deviations  $D \leftarrow \{\}$ ;
Set of pruned candidates  $P \leftarrow \{\}$ ;
Let  $prune(c)$  return True if  $c$  should be pruned;
while  $C$  is not empty do
    scan data and count support  $\forall c \in C$ ;
    foreach  $c \in C$  do
        if  $significant(c) \wedge large(c)$  then
            |  $D \leftarrow D \cup c$ 
        end
        if  $prune(c) = True$  then
            |  $P \leftarrow P \cup c$ 
        else
            |  $C_{new} \leftarrow C_{new} \cup GenChildren(c, P)$ 
        end
    end
     $C \leftarrow C_{new}$ 
end
 $D_{surprising} \leftarrow FindSurprising(D)$ 

```

Table 2.3. Example contingency table.

	$p = W$	$p \neq W$	group size
Group 1	600 (85%)	100 (15%)	700
Group 2	295 (98%)	5 (2%)	300
Overall	895 (90%)	105 (10%)	1000

groups or, differently said, it is independent of group membership. To this end, we build a contingency table like that shown in Table 2.3 reporting the occurrences of a contrast set across groups and the corresponding supports, our features of interest, that is, the frequencies of occurrence in the group.

In our example we analyze *cset1*, namely, `platform=Windows`. If group and the platform were independent variables, the proportion of crash reports with the Windows platform should be about the same across all groups. This is not the case in our example. However, the supports may differ just because of random fluctuations, and the difference may not be statistically significant. Hence, we need to determine whether such differences are the effect of a true dependency between the variables or if it can be attributed to randomness, which is why we need a statistical test. The standard test for independence

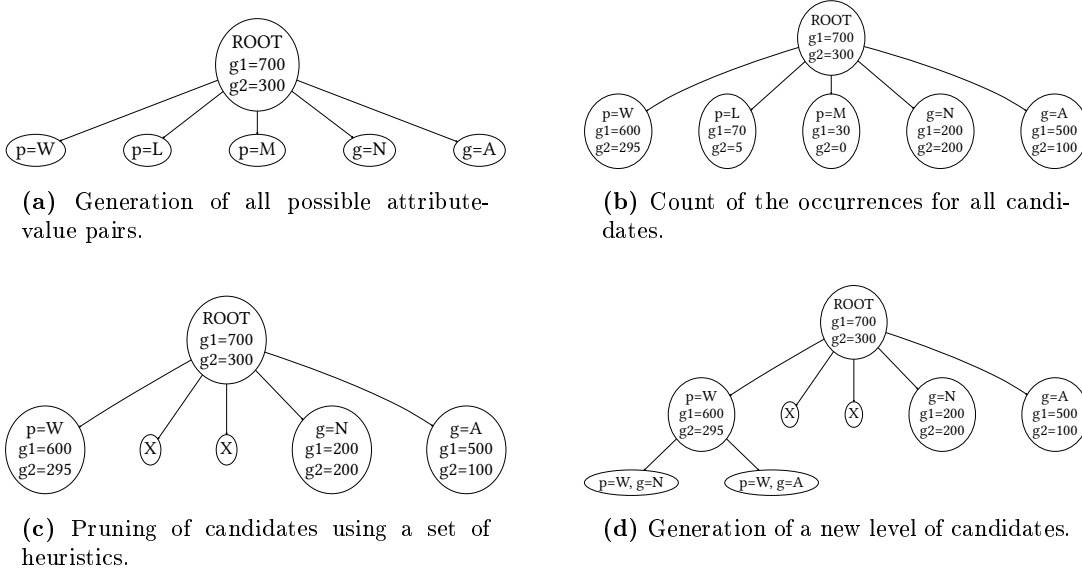


Figure 2.5. Sample run of the algorithm in the context of crash reports.

of variables in contingency tables is the chi-square test:

$$\chi^2 = \sum_{i=1}^r \sum_{j=1}^c \frac{(o_{ij} - e_{ij})^2}{e_{ij}} \quad (2.3)$$

where o_{ij} is the observed frequency in cell ij and e_{ij} is the frequency expected under the hypothesis of independence between row and column variables. We then compare the resulting value against the χ^2 distribution under the null hypothesis, selecting level of significance α , which represents the probability of rejecting the null hypothesis when it holds (false alarm).

For a single test, a level $\alpha = 0.05$, implying a false alarm probability of 5%, could be considered acceptable for our application. However, since a large number of contrast sets are typically tested for significance, the overall number of false alarms may be disturbingly large. For example, if we ran 100 tests at $\alpha = 0.05$, and the null hypothesis were always true, we would detect on the average 5 significant differences that are not actually there. To keep the false alarm rate within acceptable limits, STUCCO reduces α according to the Bonferroni correction: given H_1, H_2, \dots, H_k hypotheses, and their corresponding p -values p_1, p_2, \dots, p_k , the hypothesis H_i is rejected if $p_i < \alpha/k$. The Bonferroni correction controls the familywise error rate (FWER), which is the probability

of incorrectly rejecting at least one true hypothesis H_i , at $\leq \alpha$.

$$\begin{aligned} \text{FWER} &= P \left\{ \bigcup_{i=1}^{k_{true}} \left(p_i \leq \frac{\alpha}{k} \right) \right\} \\ &\leq \sum_{i=1}^{k_{true}} \left\{ P \left(p_i \leq \frac{\alpha}{k} \right) \right\} \leq k_{true} \frac{\alpha}{k} \leq \alpha \end{aligned} \quad (2.4)$$

This holds no matter how many of the null hypotheses are true and even with dependent tests [130].

There are two problems in the application of the Bonferroni correction in the context of STUCCO: first of all, if we report results in a level-wise fashion (shorter first, then longer), we cannot know how many tests we will perform in total, which makes it impossible to know the exact value of k . Moreover, as α gets smaller, the statistical power of the tests decreases, increasing the probability of producing false negatives. This cannot be avoided, since we want to reduce the probability of false positives. However, we can use different values of α for tests concerning different levels of the tree, ensuring a high power for tests at higher levels (which are more general and easier to understand) and accepting a lower power for tests more down the tree. Since the Bonferroni method holds as long as $\sum_i \alpha_i \leq \alpha$, STUCCO adopts level-dependent values

$$\alpha_l = \min \left(\frac{\alpha}{2^l / |C_l|}, \alpha_{l-1} \right) \quad (2.5)$$

where α_l is the cutoff for level l , and $|C_l|$ is the number of candidates at level l . This way we assign $\frac{1}{2}$ of the total α risk to tests at level 1, $\frac{1}{4}$ to tests at level 2, etc. The min rule ensures that, as we move to deeper levels, the α cutoff can only decrease, making the tests more likely not to reject the null hypothesis.

2.2.2.3 Selecting Surprising Contrast-Sets

As already said, contrast-sets are shown in a level-wise fashion given higher priority to higher levels (*e.g.*, level 1, with a single attribute-value pair) as they are easier to interpret. Further specializations are then included only if they are “surprising”, namely, when the observed frequencies depart significantly from the expected frequencies. For example, if for all G_i ’s, $S(p = W, g = N|G_i) \simeq S(p = W|G_i) \times (g = N|G_i)$, that is the support of the specialization can be derived based on an independence conjecture, than the specialization itself does not add information (is not surprising) and thus can be discarded even when it is a deviation according to the definition.

2.2.2.4 Pruning the Search Space

When building the contrast-set tree, a number of heuristics can be applied to limit its size and hence reduce the computational burden.

Minimum deviation size. When a contrast-set has support less than δ for every node, it can be pruned. In fact, if the support is smaller than δ for any given group, the difference between any two supports cannot be larger than δ .

Expected cell frequencies. The validity of a test depends on the size of the available sample, becoming scarcely reliable when only a small number of items are available. A typical lower bound for the χ^2 test is 5 [44]. Therefore, when we reach a contrast-set with a number of occurrences smaller than 5, we can safely prune it, since any further specialization can only further reduce the number of occurrences.

χ^2 bounds. Bay and Pazzani showed that it is possible to define an upper bound on the χ^2 statistic. This can be used to prune nodes, when we know that the corresponding statistic will not exceed the α cutoff.

Identical support. Specializations with the same support as the parent might be not interesting and can be discarded. They target the same set of dataset entries as the parent and often represent findings that are common knowledge (*e.g.*, the support of $\{\text{platform_detail} = \text{Debian Wheezy}\}$ will obviously be the same as the support of $\{\text{platform} = \text{Linux}, \text{platform_detail} = \text{Debian Wheezy}\}$: the addition of $\{\text{platform} = \text{Linux}\}$ provides no information).

Fixed relations. Often a group has larger support for a given contrast-set than any other group and specializing the contrast-set with additional attribute-value pairs does not change the situation. In those cases, the node can be pruned.

2.2.3 Domain-Specific Variations

The implementation of the algorithm must take into account the large number of items to deal with in our real-world application. At the time of writing, around 500000 crash reports per week are generated for a single Firefox version¹. Moreover, each report contains a large number of attributes (more than 200) spanning different possible values. This means that the number of possible candidates explodes very rapidly as soon as contrast-sets are specialized beyond level 1. Testing candidates for each couple of groups is clearly infeasible. Therefore, in our implementation, we test each group against the rest of the dataset, that is, we look for features that present anomalies w.r.t.

¹ <https://crash-stats.mozilla.com/search/?product=Firefox&version=51.0.1&date=>%3D2017-02-14&date=<2017-02-21#crash-reports>

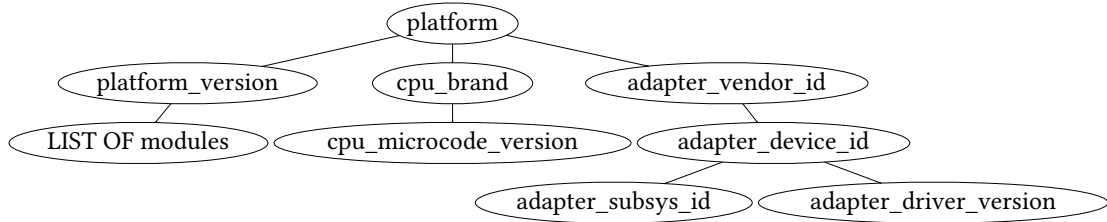


Figure 2.6. Detail of the dependency graph.

the average behavior over the whole dataset. In addition, for performance reasons, we have implemented the tool using Apache Spark [160].

Another specific feature of our application is the existence of strong dependencies among groups of attributes. For example, the presence of a given DLL might be directly linked to a particular version of Windows; the CPU microcode version is directly linked to the CPU vendor; etc. We modified STUCCO to take into account such information by means of a graph of dependencies (see Figure 2.6 for a detail of the dependency graph). When a dependency is found, the percentage of occurrence is recalculated restricting the group to the reports where the dependency holds true. For example, in a group we studied, the module “bcryptPrimitives.dll” was present in 83.9% of crash reports *vs.* 33.91% overall, qualifying for a likely deviation. However, if we take into account the operating system (Windows 10), the percentages change to 100% *vs.* 98.44%, and hence this rule could be ignored.

One of the fields of the crash reports is a small text area where the user who experiences the crash can write a short comment. Most users do not provide useful information but express only their frustration, which makes the comments field widely different from usual bug reports. Nonetheless, in our manual inspections, we have found comments to be sometimes useful, even if just as hints.

With the aim to extract some useful information from the comments field, we employed a well known information retrieval technique, term frequency and inverse document frequency (TF-IDF) [66], which highlights the words most frequently used in the comments for a given crash group *vs.* other groups. This allows developers to quickly glance if there is something wrong with a particular setting. For example, in one particular instance, many users were mentioning “playing”, and the crash turned out to be due to a resource exhaustion due to videogames running in the background.

2.3 Validation of Results

To validate the results, we have selected a set of bug reports where we knew developers used our tool and we have verified whether the tool

- helped in the resolution of the bug,
- gave compatible clues but did not help solving the bug,
- gave some misleading clues.

The tool has been integrated in Socorro, but we do not know when the developers use it for their investigations. Some developers, when using the tool, copied the results of the tool in the bug report they are working on. This allows us to select a set of real world cases that we can analyze, given that developers have fixed them already, so we can evaluate if the results of the tool have been useful for fixing the bug.

We considered about 800 crash bug reports (approximately 400 closed) generated from September 2016, when our tool has been put in production, to February 2017, mostly from Mozilla developers. For 90 of these reports (41 closed) we have definitive evidence that our tool was used. We have used regular expressions based on the output format of our tool to extract these 90 bugs out of the original 800. We have manually analyzed this set of bug reports and the code changes that are attached to them, finding 19 cases where the tool has been really useful (that is, the results of the tool directly helped the developers in fixing the bug, as they used the results to understand the root cause); 19 cases where the tool generated results that were compatible with the resolution of the bug, but did not help solving it (that is, the information provided by the tool was not useful for developers to understand the root cause); 3 cases where the tool has produced misleading results (that is, the results of the tool did not help solving the bug and were not compatible with the resolution of the bug). These results are summarized in Table 2.4.

In some of the cases where the tool has been useful, we believe the bug would not have been solved if not with very large investigative effort. Out of the three cases where the tool has been misleading, we believe that, by improving the initial clustering algorithm, two misleading results would have been avoided. These are analyzed in more detail in section 4 and 5. As already said in the Introduction, the quality of clustering can strongly affect the results of the algorithm, polluting group statistics with unrelated reports, or generating groups too small to provide meaningful statistics at all.

When the clustering algorithm fails by generating groups that are too large (clustering together crashes that have no relation with each other), it is harder for the correlation

Table 2.4. Summary of the results of the validation.

Type	Number of bugs
Very useful – results that directly helped fixing the bug.	19
Compatible – results that were compatible with the resolution of the bug, but were not useful for fixing the bug.	19
Misleading – results not compatible with the resolution of the bug.	3

tool to find interesting properties. Indeed, as many crashes which are actually really different from each other get clustered together, it gets more difficult to analyze them (both manually and automatically).

When the clustering algorithm fails by generating groups that are too small (allocating reports for the same crash to different groups), the correlation tool, and manual analysis, is more prone to find spurious correlations.

The clusters' dimensions can vary wildly between thousands of reports (the most crowded cluster contains around 20000 crashes) and a very small number of reports (even a single one). We only apply the tool to the largest 200 clusters, as they are the most important ones (after the 200th cluster, we only have clusters with less than 100 reports). These top clusters account for around 55% of all reports, but there is a very long tail of clusters with very few reports.

2.3.1 Deployment on Socorro

We tested the tool on crash groups which we already analyzed in the past, to assess its validity, and we put it in production for new crash groups. In this section, we summarize a few interesting results that we obtained during our analysis.

1. **AMD CPU Bug:** A group of crashes was found to be correlated with a particular family of AMD CPUs. We later found that the particular family of AMD CPUs that was involved in the crash group was affected by a hardware bug, and developers were able to find a workaround for it.
2. **Antivirus-Related Crash:** A group of crashes was found to be correlated with a version of an addon of an antivirus suite. In cases like this, the tool allows us to act quickly and simply block the addons (or modules) that cause problems, while we talk with the vendors to solve the problem in the long term.

3. **Crash Without AdBlock:** The tool also generates results that are quite open to interpretation. For example, there was a crash group that was more common to users without ad-blocking addons. It was a crash happening often with a very famous Flash game. We believe the crash was caused by some ad network serving particular advertisement that would cause the browser to crash. The crash disappeared quickly on its own, which supports that hypothesis.
4. **Misleading Result Caused by Clustering Failure (Too Few Clusters):** Crashes related to the JIT compiler for JavaScript are a clear example of how crash clustering can affect the results of the tool. The clustering algorithm employed by Socorro does not work well for those kind of crashes, often lumping unrelated crashes together. The correlation tool is only able to tell that the group of crashes is related to the JIT, but cannot say much more.
5. **Misleading Result Caused by Clustering Failure (Too Many Clusters):** There was a crash, which was later diagnosed to be due to concurrency issues, which was happening in different functions according to CPU brand or graphics card. This caused the clustering algorithm used by Socorro to generate a new cluster for each CPU brand / graphics card, making each cluster obviously correlated to those. Clearly, the correlations were spurious.
6. **Analyzing Crash Reports Before/After a Change:** The algorithm is really useful when analyzing a crash group generated by Socorro, but can be used for generic groups as well. For example, to analyze the differences in the properties of crash reports before/after a change, *e.g.*, to assess the effectiveness of the change and as another means to ensure that it did not cause regressions.

We employed the tool to analyze the differences between the crashes before/after a change that relaxed the blacklist for graphics acceleration on NVIDIA graphics cards. We found that the change improved the stability with a particular version of the NVIDIA drivers (one where hardware acceleration was previously blocked and unblocked by the change), probably because hardware acceleration is a more common and thoroughly tested code path.

2.3.2 Feedback from Developers

Developers and people triaging crash bugs generally expressed favourable opinions about the tool (*e.g.*, posts on one of Mozilla's mailing lists [46]). We collected suggestions from them since the deployment to Socorro. Developers and triagers were able

to provide suggestions and requests by filing bugs on Mozilla’s bug tracking system, or by contacting the author directly. Most of the suggestions were requests of addition of new possible fields to the analysis (sometimes meta-information dynamically generated from already existing fields, *e.g.*, https://bugzilla.mozilla.org/show_bug.cgi?id=1506012). Some of the suggestions were instead related to the way results are shown, which is actually a pretty important aspect. Indeed, we empirically noticed that, if the information presented to the user is too crowded (*e.g.*, too many useless attributes, too much information), the user is more likely to complain or overlook something. In the remainder of this section, we present some of the more specific suggestions that we received from developers.

1. **Employing the Correlation Results Themselves to Improve Clustering:**

The correlation analysis itself might be useful to improve the clustering algorithm. For example, two groups which present similar correlations might be clustered together. Groups which do not have any interesting correlation, might be candidates to be split.

We observed that this operation was done manually by developers in the results validation. Concerning two bugs where the correlations were very similar, the developers noticed that the two groups were actually a single one (and closed a bug as a duplicate of the other).

2. **Extract Information from Unstructured Crash Report Fields:** The algorithm we presented only works with discrete fields, but crash reports often contain unstructured information too. The user comment is a clear example. The TF-IDF solution works for simple cases and it could be greatly improved. For example, if several users mention the same thing in different ways, TF-IDF will not notice it. Using a more powerful text mining algorithm might improve the results, although it is still not clear to us how much information is actually contained in the users’ comments. We noticed some cases where it turned out to be useful, but devolving time and resources for this might not be too valuable.

3. **Driving Automated Tests Configuration:** At Mozilla, we developed a tool which automatically tries to reproduce crashes with different settings and under different configurations, called BugHunter [91]. The correlation results could help in driving the tool to directly test under a configuration that is more likely to reproduce the crash, both saving running time (*e.g.*, if a crash is only happening with a specific graphics card vendor and a specific driver, there is no point in

trying to reproduce it with a graphics card from a different vendor) and making reproducibility easier.

- 4. Predicting Volume of a Crash in a Release Channel from Pre-release Channels:** By linking the data generated by the correlation tool with data about the user population distribution, we can estimate how a crash that is affecting a pre-release version will affect the release version. The reader can refer to the work by Khomh et al. [71] for an explanation of the Firefox pipelined release model. This has been attempted in the past using machine learning techniques: in Kim et al. [72] it was used to predict which crash stack is more probable to become a “top crash” and should be fixed first. For example, Firefox Beta users are predominantly from the United States. The percentage of those users is fairly lower in Firefox Release. This means that crashes that are easily reproducible on a website that is not in the English language, are very likely to go unnoticed during the Beta cycle and explode when Firefox is released. If we had a way to re-rank the crashes considering the attributes to whom they are correlated and the incidence of those attributes in different channels, then those crashes would less likely go unnoticed.

2.4 Threats to Validity

Internal validity threats concern factors that may affect a dependent variable and were not considered in the study. We evaluated our tool on 41 closed bugs, which might not be a representative dataset. We have chosen to evaluate the results on the fixed bugs as we needed to check if the fix was compatible with the findings of the tool. We have manually analyzed the cases where the tool was used, thus reducing false positives from our regular expression search, but our search might not be complete (there could be bugs where the tool was used but where developers did not leave any evidence of it).

External validity threats are concerned with the generalizability of our results. In this study, we only evaluated the results of the tool applied to Mozilla Firefox crashes, because Mozilla has an automated crash reporting system and its crash data, bug reports and source code are publicly available. Our findings may not be generalizable to other systems.

Conclusion validity threats concern the relationship between the treatments and the outcome. We only analyzed bugs for which we were sure the tool was used, thus our conclusions on the usefulness of the tool on those bugs should be correct. Our conclusions on the rates of usefulness instead might suffer from the uncertainty about other crash

bugs, which we could not analyze because we had no way to tell whether the tool was used or not for all of them.

Reliability validity threats are concerned with the replicability of the study. To aid in future replication studies, we share the source code of our tool: <https://github.com/marco-c/crashcorrelations> and <https://github.com/mozilla-services/socorro>.

2.5 Related Work

Bird et al. [19] studied the effect of extrinsic factors on software reliability. In our experience we found evidence that corroborates their findings: there are several crashes that are due to external software badly interacting with Firefox. In our case though we often noticed security applications being the root cause of the crashes.

2.5.1 Automatic Crash Reporting Systems

Several past studies have shown how a crash reporting system, such as Socorro, can be very valuable for discovering and fixing crashes. For example, Glerum et al. [50] presented their experience with WER (Windows Error Reporting). Ahmed et al. [4] studied the Mozilla crash reporting system. One of the problems presented in [4] is the overwhelming amount of data that is made available through a crash reporting system. Our work tries to solve this problem by using data mining techniques to handle the complexity of the data and provide a way to automatically understand it.

2.5.2 Crash Clustering

The crash clustering problem has been studied extensively in the literature and is closely related to the technique presented in our work. Indeed, a good clustering technique is needed in order to avoid false positives or false negatives. Lohman et al. [81] and Modani et al. [88] adapted stop-word removal to call stacks, removing recursive calls, and using similarity measures like edit distance, longest common subsequence, and prefix matching. Bartz et al. [13] used edit distance, proposing seven types of edits assigned with different weights. Dhaliwal et al. [42] proposed a two-level grouping of crash reports, using Levenshtein distance [125] to evaluate the similarity between stack traces. Dang et al. [41] presented *ReBucket*, an algorithm for clustering crashes based on a custom method (called PDM, Position Dependent Model) that uses the position of a function in the stack trace and the offset between matched functions for calculating

the similarity between stack traces. Lerch et al. [77] proposed using a well known information retrieval technique, term frequency and inverse document frequency, to rate stack traces. Campbell et al. [22] presented an overview of several clustering algorithms, including the one presented by Lerch et al., evaluating their results in the same setting (Ubuntu Apport crashes). They found traditional information retrieval techniques to outperform techniques specifically designed for crash clustering. The proposed algorithm is strongly related to crash clustering, as it operates on clusters of crashes. Thus, its performance is directly affected by the quality of the clustering algorithm employed.

2.5.3 Visualization of Crash Reports

Another related area of research is the visualization of crash reports to aid in the understanding by developers. For example, Kim et al. [73] proposed an approach based on an aggregated graph view of multiple crashes. They also presented a way to use the crash graphs for clustering. Chan et al. [29] presented three types of graphs to analyze field testing results under three different perspectives. The above approaches could be combined with our proposed approach to improve understanding of group of crash reports.

2.5.4 Triaging of Crash Reports

Kim et al. [72] presented a machine learning technique to predict which crash stacks are more probable to become “top crashers” and should be fixed first. Khomh et al. [70] proposed an entropy evaluation approach, taking into account volume of crash groups and distribution among users, to rank the crash clusters by importance. The above approaches focused on prioritizing the groups of crash reports for bug fixing. Our approach instead identifies generic properties of the groups, which can be later used by developers and managers, not only for prioritization, but also to directly understand possible causes.

2.6 Conclusion

Crashes are one of the main pain points for users of a software. Fixing them promptly can improve the users’ perception of the quality of a software. We found that analyzing crash reports in an automated manner can help developers in fixing crashes, by removing manual analysis burden from developers, or by finding properties that would have been really difficult to find with manual analysis, or can give clues in the characterization of

crashes. Software organizations can use these data mining techniques to speed up and simplify the resolution of crashes and to reduce the amount of manual tedious work for developers.

2.6.1 Future Work

We identified two interesting directions for future work. First, as discussed in the Validation section (section 2.3), with examples in section 4 and 5, the results of the crash clustering can greatly affect the results of our tool. Thus, improvements to the clustering algorithm used by Socorro, other than being useful by themselves, would benefit our results as well. Second, it could be useful to have a dashboard to simplify finding reproducible crashes. At Mozilla, we are often helped by volunteers in reproducing crashes that are specific to some configuration that we do not have readily available. The correlation results might be useful to create a way for volunteers to automatically find the crashes that they might be able to reproduce, by showing them the crash groups that are related to their hardware or software (*e.g.*, installed addons, antivirus, etc.) configuration.

Why Did This Reviewed Code Crash? An Empirical Study of Mozilla Firefox

A software crash refers to an unexpected interruption of software functionality in an end user environment. Crashes may cause data loss and frustration for users. Frequent crashes can decrease user satisfaction and cause them to leave. 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 [1]. 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. [74] find that 54% of reviewed code changes still introduce defects in Mozilla projects.

In this study, we intend 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?

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.

The rest of the chapter is organized as follows. 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. Section 3.6 discusses related work.

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 [90]. Each crash report includes a stack trace

3.1. THE MOZILLA CRASH COLLECTING SYSTEM AND CODE REVIEW PROCESS³³

mozilla crash reports

Product: Firefox | Current Versions | Report: Overview | Super Search

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

Details | Metadata | Modules | Raw Dump | Extensions

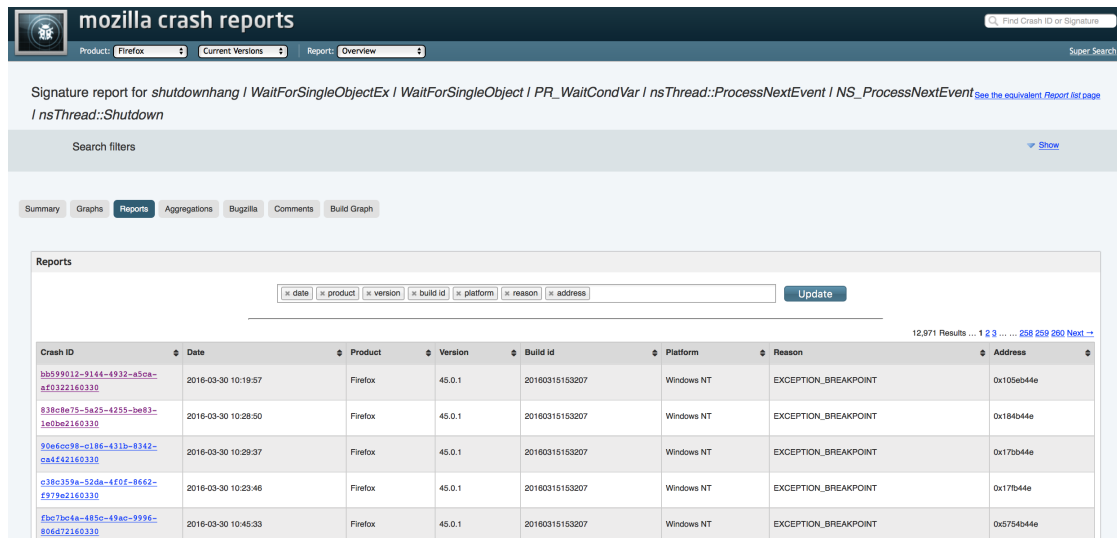
Signature	shutdownhang WaitForSingleObjectEx WaitForSingleObject PR_WaitCondVar nsThread::ProcessNextEvent NS_ProcessNextEvent nsThread::Shutdown
UUID	bb599012-9144-4932-a5ca-af0322160330
Date Processed	2016-03-30T10:19:57.623299+00:00
Uptime	4438
Last Crash	357810 seconds before submission
Install Age	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.

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 system. Quality assurance teams use Socorro to triage these crash-related issue reports and assign severity levels to them [9]. 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 [70].



Signature report for `shutdownhang | WaitForSingleObjectEx | WaitForSingleObject | PR_WaitCondVar | nsThread::ProcessNextEvent | NS_ProcessNextEvent | nsThread::Shutdown` [See the equivalent Report list page](#)

Search filters [Show](#)

Summary | Graphs | **Reports** | Aggregations | Bugzilla | Comments | Build Graph

Reports

12,971 Results ... 1 2 ... 228 229 230 Next ...

Crash ID	Date	Product	Version	Build id	Platform	Reason	Address
bb59012-9144-4932-a5ca-af0322160330	2016-03-30 10:19:57	Firefox	45.0.1	20160315153207	Windows NT	EXCEPTION_BREAKPOINT	0x105eb44e
838e8e75-5a25-4255-be83-1e0ba2160330	2016-03-30 10:28:50	Firefox	45.0.1	20160315153207	Windows NT	EXCEPTION_BREAKPOINT	0x184b44e
90e6ec98-c186-431b-8342-ca4f42160330	2016-03-30 10:29:37	Firefox	45.0.1	20160315153207	Windows NT	EXCEPTION_BREAKPOINT	0x17b44e
e38e359a-52da-4f0f-8662-f979a2160330	2016-03-30 10:23:46	Firefox	45.0.1	20160315153207	Windows NT	EXCEPTION_BREAKPOINT	0x17b44e
fbc7bc4a-485c-49ac-999c-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.

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 [58]. 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 [120], 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 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 [35]; 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 [139]. 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 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 [119]. Thus, in this study, we examine the review characteristics at the issue level. Finally, the Tree Sheriffs [109] (*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 [135]. 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 archive¹. 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 [134] to identify commits that introduce crash-related issues. First of all, we use Fischer et al.'s heuristic [47] 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}
```

¹ <https://crash-stats.mozilla.com/api/>

By using the CLOC tool [34], 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 study, 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 F 's first crash report. The remaining commits are referred to as *crash-inducing commits*.

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 this study, only the Mozilla Foundation has opened its crash data to the public [148]. It is also the reason why in most previous empirical studies of software crashes (*e.g.*, [72, 70]), researchers analyzed data from the Mozilla Socorro crash reporting system [90]. Though Wang et al. [148] studied another system, Eclipse, they could obtain crash information from the issue reports (instead of crash reports). However, the exact crash date cannot be obtained from the issue reports, which hampers our ability to apply the SZZ algorithm. Dang et al. [41] 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.

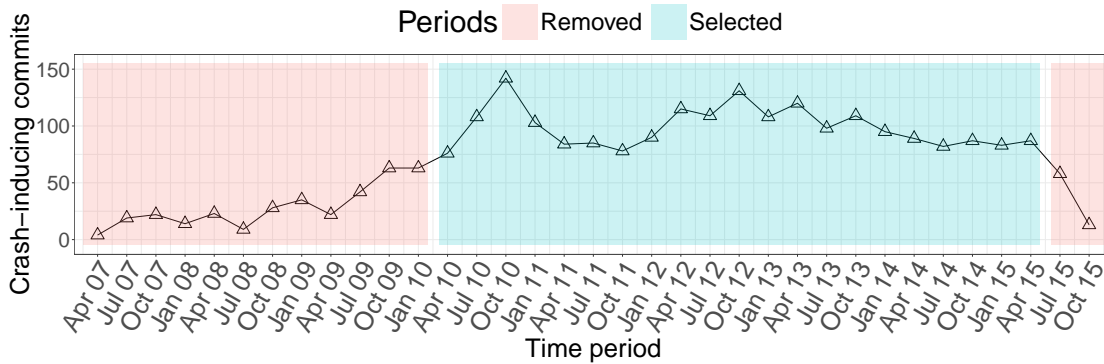


Figure 3.3. Number of crash-inducing commits during each three months from March 2007 to September 2015. Periods with low number of crash-inducing commits are removed.

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 3.3 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-inducing commits during the last months. Using Figure 3.3, 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 3.2.2, we find 1,202 (2.9%) issue reports containing reviewed patches that are implicated in crashes.

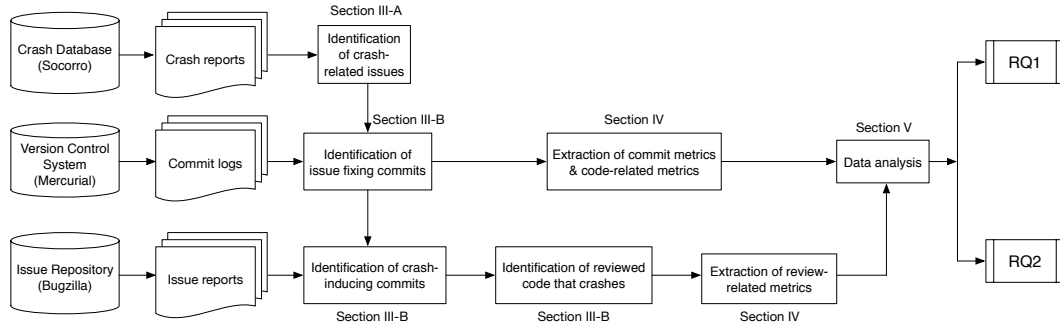


Figure 3.4. Overview of our approach to identify and analyze reviewed code that crashed in the field.

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

3.3.3.1 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 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 [142], 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 study, we remove the patch author from the reviewer

² https://github.com/swatlab/crash_review

list of each of the studied issues. More details on our review metrics are provided in Section 3.4.

3.3.3.2 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 [126]. 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.

3.3.3.3 Social Network Analysis Metrics

To measure the relationship among classes, we apply Social Network Analysis (SNA) [54] to measure the centrality [127] of each C/C++ class, *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 [17] with these vertex pairs. By using the *igraph* network analysis tool [39], 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.

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 [74].
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 [74].
LOC	Mean number of the lines of code in the changed classes to fix an issue.	Large classes are more likely to crash [72].
McCabe	Mean value of McCabe cyclomatic complexity [83] in all classes of the issue fixing commit(s).	Classes with high cyclomatic complexity are more likely to lead to crashes [72].
Function number	Mean number of functions in all classes in the issue fixing commit(s).	High number of functions indicates high code complexity [18], 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 [72].
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 [59], thus miss crash-prone code.

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 3.3.3.3) 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. [142]. In addition, we also take *obsolete* patches

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 [127] can lead to malfunctions in the dependent classes; even cause crashes [72].

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.

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* [57] to compare the differences in metric values 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 [43] to control the familywise error rate of the tests. In this study, 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* [33]. Effect size measures report on the magnitude of the difference while controlling for the confounding factor of sample size [36].

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 [114], used the suffix of an email address to determine the affiliation of a developer. However, many Mozilla employees use an email address other than

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 [142].
Number of comments	Number of comments related with a reviewed patch.	Review with a long discussion may help developers to discover more defects [142].
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 [121].
Proportion of reviewers writing comments	Number of reviewers writing comments over all reviewers.	Reviews without comments have higher likelihood of defect proneness [142, 84].
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 [121].
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.

mozilla.com in Bugzilla, when they review code. To make our results more accurate, we 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

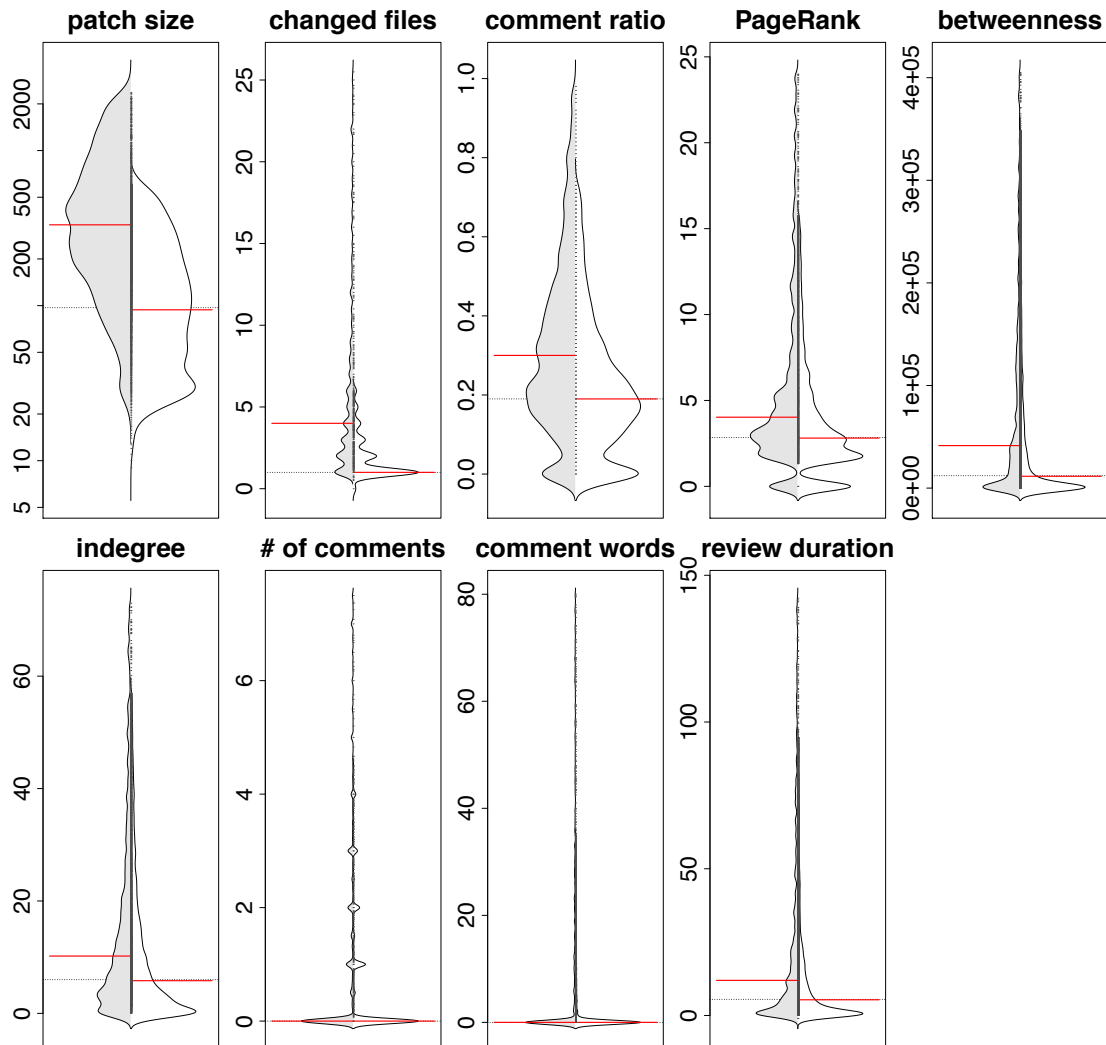


Figure 3.5. Comparison between crash-inducing patches (left part, grey) *vs.* clean patches (right part, white). Since we removed outliers from the plots, the median values may not correspond to the values in Table 3.4, which includes the outliers.

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
<i>Code complexity metrics</i>				
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)
<i>Social network analysis metrics</i>				
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)
<i>Review metrics</i>				
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	–

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

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%

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 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 [74], 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 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. On 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.

Reviewed patches that crash tend to be related with large patch size and high centrality. These patches often take a long time to be reviewed and are involved with many rounds of review discussions. More review effort should be invested on the patches with high complexity and centrality values.

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 [140], 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”. In addition, since we do not only study defect-related issues as Tan et al., 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 researchers individually classified each issue into one or more

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/plugin 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, <i>e.g.</i> , data corruption and adding logging.

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.

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.

Table 3.7. Crash root causes 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, <i>e.g.</i> , incorrect mutex usage.

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* [37] and *Clang-tidy* [31], 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 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 https://bugzilla.mozilla.org/show_bug.cgi?id=1121661#c1

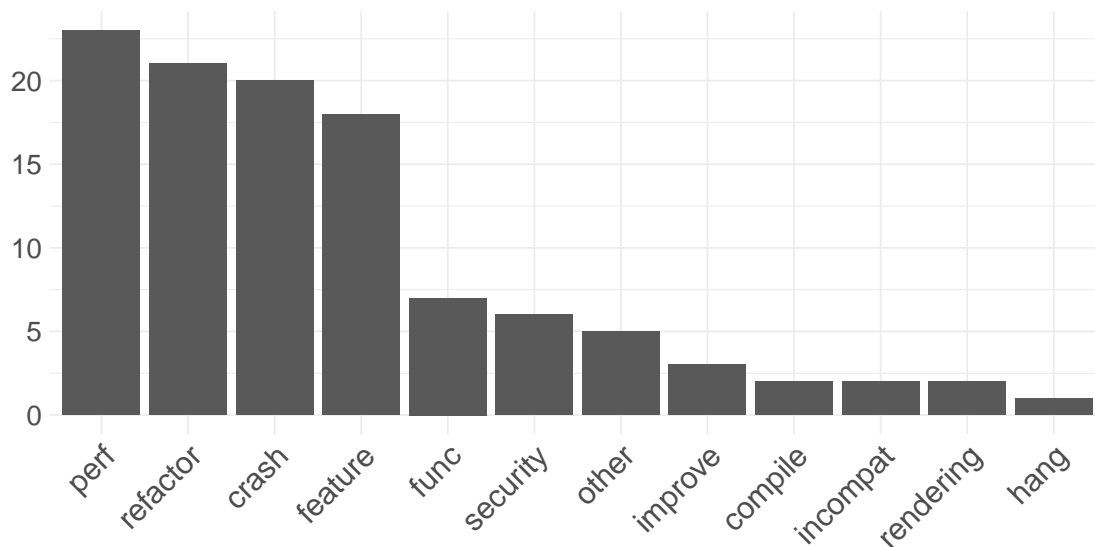


Figure 3.6. Distribution of the purposes of the reviewed issues that lead to crashes.

Reviewers should focus their effort on patches that are used to improve the performance of the software, refactor source code, fix crashes, and introduce new features, since these types of patches are more likely to lead to crashes. If possible, a super review or inspection from additional reviewers should be conducted for these patches. Memory and semantic errors are major causes of the crashes; suggesting that static analysis tools and additional scrutiny should be applied to semantic 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.

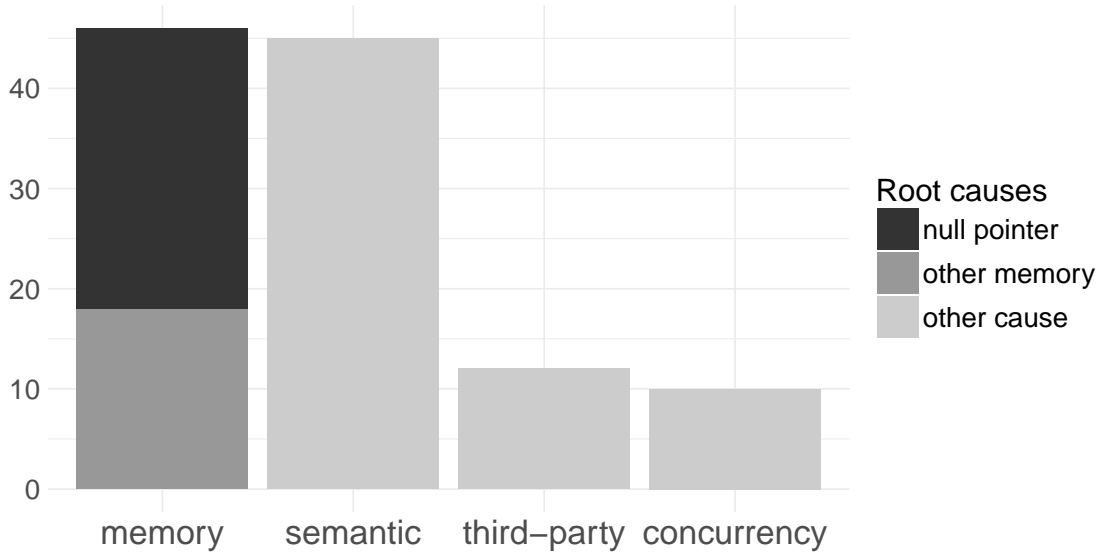


Figure 3.7. Distribution of the root causes of the reviewed issues that lead to crashes.

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 [136].

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 Related Work

In this section, we discuss the related research on crash analysis and code review analysis.

3.6.1 Crash 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 on crash-related defects. Podgurski et al. [115] 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. [70] proposed an entropy-based approach to identify crash-types that frequently occurred and affect a large number of users. Kim et al. [72] 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. [156] proposed a framework, ChangeLocator, which can automatically locate crash-inducing changes from a given bucket of crash reports.

In this work, 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.

3.6.2 Code Review & Software Quality

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. McIntosh et al. [84, 85] 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. [89] found that code review activity shares a relationship with design quality in the same studied systems. Thongtanunam et al. [142] 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. [74] 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. [124] conducted a qualitative study on 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 [84, 85, 142, 74, 124] suggest that despite being reviewed, many changes still introduce defects. Therefore, in this study, 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 [142, 67]. We also draw inspiration from Tan et al.’s work [140] to conduct a qualitative study by identifying the root causes of the reviewed patches that induce crashes and the purpose of these patches.

3.7 Conclusion

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.

An Empirical Study of Patch Uplift in Rapid Release Development Pipelines

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 [133], and can significantly delay a release process, if not enough time is allowed for stabilization. However, this well organized release process is frequently subverted by urgent patches, implementing high-value features or critical fixes, that cannot wait for the next release train. These features and fixes are directly promoted from the development channel to stable channels (*i.e.*, Aurora, Beta, and main Release), a practice called *patch uplift*. Patch uplift is risky because the time allowed for the stabilization of uplifted patches is reduced by six weeks for each skipped channel. Therefore, it is important to carefully pick the patches that are uplifted and ensure that developers scrutinize them properly,

to reduce the risk of regressions. There are a set of rules in place at Mozilla to govern this uplift process. One of these rules is that patches uplifted to the Beta channel should be (1) *ideally reproducible by the QA team, so that they can be verified*; (2) *should have been verified on Aurora/Nightly first*; and (3) *should not contain string changes (i.e., changes in the text which is visible to users)*. However, despite these rules, multiple uplifted patches still introduce regressions in the code. Hence, it is unclear if—and how the rules are enforced at Mozilla and why certain uplifted patches introduce post-release bugs.

In this chapter, 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.

The remainder of this chapter is organized as follows. 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. Section 4.5 summarizes related works.

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 [71], 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 [109] (*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

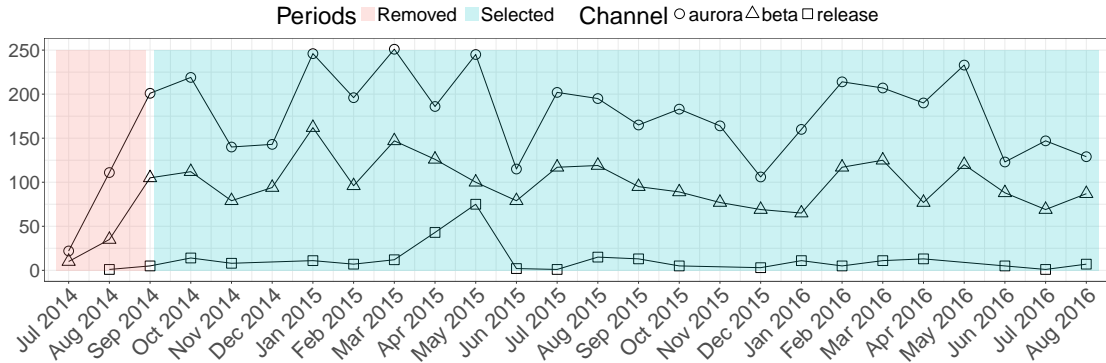


Figure 4.1. Number of uplifts during each month from July 2014 to August 2016. Periods with low number of uplifts or not covering all the three channels are removed.

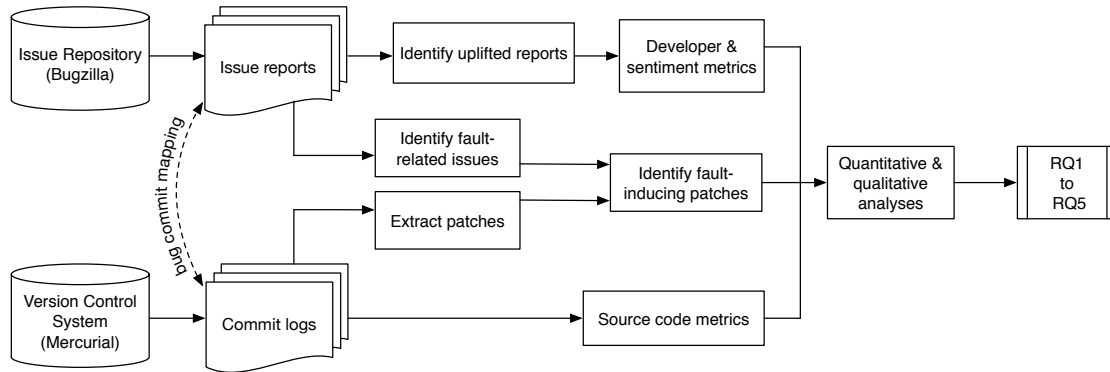


Figure 4.2. Overview of our data processing approach.

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>.

4.2.2.1 Identification of Fault-related Issues

Mozilla uses Bugzilla to manage and track its issues. All types of issues, whether they are faults or new features, are managed in this system. Unlike JIRA [147], 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%. Two researchers 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 researchers. To resolve these discrepancies, we created an online document for the 41 issues; allowing all of the researchers 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.

4.2.2.2 Identification of Fault-inducing Patches

We used the SZZ algorithm [134] 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 [47] to map each studied issue to its corresponding patch(es) (*i.e.*, commits). This heuristic consists in looking for issue IDs in commit messages using regular expressions. Next, for each fault-related issue, we used the following Mercurial command to extract the list of files that were changed to fix the issue:

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

In this step, we only considered modified and deleted lines, since added lines could not have been changed by prior commits. We denoted an issue’s fault-fixing file by F_{fix} .

Then, for each changed file $f_{fix} \mid f_{fix} \in F_{fix}$, we used Mercurial’s `annotate` command as follow to check which prior commits changed the lines that were modified by the fault-fixing commits. The SZZ algorithm assumes that the fault is located in these lines.

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

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

4.2.2.3 Identification of Duplicate Issues

There has not been an approach that can identify duplicate issues¹ 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 [123, 138, 149]. The other thread predicts whether two given issue reports are duplicate or not, such as [62, 137, 143]. 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 [137, 143], we used *Okapi BM25* algorithm [154] (referred as BM25 in the rest of the chapter) to rank of the similarity between any pair of issues: $\{(issue_i, issue_j) \mid i \neq j, issue_i \in \text{uplift bugs}, issue_j \in \text{all bugs}\}$. 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 issues², 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

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.

title (respectively description) pair by carefully analyzing the whole issue reports. There are 8.1 million pairs of duplicate issues candidates, our manual validation cannot cover all these but can only focus on the most likely candidates. First, we narrowed down our manual analysis scope to the top 1,000 candidates because correct duplicate cases can hardly be observed beyond the top 1,000 candidates (in which the highest BM25 value is 97.5, and the lowest value is 29.1) through a preliminary analysis. Second, we designed a heuristic to further filter out the pairs in which the two issue reports are not linked to each other: if Issue A is never mentioned in Issue B (either in one of the comments, or in the “Blocking”, “Depends On”, “See Also” fields), we considered the two issues to be “not linked” (meaning that, in practice, developers did not notice any relationship between the issues). To calculate the false positive rate of this heuristic, we manually examined the top 50 and 100 other randomly selected candidates, and found that only two correct duplicate pairs were misclassified by the “unlinked” heuristic. As a result, 137 candidates survived this step. Our manual validation was then performed on these candidates.

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

Compared to any fully automated approach, our approach can achieve a very high precision because all of the reported duplicate issue pairs have been carefully examined by (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 [143]. Some approaches, such as [137, 143], used additional information (*e.g.*, priority, product, and version fields from the analyzed issue reports), but such information cannot help to retrieve more possible candidate (*i.e.*, it cannot increase the recall). In this work, we only ignored the issue pairs where the titles or descriptions have no relevance (*i.e.*, BM25 value is 0) or have little relevance (*i.e.*, the two issues are not linked and the BM25 value is weak).

Table 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.

4.2.2.4 Mining Issue Reports

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

4.2.2.5 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:

1. **Developer experience and participation metrics** Our rationale for computing

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 [102]) involved by a patch.	Integer, from 0 to 76.

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.

2. **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

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>i.e.</i> , 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.

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.

- 3. Sentiments** We computed sentiment metrics because we believe that sentiments can affect uplift decisions and their success rate: for example, a release manager who is not happy about a patch might be less willing to accept it. From each studied issue, we extract developers’ comments to compute their sentiments. We leverage the sentiment mining tool, *SentiStrength* [87], 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 work [87, 144]. 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.
- 4. Code Complexity** Previous work, such as [72], has 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* [126] 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.
- 5. Code centrality (SNA) metrics** Kim et al. [72] observed that functions close to the centre of a call graph are likely to experience more faults. Hence, we

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

computed metrics capturing the centrality of functions involved in uplifted patches and uplifted patch candidates. We used the network analysis tool, *igraph* [39], in combination to *Understand* [126], as in [5], 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 [5]. 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 [106], 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 $i \in \{1, \dots, 22\}$*

Table 4.6. Accepted *vs.* rejected patch uplift candidates.

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

We used the Mann-Whitney U test [57] 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 [43] 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 [32] 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” [122]. 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, 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.

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, <i>e.g.</i> , data corruption and license incompatibility.

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 [140], 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 researchers 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.

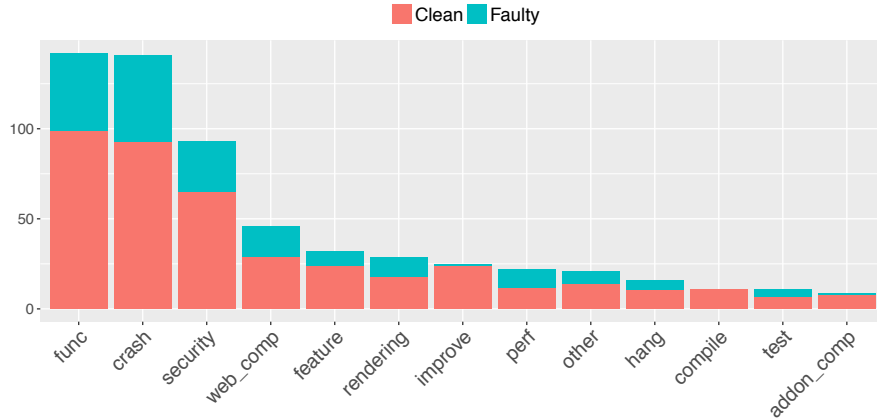


Figure 4.3. Distribution of uplift reasons in Beta.

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

The questions of both the 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

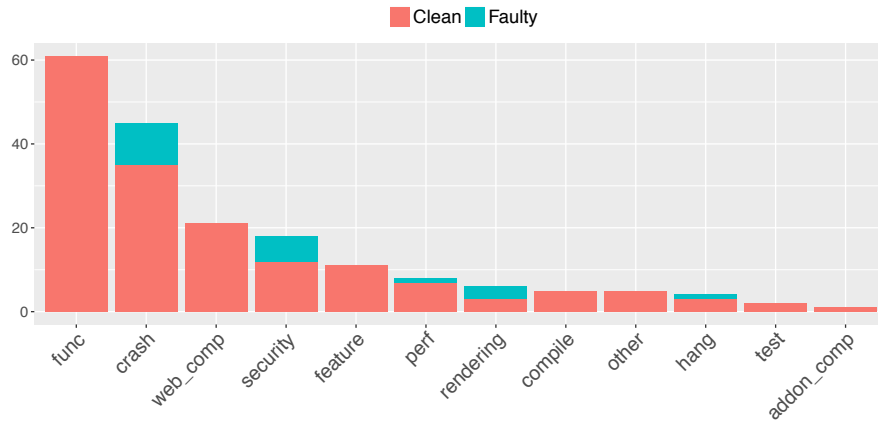


Figure 4.4. Distribution of uplift reasons in Release.

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/plugin incompatibility and compiling errors, few patches lead to faults in our studied sample. However, a high percentage of patches resolving performance and rendering problems introduced new regressions.

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

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

1. Importance of the issue. This is measured through the impact that rejecting the uplift would have on users.
2. Risk associated with the patch. Release managers share the same view on the risks. They generally trust developers' words, unless they have had bad experiences with them (*e.g.*, developers who caused regressions and did not fix them); they evaluate the risk of the patch by looking at its size and complexity, the presence/absence of

automated tests, the reviewers of the patch. In case of doubts, release managers consult other release managers or engineering managers to get a clearer picture.

3. Timing of the uplift in the Aurora/Beta cycle. They tend to trust more patches that have been in Nightly for some time and patches that are far from the next release date. They almost always accept uplifts requested during the first weeks of the Aurora cycle.
4. Verification of the patch. In particular for more stable channels, they make sure that the patch has been verified to actually fix the problems it was supposed to fix. If needed, they ask QA to manually verify the patch. If it is a patch that fixes a Nightly crash, before uplifting the patch to Aurora, they will verify if users are no longer reporting the crash.

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

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

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

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

The interviewed release managers also told us that they take into account the fault-proneness of components when making uplift decisions; which is in line with what we found (some components have a smaller acceptance rate). One release manager told us that *“some components always come out as causing the most regressions, e.g., graphics layers, DOM”*. Regarding the trust in developers, they all mentioned the assessment of risk as one of the first factors. One release manager explained that *“when they seem really overconfident or aren’t telling me the whole story I lose some trust”*, another one stated

that “some developers are taking a lot of risks, some other less and are super reactive to fix potential fallout”. This finding is consistent with the uplift criteria followed at Facebook [158], 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.

Patches accepted for uplift tend to have lower code complexity in development channels. Release manager tend to take a longer time to decide when rejecting patches than when accepting them. The top four reasons for uplift are wrong functionality, crash, security vulnerability, incompatibility with major websites. Release managers take decisions based on their past experience with developers or code components, focusing on the importance of the uplift, then the risk associated with it, then the timing of the uplift in the release cycle, then on its verification by QA.

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 [86], cloned [141], duplicate, or resolved by multiple patches [113] (which also includes backouts made by tree sheriffs, [110]) 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 4.2.2.3.

For each identified and verified issue that was not effectively fixed by an uplift, two researchers 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

Table 4.8. Root causes of the ineffective uplifts.

Category	Description
Not fixed	The issue was completely not fixed, <i>i.e.</i> , the uplifted patch did not have any effect.
Partially fixed	The issue was only partially fixed, <i>i.e.</i> , 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, <i>e.g.</i> , 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.

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. [113] and An et al. [6] respectively detected 32.8% and 23.8% general Mozilla issues (in different time periods) that were resolved by multiple patches. Shihab et al. [131] detected 6.5% to 26% reopened issues from Eclipse, Apache HTTP, and OpenOffice. Compared to these results, uplifted patches are more likely to fix a problem in one shot than other patches, even though we analyzed ineffective uplifted patches from different angles, including reopened, cloned, duplicate issues, and issues fixed by multiple uplifts. This implies that uplifted patches have a better general quality than other patches.

“The original uplifted patches did not completely fix the problem” is the most frequent root cause behind the issues that were ineffectively fixed and were later reopened, cloned, or duplicate. An example of such case is issue #1156182; the original uplifted patch of issue #1156182⁴ 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

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

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

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 #1044975⁵, 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 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 losing 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 #1154003⁶ was created due to crashes in the drawing method **DrawingCon-**

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

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

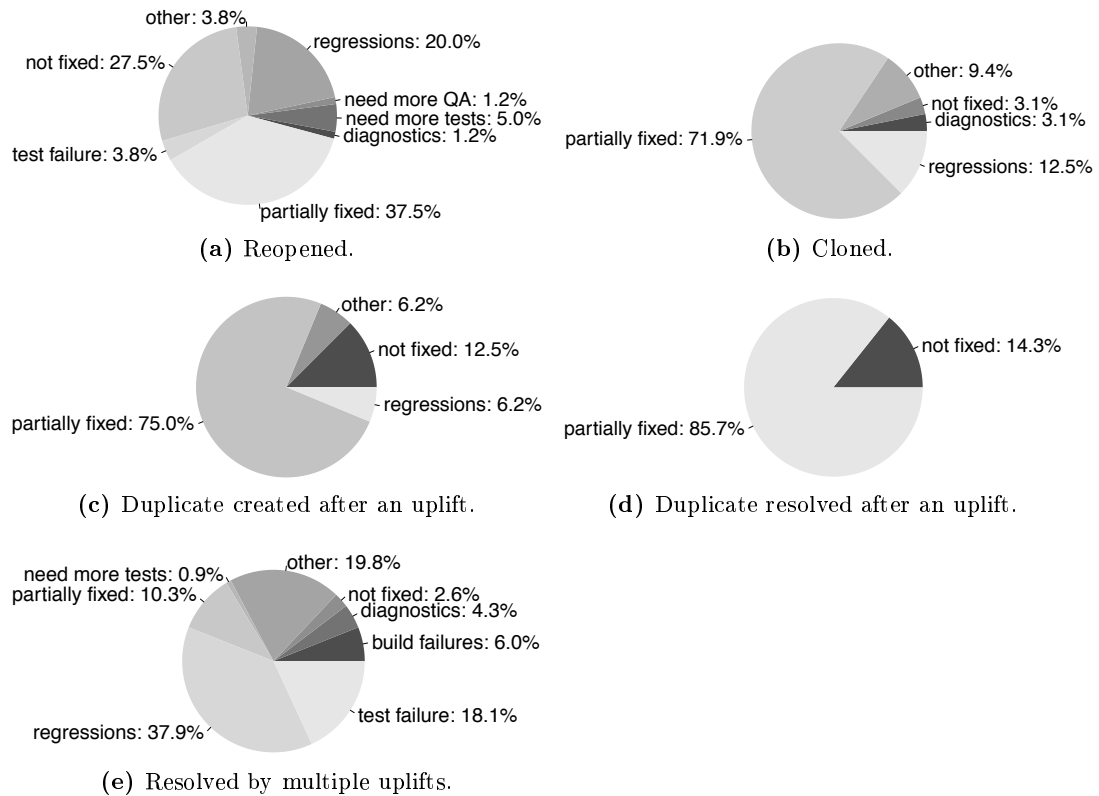


Figure 4.5. Root causes of the ineffective uplifts.

text::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 #1162520⁷. 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 #1297390⁸ was created as a follow-up to the crashes fixed in issue #1280110⁹. 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 investi-

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

gate how many issues fixed by ineffective uplifts have been “completely and eventually” resolved, we can hardly get an exact answer because first, our subject dataset is dated from September 2014 to August 2016. Answering this question is beyond the scope of our study. Second, developers and testers can hardly know whether the most recent patch has covered all possible aspects to fix a certain issue, in other words, a “fixed” problem may come back again in the future. A lesson from this finding is that some issues are more difficult to get fixed than others. If an issue has recurred in the field, a proper follow-up is required even after the issue has been closed.

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

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

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. Sometimes release managers specifically request partial fixes in order to mitigate risk.

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 4.2.2.2) 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.

Table 4.10. Fault-inducing uplifts *vs.* clean uplifts.

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

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 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 devel-

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, <i>e.g.</i> , incorrect mutex usage.
Compile	Compile-time errors.
Other	Other errors.

opers 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 [140], 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**, two researchers 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.

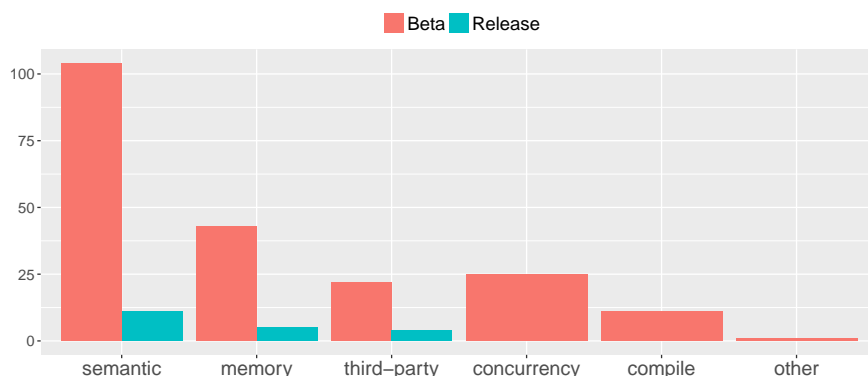


Figure 4.6. Reasons of fault-inducing uplifts.

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 [37]) 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.

Uplifted patches that introduced regressions in the code 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.

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 researchers 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 #1059797¹⁰ (which was uplifted to address a hang problem) caused a regression as issue #1239789¹¹ (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 #1075199¹² (which was uplifted to add a mock GMP plugin for testing) caused issue #1160914¹³ (which is

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

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

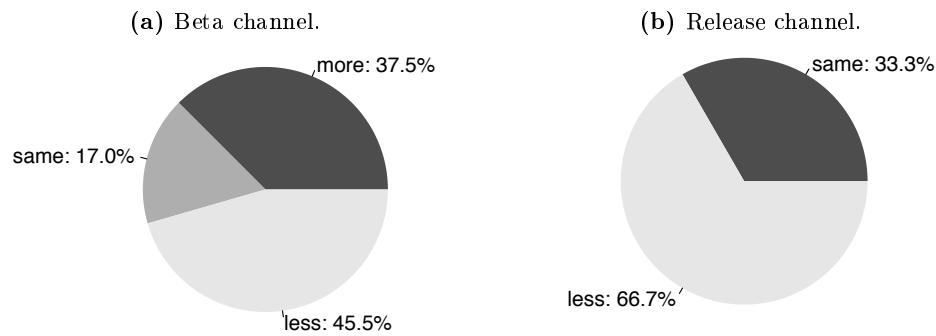
¹² https://bugzilla.mozilla.org/show_bug.cgi?id=1075199

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

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



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 degra-

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
<i>compile</i>	<i>crash</i>	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
<i>crash</i>	<i>security</i>	1	0.02
func	func	35	0.57
<i>func</i>	<i>crash</i>	14	0.23
func	perf	7	0.11
func	compile	4	0.07
func	other	1	0.02
<i>improve</i>	<i>crash</i>	7	0.37
<i>improve</i>	<i>func</i>	7	0.37
improve	compile	2	0.11
<i>improve</i>	<i>perf</i>	2	0.11
<i>improve</i>	<i>security</i>	1	0.05
<i>perf</i>	<i>func</i>	5	0.50
<i>perf</i>	<i>crash</i>	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

dation (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.

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

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

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

Release managers were, as one might have predicted, happy to see our results regarding the release channel, but were not surprised because, compared to other channels, release uplifts are inspected with more QA efforts and are more carefully approved. When using the metrics listed in Tables 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 uplifts¹⁴, we observed that the former uplifts tended to happen closer to the release date and tended to have a shorter review duration (but these results are not statistically significant as the sample we analyzed is probably small). Release managers thought that these patches might have been uplifted in a rush and under pressure, which would explain both the closeness to the release date and the short review duration.

More than one third of the fault-inducing Beta uplifts, but none of the Release uplifts, led to a regression that is more severe than the problem they aimed to address.

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

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

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 (<i>e.g.</i> , 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.

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

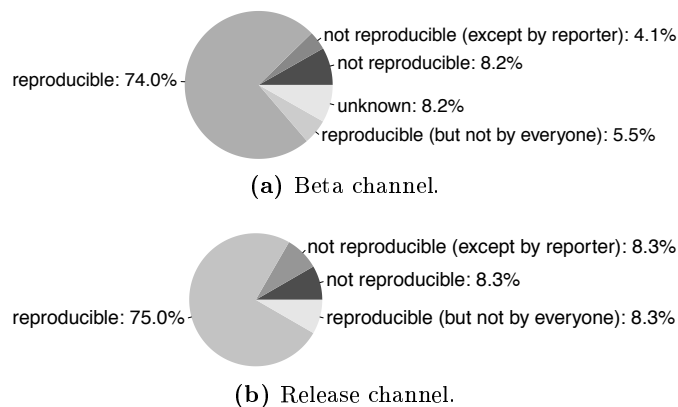


Figure 4.8. Whether the regressions caused by an uplift were reproducible.

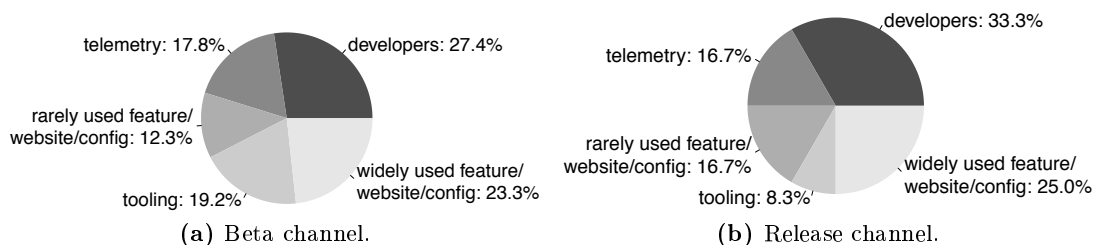


Figure 4.9. How the regressions caused by uplifts were found.

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

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

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

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

Coming back to the trade-off aspect we briefly discussed in the “Motivation” part, it applies to our suggestions too. An effective alerting system should not need to collect data for too long 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 scenario 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.

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.

4.4 Threats to Validity

In this section, we discuss the threats to validity of our study following the guidelines for case study research [157].

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

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

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 modifying 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 we performed. Specifically, in **RQ1** and **RQ3**, we applied non-parametric tests that do not require making assumptions on the distribution of our dataset. We used SentiStrength as the sentiment detection tool. We compared the performance of this tool with SentiStrengthSE [61], 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, currently the latter tool can 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. 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 [143]. 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 study, we only considered 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 Related Work

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

Release engineering encompasses all the activities aimed at “building a pipeline that transforms source code into an integrated, compiled, packaged, tested, and signed product that is ready for release” [2].

Since the adoption of the rapid release model [71] by Mozilla in 2011, a plethora

of studies have focused on the impact of rapid release strategies on software quality. Khomh et al. [71] 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 [69], that one of the major challenges of fast release cycles is the automation of the release engineering process. Da Costa et al. [40] 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. [3] analyzed the six major phases of release engineering practices and proposed a roadmap for future research, highlighting the need for more empirical studies that validate the best practices and assess the impact of release engineering processes on software quality.

Another important aspect of release engineering that has been investigated by the community is the integration of urgent patches that are used to fix severe problems, such as frequent crashes or security bugs, or to introduce important features. Urgent patches break the balance between new feature work and software quality, and hence could lead to faults and failures. Hassan et al. [56] 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. [78] 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. [117] 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. [116] observed that feature toggles [82] can be effectively turned off faulty urgent patches, which limits the impact of faulty patches.

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

4.6 Conclusion

Mozilla follows a rapid release model, which uses 18 weeks to deliver fault fixes and new features to users. Frequently, certain patches that fix critical issues, or implement high-value features are promoted directly from the development channel to a stabilization channel, because they are too urgent and cannot wait for the next release train. This practice, known as *patch uplift*, is risky because the time allowed for the stabilization of the uplifted patches is short. In average, 8% of uplifted patches introduced a regression in the code of Firefox. In this 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.

An Empirical Study of DLL Injection Bugs in the Firefox Ecosystem

5.1 Introduction

Firefox, since its inception, 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 employ 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 [96], 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, 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.

The rest of the paper is organized as follows. Section 5.2 provides background knowledge on the Firefox ecosystem as well as the risks and countermeasures of DLL injection in the system. Section 5.3 describes the design of the case study. Section 5.4 shows and analyzes the results of the case study. Section 5.5 discusses the implications of our findings. Section 5.6 discusses the threats to the validity of our study. Section 5.7 summarizes related work, and Section 5.8 draws conclusions.

5.2 Background

5.2.1 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 [99], 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 [95] [94]. 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.

5.2.2 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, EasyHook¹. 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** function² 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

¹ <https://github.com/EasyHook/EasyHook>

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

```

1 public Injection_ASM_x86@0
2 Injection_ASM_x86@0 PROC
3 ; no registers to save, because this is the thread main function
4 ; save first param (address of hook injection information)
5
6     mov esi, dword ptr [esp + 4]
7
8 ; call LoadLibraryW(Inject->EasyHookPath);
9     push dword ptr [esi + 8]
10
11     call dword ptr [esi + 40] ; LoadLibraryW@4
12     mov ebp, eax
13     test eax, eax
14     je HookInject_FAILURE_A
15
16 ; call GetProcAddress(eax, Inject->EasyHookEntry);
17     push dword ptr [esi + 24]
18     push ebp
19     call dword ptr [esi + 56] ; GetProcAddress@8
20     test eax, eax
21     je HookInject_FAILURE_B

```

Figure 5.1. An example of DLL injection performed by RoboSizer.

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**.

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 [111]. 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 [25]), and, if the problem is widespread enough, generate a new release build to ship to users.

5.3 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.

5.3.1 Data Collection

From the Mozilla bug tracking system, Bugzilla [150], 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.

5.3.2 Data Processing

We manually identified a series of characteristics from the 103 bugs obtained in Section 5.3.1. Table 5.1 shows the names and the descriptions of the characteristics. To reduce biases in the manual identification, two researchers 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

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, <i>e.g.</i> , antivirus, malware, and hardware vendor driver.
How resolved	How a bug is resolved, <i>e.g.</i> , 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 [105].

scripts to automatically extract some other characteristics as shown in the bottom of Table 5.1.

5.3.3 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 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 [96].

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?

5.4 Case Study Results

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

5.4.1 (RQ1) What are the characteristics of the bugs caused by DLL injections?

According to Mozilla telemetry³, 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.

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%

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 terminates. 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

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%

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, *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

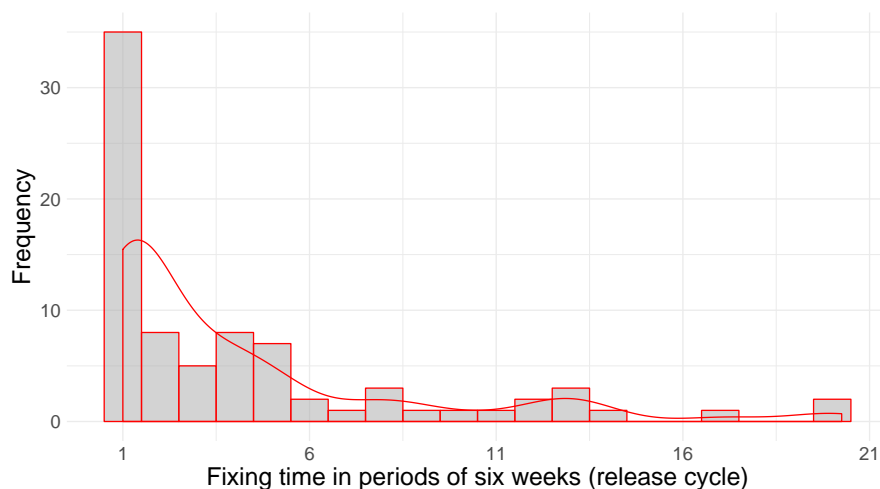


Figure 5.2. Distribution of the bug fixing time. Each bin represents a period of six weeks, *e.g.*, the first bin means bugs fixed within six weeks (*i.e.*, one release cycle).

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 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 paper. 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 [105]. 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 [57]. 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.

93 bugs (90.3%) of the DLL injection bugs led to crashes. 57 bugs (55.3%) of the bugs are from antivirus software, 19 (18.4%) of them from hardware vendor drivers, and 10 (9.7%) from malware. 1% to 15% of Firefox users also have some of the software that caused these bugs.

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)

5.4.2 (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).

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 [129] from user32.dll."* *"We used a general mechanism (SetWindowsHookEx [128]) 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 [10] registry entry."* *"CreateRemoteThread+LoadLibrary [38, 80]"*.

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.

In many cases, DLL injection bugs are triggered by injection engine errors, compiler/runtime incompatibility, or version incompatibility between the host and third-party software.

5.4.3 (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 [30] (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 paper, 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 pur-**

poses. 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 [155] 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 [30]) 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.

Completely blocking DLL injection might not be the best strategy to reduce bugs caused by third-party software. Instead, host software vendors should strengthen their collaboration and communication with third-party vendors, and build a publicly accessible validation test framework. To attract third-party vendors to use the standard extension API, host software should improve the API’s reliability and functionality (i.e., available functions). A whitelist might be beneficial, but more empirical evidences are needed to support this claim.

5.5 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' advice 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.

5.6 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 [132]).

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 researchers 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.

5.7 Related Work

5.7.1 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 [20] 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 [55] 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. [64] 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. [146] 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.*, [145]), Debian distribution (*e.g.*, [49, 51]), Eclipse (*e.g.*, [151, 21]), and R (*e.g.*, [48]) 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. [79] studied the extension security model of Chrome and Karim et al. [68] 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 [79, 68] is that DLL injection is completely arbitrary, *i.e.*, a third-party software can execute whatever it requires; while the extension API can constrain third-party software's behaviour.

5.7.2 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. [8] proposed a framework to detect code injection attacks [152]. Lam et al. [76] 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. [16] 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 [60]). 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 [153], malware producers can also take advantage of this technique to exploit computers. Jang et al. [63] 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 [45] 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 this case study on the Firefox ecosystem.

5.8 Conclusion

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 paper, 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.

Conclusion

In this thesis we showed that the development and the use of machine learning and data mining techniques can support several software engineering phases, ranging from crash handling, to code review, to patch uplifting, to software ecosystem management.

We presented the motivation behind our work, we explained the approach we followed, and we presented the challenges we faced. To validate our thesis, we conducted several studies tackling different problems in an industrial open-source context, focusing on the case of Mozilla. We have also applied some of the results presented in the dissertation in the same industrial open-source context, at Mozilla, by either providing tools to actors involved in the software development process or contributing to changes in processes.

6.1 Contributions

During the course of this dissertation, we made a series of contributions to the state of the art in machine learning and data mining techniques applied to software engineering. We summarize the major ones in the following.

6.1.1 Automatic analysis of groups of crashes for finding correlations

In Chapter 2, presenting our work from [28], we found that analyzing crash reports in an automated manner can help developers in fixing crashes, by removing manual analysis burden from developers, or by finding properties that would have been really difficult to find with manual analysis, or can give clues in the characterization of crashes.

Software organizations can use these data mining techniques to speed up and simplify the resolution of crashes and to reduce the amount of manual tedious work for developers.

6.1.2 Relation between code review and crashes

In Chapter 3, presenting our work from [7], we found that some high-impact defects, such as crash-related defects, can still pass through the review process and negatively affect end users. We compared 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. 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.

6.1.3 Patch uplift in rapid release development processes

In Chapter 4, presenting our work from [24] and [25], we found that in average, 8% of uplifted patches introduced a regression in the code of Firefox. We investigated the decision making process of patch uplift at Mozilla and observed that release managers are more inclined to accept patch uplift requests that concern certain specific components, and/or that are submitted by certain specific developers. We 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. We examined the characteristics of uplifted patches that introduced regressions 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. 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. 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.

6.1.4 DLL injection bugs in the Firefox ecosystem

In Chapter 5, presenting our work from an article that is currently under submission, 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. 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. 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.

6.2 Implications

The implications of our dissertation are important for both researchers and practitioners. Our work shows that machine learning and data mining techniques can be leveraged to improve software engineering processes, supporting several of their phases.

6.2.1 Crash handling

The part of our work on automatically analyzing groups of crashes for finding correlations, presented in Chapter 2, shows that practitioners can use data mining techniques to improve their understanding of crashes, and automate it effectively, making better use of their automated crash reporting systems. Researchers could find improvements on our proposed algorithm, or improvements for the crash clustering phase which is at the basis of our proposed techniques. Moreover, they could find additional possible applications for our correlations results (or investigate the ones suggested in Section 2.3.2), for example to try to automatically reproduce field crashes in a controlled environment.

6.2.2 Code review and crash-related defects

The part of our work on the relation between code review and crashes, presented in Chapter 3, shows that practitioners can use the results of our study to inform their review processes, increasing the scrutiny for risky patches and decreasing it for less risky ones. Researchers could find ways to automatically detect patches that would require more scrutiny, and they could propose ways to automate parts of the review process to ease prevent crash-related defects (for example, by investigating the effects of static analysis tools).

6.2.3 Uplift/urgent patches processes

The part of our work on patch uplift in rapid release development processes, presented in Chapter 4, shows that practitioners can use the results of our study to inform their uplift processes: for example, release managers can be more careful when deciding about certain kinds of patches and QA processes could be modified to prevent regressions at an earlier stage for uplifts. Researchers could propose automated ways to help release managers in their decisions, employing the data which can be collected on the uplift process.

6.2.4 Software ecosystems

The part of our work on DLL injection bugs in the Firefox ecosystem, presented in Chapter 5, shows practitioners that increasing collaboration between developers of host software and developers of software extending it might help in reducing bugs and improving the quality of both host and guest software. Researchers could perform similar studies applied to other ecosystems to corroborate our findings and expand the number of responses from third-party developers, moreover they can investigate whether other ways adopted by other software organizations to mitigate DLL injection bugs could be more or less effective.

Bibliography

- [1] A. F. Ackerman, P. J. Fowler, and R. G. Ebenau. Software inspections and the industrial production of software. In *Proceedings of a Symposium on Software Validation: inspection-testing-verification-alternatives*, pages 13–40, 1984.
- [2] 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*, 32(2):42–49, 2015.
- [3] B. Adams and S. McIntosh. Modern release engineering in a nutshell—why researchers should care. In *Proceedings of the 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER 2016)*, volume 5, pages 78–90. IEEE, 2016.
- [4] I. Ahmed, N. Mohan, and C. Jensen. The impact of automatic crash reports on bug triaging and development in Mozilla. In *Proceedings of the International Symposium on Open Collaboration*, pages 1:1–1:8, 2014.
- [5] L. An and F. Khomh. An empirical study of highly-impactful bugs in Mozilla projects. In *Proceedings of the International Conference on Software Quality, Reliability and Security (QRS 2015)*, pages 262–271. IEEE Computer Society, 2015.
- [6] L. An, F. Khomh, and B. Adams. Supplementary bug fixes vs. re-opened bugs. In *Proceedings of the 14th International Working Conference on Source Code Analysis and Manipulation (SCAM 2014)*, pages 205–214. IEEE Computer Society, 2014.
- [7] L. An, F. Khomh, S. McIntosh, and M. Castelluccio. Why did this reviewed code crash? an empirical study of Mozilla Firefox. In *Proceedings of the 2018 25th Asia-Pacific Software Engineering Conference (APSEC 2018)*, 2018.

- [8] 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 *Proceedings of the 21st Annual Computer Security Applications Conference (ACSAC 2005)*, pages 49–58. IEEE Computer Society, 2005.
- [9] J. Anvik, L. Hiew, and G. C. Murphy. Who should fix this bug? In *Proceedings of the 28th international conference on Software engineering*, ICSE '06, pages 361–370. ACM, 2006.
- [10] AppInit_DLLs in Windows 7 and Windows Server 2008 R2. [https://msdn.microsoft.com/en-us/library/windows/desktop/dd744762\(v=vs.85\).aspx](https://msdn.microsoft.com/en-us/library/windows/desktop/dd744762(v=vs.85).aspx), 2018. Online; Accessed April 12th, 2018.
- [11] D. L. Atkins, T. Ball, T. L. Graves, and A. Mockus. Using version control data to evaluate the impact of software tools: A case study of the version editor. *IEEE Transactions on Software Engineering*, 28(7):625–637, July 2002.
- [12] T. Ball, J.-M. K. Porter, and H. P. Siy. If your version control system could talk ... In *Proceedings of the International Conference on Software Engineering - Workshop on Process Modeling and Empirical Studies of Software Engineering*, 1997.
- [13] K. Bartz, J. Stokes, J. Platt, R. Kivett, D. Grant, S. Calinoiu, and G. Loihle. Finding similar failures using callstack similarity. In *Proceedings of the 3rd Conference on Tackling Computer Systems Problems with Machine Learning Techniques (SysML 2008)*, pages 1–1, 2008.
- [14] S. Bay and M. Pazzani. Detecting change in categorical data: mining contrast sets. In *Proceedings of the 5th International Conference on Knowledge Discovery and Data Mining (KDD 1999)*, pages 302–306. ACM SIGKDD, 1999.
- [15] S. Bay and M. Pazzani. Detecting group differences: mining contrast sets. *Data Mining and Knowledge Discovery*, 5(3):213–246, July 2001.
- [16] J. Berdajs and Z. Bosnić. Extending applications using an advanced approach to DLL injection and API hooking. *Software: Practice and Experience*, 40(7):567–584, 2010.
- [17] N. Biggs. *Algebraic graph theory*. Cambridge university press, 1993.

- [18] C. Bird, N. Nagappan, P. Devanbu, H. Gall, and B. Murphy. Does distributed development affect software quality? an empirical case study of Windows Vista. *Communications of the ACM*, 52(8):85–93, 2009.
- [19] C. Bird, V. Ranganath, T. Zimmermann, N. Nagappan, and A. Zeller. Extrinsic influence factors in software reliability: a study of 200000 windows machines. In *Proceedings of the 36th International Conference on Software Engineering - Companion Volume (ICSE-Companion 2014)*, pages 205–214, 2014.
- [20] J. Bosch. From software product lines to software ecosystems. In *Proceedings of the 13th International Software Product Line Conference (SPLC 2009)*, pages 111–119. Carnegie Mellon University, 2009.
- [21] 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 2010) and International Workshop on Principles of Software Evolution (IWPSE 2010)*, pages 63–72. ACM, 2010.
- [22] J. Campbell, E. Santos, and A. Hindle. The unreasonable effectiveness of traditional information retrieval in crash report deduplication. In *Proceedings of the 13th International Conference on Mining Software Repositories (MSR 2016)*, pages 269–280, 2016.
- [23] M. Castelluccio. Mozilla software engineering research ideas and documentation, Dec. 2018.
- [24] M. Castelluccio, L. An, and F. Khomh. Is it safe to uplift this patch? an empirical study on Mozilla Firefox. In *Proceedings of the 33rd International Conference on Software Maintenance and Evolution (ICSME 2017)*, pages 411–421. IEEE, 2017.
- [25] M. Castelluccio, L. An, and F. Khomh. An empirical study of patch uplift in rapid release development pipelines. *Empirical Software Engineering*, 2018.
- [26] M. Castelluccio, G. Poggi, C. Sansone, and L. Verdoliva. Land use classification in remote sensing images by convolutional neural networks. *CoRR*, abs/1508.00092, 2015.
- [27] M. Castelluccio, G. Poggi, C. Sansone, and L. Verdoliva. Training convolutional neural networks for semantic classification of remote sensing imagery. In *Proceedings of the Joint Urban Remote Sensing Event (JURSE 2017)*, pages 1–4, 2017.

- [28] M. Castelluccio, C. Sansone, L. Verdoliva, and G. Poggi. Automatically analyzing groups of crashes for finding correlations. In *Proceedings of the 11th Joint Meeting of the European Software Engineering Conference and the Symposium on the Foundations of Software Engineering (ESEC/FSE 2017)*, pages 717–726. ACM, 2017.
- [29] B. Chan, Y. Zou, A. Hassan, and A. Sinha. Visualizing the results of field testing. In *Proceedings of the 18th International Conference on Program Comprehension (ICPC 2010)*, pages 114–123. IEEE, 2010.
- [30] 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.
- [31] Clang-Tidy tool. <http://clang.llvm.org/extra/clang-tidy>, 2017. Online; Accessed March 31st, 2017.
- [32] N. Cliff. Dominance statistics: Ordinal analyses to answer ordinal questions. *Psychological Bulletin*, 114(3):494, 1993.
- [33] N. Cliff. *Ordinal methods for behavioral data analysis*. Psychology Press, 2014.
- [34] CLOC. <http://cloc.sourceforge.net>, 2017. Online; Accessed May 22nd, 2017.
- [35] Firefox code review. https://wiki.mozilla.org/Firefox/Code_Review, 2016. Online; Accessed March 31st, 2016.
- [36] R. Coe. It’s the effect size, stupid: what effect size is and why it is important. In *Proceedings of the Annual Conference of the British Educational Research Association*. Education-line, 2002.
- [37] Coverity tool. <http://www.coverity.com>, 2017. Online; Accessed March 31st, 2017.
- [38] CreateRemoteThread function. [https://msdn.microsoft.com/en-us/library/windows/desktop/ms682437\(v=vs.85\).aspx](https://msdn.microsoft.com/en-us/library/windows/desktop/ms682437(v=vs.85).aspx), 2018. Online; Accessed April 12th, 2018.
- [39] G. Csardi and T. Nepusz. The igraph software package for complex network research. *InterJournal, Complex Systems*, 1695(5):1–9, 2006.

- [40] 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)*, pages 374–385, 2016.
- [41] 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 2012)*, pages 1084–1093. IEEE Press, 2012.
- [42] T. Dhaliwal, F. Khomh, and Y. Zou. Classifying field crash reports for fixing bugs: a case study of Mozilla Firefox. In *Proceedings of the 27th International Conference on Software Maintenance (ICSM 2011)*, pages 333–342. IEEE, 2011.
- [43] A. Dmitrienko, G. Molenberghs, C. Chuang-Stein, and W. Offen. *Analysis of Clinical Trials Using SAS: a Practical Guide*. SAS Institute, 2005.
- [44] B. Everitt. *The Analysis of Contingency Tables*. Chapman & Hall/CRC, February 1992.
- [45] S. Fewer. Reflective DLL injection. *Harmony Security, Version, 1*, 2008.
- [46] Announcement of the crash correlations tool in a Mozilla mailing list. <https://mail.mozilla.org/pipermail/firefox-dev/2016-November/004804.html>, 2016. Online; Accessed December 10th, 2018.
- [47] 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 2003)*, pages 23–32. IEEE, 2003.
- [48] D. M. German, B. Adams, and A. E. Hassan. The evolution of the R software ecosystem. In *Proceedings of the 17th European Conference on Software Maintenance and Reengineering (CSMR 2013)*, pages 243–252. IEEE, 2013.
- [49] D. M. German, J. M. Gonzalez-Barahona, and G. Robles. A model to understand the building and running inter-dependencies of software. In *Proceedings of the 14th Working Conference on Reverse Engineering (WCRE 2007)*, pages 140–149. IEEE, 2007.

- [50] K. Glerum, K. Kinshumann, S. Greenberg, G. Aul, V. Orgovan, G. Nichols, D. Grant, G. Loihle, and G. Hunt. Debugging in the (very) large: ten years of implementation and experience. In *Proceedings of the 22nd Symposium on Operating Systems Principles (SOSP 2009)*, pages 103–116. ACM SIGOPS, 2009.
- [51] 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*, 14(3):262–285, 2009.
- [52] T. L. Graves and A. Mockus. Inferring change effort from configuration management databases. In *Proceedings of the 5th International Symposium on Software Metrics (METRICS 1998)*, pages 267–. IEEE Computer Society, 1998.
- [53] T. Hall, S. Beecham, D. Bowes, D. Gray, and S. Counsell. A systematic literature review on fault prediction performance in software engineering. *IEEE Trans. Softw. Eng.*, 38(6):1276–1304, Nov. 2012.
- [54] R. A. Hanneman and M. Riddle. *Introduction to social network methods*, 2005.
- [55] G. K. Hanssen. A longitudinal case study of an emerging software ecosystem: implications for practice and theory. *Journal of Systems and Software*, 85(7):1455–1466, 2012.
- [56] S. Hassan, W. Shang, and A. E. Hassan. An empirical study of emergency updates for top android mobile apps. *Empirical Software Engineering*, pages 1–42, 2016.
- [57] M. Hollander, D. A. Wolfe, and E. Chicken. *Nonparametric statistical methods*. John Wiley & Sons, 3rd edition, 2013.
- [58] 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.
- [59] 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 2005)*, pages 495–504. IEEE, 2005.
- [60] API hooking. <http://resources.infosecinstitute.com/api-hooking>, 2014. Online; Accessed April 12th, 2018.

- [61] M. R. Islam and M. F. Zibran. Leveraging automated sentiment analysis in software engineering. In *Proceedings of the 14th International Conference on Mining Software Repositories (MSR 2017)*, pages 203–214. IEEE/ACM, 2017.
- [62] N. Jalbert and W. Weimer. Automated duplicate detection for bug tracking systems. In *Proceedings of the 38th International Conference on Dependable Systems and Networks With FTCS and DCC (DSN 2008)*, pages 52–61. IEEE, 2008.
- [63] M. Jang, H. Kim, and Y. Yun. Detection of DLL inserted by Windows malicious code. In *Proceedings of the International Conference on Convergence Information Technology (ICCIT 2007)*, ICCIT, pages 1059–1064. IEEE, 2007.
- [64] S. Jansen, A. Finkelstein, and S. Brinkkemper. A sense of community: a research agenda for software ecosystems. In *Proceedings of the 31st International Conference on Software Engineering - Companion Volume (ICSE-Companion 2009)*, pages 187–190. IEEE, 2009.
- [65] Seta - search for extraneous test automation. <https://elvis314.wordpress.com/2015/02/06/seta-search-for-extraneous-test-automation/>, 2018. Online; Accessed December 3rd, 2018.
- [66] K. S. Jones. A statistical interpretation of term specificity and its application in retrieval. *Journal of Documentation*, 28(1):11–21, 1972.
- [67] 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*, 39(6):757–773, 2013.
- [68] R. Karim, M. Dhawan, V. Ganapathy, and C.-c. Shan. An analysis of the Mozilla Jetpack extension framework. In *Proceedings of the 26th European Conference on Object-Oriented Programming (ECOOP 2012)*, pages 333–355. Springer, 2012.
- [69] F. Khomh, B. Adams, T. Dhaliwal, and Y. Zou. Understanding the impact of rapid releases on software quality. *Empirical Software Engineering*, pages 1–38, 2014.
- [70] 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 *Proceedings of the 18th Working Conference on Reverse Engineering (WCRE 2011)*, pages 261–270. IEEE, 2011.

- [71] 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 Working Conference on Mining Software Repositories (MSR 2012)*, MSR '12, pages 179–188. IEEE Press, 2012.
- [72] 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*, 37(3):430–447, 2011.
- [73] S. Kim, T. Zimmermann, and N. Nagappan. Crash graphs: An aggregated view of multiple crashes to improve crash triage. In *Proceedings of the 41st International Conference on Dependable Systems and Networks (DSN 2011)*, pages 486–493. IEEE/IFIP, IEEE Computer Society, 2011.
- [74] O. Kononenko, O. Baysal, L. Guerrouj, Y. Cao, and M. W. Godfrey. Investigating code review quality: do people and participation matter? In *Proceedings of the 31st International Conference on Software Maintenance and Evolution (ICSME 2015)*, pages 111–120. IEEE, 2015.
- [75] P. Kralj, N. Lavrač, D. Gamberger, and A. Krstačić. Contrast set mining for distinguishing between similar diseases. In *Proceedings of the 11th Conference on Artificial Intelligence in Medicine (AIME 2007)*, pages 109–118, 2007.
- [76] L.-c. Lam, Y. Yu, and T.-c. Chiueh. Secure mobile code execution service. In *Proceedings of the 20th conference on Large Installation System Administration (LISA 2006)*, pages 53–62, 2006.
- [77] J. Lerch and M. Mezini. Finding duplicates of your yet unwritten bug report. In *Proceedings of European Conference on Software Maintenance and Reengineering (CSMR 2013)*, pages 69–78. IEEE, 2013.
- [78] D. Lin, C.-P. Bezemer, and A. E. Hassan. Studying the urgent updates of popular games on the steam platform. *Empirical Software Engineering*, pages 1–32, 2016.
- [79] L. Liu, X. Zhang, G. Yan, S. Chen, et al. Chrome extensions: threat analysis and countermeasures. In *Proceedings of the 19th Network and Distributed System Security Symposium (NDSS 2012)*, 2012.
- [80] LoadLibrary function. [https://msdn.microsoft.com/en-us/library/windows/desktop/ms684175\(v=vs.85\).aspx](https://msdn.microsoft.com/en-us/library/windows/desktop/ms684175(v=vs.85).aspx), 2018. Online; Accessed April 12th, 2018.

- [81] G. Lohman, J. Champlin, and P. Sohn. Quickly finding known software problems via automated symptom matching. In *Proceedings of the 2nd International Conference on Automatic Computing (ICAC 2005)*, pages 101–110. IEEE Computer Society, 2005.
- [82] Feature toggle. <https://martinfowler.com/bliki/FeatureToggle.html>, 2017. Online; Accessed March 22nd, 2017.
- [83] T. J. McCabe. A complexity measure. *Software Engineering, IEEE Transactions on*, (4):308–320, 1976.
- [84] 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 (MSR 2014)*. ACM, 2014.
- [85] S. McIntosh, Y. Kamei, B. Adams, and A. E. Hassan. An empirical study of the impact of modern code review practices on software quality. *Empirical Software Engineering*, page To appear, 2015.
- [86] A Bug’s Life. https://developer.mozilla.org/en-US/docs/Mozilla/QA/A_Bugs_Life, 2017. Online; Accessed May 20th, 2018.
- [87] T. Mike, B. Kevan, P. Georgios, C. Di, and K. Arvid. Sentiment in short strength detection informal text. *JASIST*, 61(12):2544–2558, 2010.
- [88] N. Modani, R. Gupta, G. Lohman, T. Syeda-Mahmood, and L. Mignet. Automatically identifying known software problems. In *Proceedings of the 23rd International Conference on Data Engineering Workshops (ICDE 2007)*, pages 433–441, 2007.
- [89] 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 *Proceedings of the 22nd International Conference on Software Analysis, Evolution and Reengineering (SANER 2015)*, pages 171–180. IEEE, 2015.
- [90] Mozilla. Socorro server - <https://github.com/mozilla/socorro>.
- [91] Mozilla. BugHunter. <https://wiki.mozilla.org/Auto-tools/Projects/BugHunter>, 2017. Online; Accessed February 21st, 2017.

- [92] Mozilla. Socorro crash report schema. https://github.com/mozilla/socorro/blob/master/socorro/schemas/crash_report.json, 2017. Online; Accessed February 21st, 2017.
- [93] Mozilla. Socorro crash report schema. <https://developer.mozilla.org/en-US/docs/Mozilla/Tech/XPCOM>, 2017. Online; Accessed February 21st, 2017.
- [94] Advantages of WebExtensions for developers. <https://blog.mozilla.org/addons/2016/03/14/webextensions-whats-in-it-for-developers/>, 2018. Online; Accessed April 16th, 2018.
- [95] 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.
- [96] 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.
- [97] Dawn project or the end of Aurora. <https://release.mozilla.org/firefox/release/2017/04/17/Dawn-Project-FAQ.html>, 2018. Online; Accessed December 3rd, 2018.
- [98] Priority field. https://wiki.mozilla.org/Bugmasters/Projects/Folk_Knowledge/Priority_Field, 2016. Online; Accessed May 20th, 2018.
- [99] WebExtensions API. <https://wiki.mozilla.org/WebExtensions>, 2017. Online; Accessed April 12th, 2018.
- [100] Mozilla bug triaging guidelines. <https://github.com/mozilla/bug-handling/blob/ed3bd3778e233715e4c79b85186ba86c2ce9352b/policy/triage-bugzilla.md>, 2018. Online; Accessed November 30th, 2018.
- [101] Mozilla developer guide. https://developer.mozilla.org/en-US/docs/Mozilla/Developer_guide, 2018. Online; Accessed November 30th, 2018.
- [102] Mozilla modules. <https://wiki.mozilla.org/Modules>, 2018. Online; Accessed September 22nd, 2018.
- [103] Mozilla Phabricator. <https://wiki.mozilla.org/Phabricator>, 2018. Online; Accessed November 26th, 2018.

- [104] Mozilla release management. https://wiki.mozilla.org/Release_Management, 2018. Online; Accessed November 30th, 2018.
- [105] Mozilla release management tracking rules. https://wiki.mozilla.org/Release_Management/Release_Process, 2018. Online; Accessed March 28th, 2018.
- [106] Mozilla release management uplift rules. https://wiki.mozilla.org/Release_Management/Uplift_rules, 2018. Online; Accessed May 20th, 2018.
- [107] Mozilla Splinter. <https://bugzilla.mozilla.org/page.cgi?id=splinter/help.html>, 2018. Online; Accessed November 26th, 2018.
- [108] Mozilla tree rules. https://wiki.mozilla.org/Tree_Rules, 2018. Online; Accessed October 23th, 2018.
- [109] Mozilla tree sheriffs. <https://wiki.mozilla.org/Sheriffing>, 2018. Online; Accessed May 20th, 2018.
- [110] Mozilla tree sheriffs - backouts. https://wiki.mozilla.org/Sheriffing/How_To/Backouts, 2018. Online; Accessed September 22nd, 2018.
- [111] Mozilla’s blocklisting policy. <https://wiki.mozilla.org/Blocklisting>, 2018. Online; Accessed April 16th, 2018.
- [112] P. Novak, N. Lavrač, and G. Webb. Supervised descriptive rule discovery: a unifying survey of contrast set, emerging pattern and subgroup mining. *Journal of Machine Learning Research*, 10:377–403, 2009.
- [113] J. Park, M. Kim, B. Ray, and D.-H. Bae. An empirical study of supplementary bug fixes. In *Proceedings of the 9th Working Conference on Mining Software Repositories (MSR 2012)*, pages 40–49. IEEE Press, 2012.
- [114] M. Pinzger and H. C. Gall. Dynamic analysis of communication and collaboration in OSS projects. In *Collaborative Software Engineering*, pages 265–284. Springer, 2010.
- [115] A. Podgurski, D. Leon, P. Francis, W. Masri, M. Minch, J. Sun, and B. Wang. Automated support for classifying software failure reports. In *Proceedings of the 25th International Conference on Software Engineering (ICSE 2003)*, pages 465–475. IEEE, 2003.

- [116] 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 (MSR 2016)*, pages 201–211. ACM, 2016.
- [117] M. T. Rahman and P. C. Rigby. Release stabilization on Linux and chrome. *IEEE Software*, 32(2):81–88, 2015.
- [118] A. Ram, A. A. Sawant, M. Castelluccio, and A. Bacchelli. What makes a code change easier to review: an empirical investigation on code change reviewability. In *Proceedings of the 26th ACM Joint Meeting of the European Software Engineering Conference and the Symposium on the Foundations of Software Engineering (ESEC/FSE 2018)*, pages 201–212. ACM, 2018.
- [119] 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.
- [120] Mozilla reviewer checklist. https://developer.mozilla.org/en-US/docs/Mozilla/Developer_guide/Reviewer_Checklist, 2017. Online; Accessed May 31st, 2017.
- [121] 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 2008)*, pages 541–550. ACM, 2008.
- [122] 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’s d for evaluating group differences on the NSSE and other surveys? In *Annual Meeting of the Florida Association of Institutional Research*, pages 1–33, 2006.
- [123] 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 (ICSE 2007)*, pages 499–510. IEEE Computer Society, 2007.
- [124] 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 (ICSE: SEIP 2018)*, pages 181–190. ACM, 2018.

- [125] D. Sankoff and J. Kruskal. *Time Warps, String Edits, and Macromolecules: The Theory and Practice of Sequence Comparison*. 1983.
- [126] Understand tool. <https://scitools.com>, 2016. Online; Accessed March 31st, 2016.
- [127] J. Scott. *Social network analysis*. Sage, 2012.
- [128] SetWindowsHookEx function. [https://msdn.microsoft.com/en-us/library/windows/desktop/ms644990\(v=vs.85\).aspx](https://msdn.microsoft.com/en-us/library/windows/desktop/ms644990(v=vs.85).aspx), 2018. Online; Accessed April 12th, 2018.
- [129] SetWinEventHook function. [https://msdn.microsoft.com/en-us/library/windows/desktop/dd373640\(v=vs.85\).aspx](https://msdn.microsoft.com/en-us/library/windows/desktop/dd373640(v=vs.85).aspx), 2018. Online; Accessed April 12th, 2018.
- [130] J. Shaffre. Multiple hypothesis testing. *Annual Review of Psychology*, 46(1):561–584, 1995.
- [131] 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*, pages 1–38, 2012.
- [132] J. Singer, S. E. Sim, and T. C. Lethbridge. Software engineering data collection for field studies. In *Guide to Advanced Empirical Software Engineering*, pages 9–34. Springer, 2008.
- [133] A. Laforge. chrome release cycle. <http://www.slideshare.net/Jolicloud/chrome-release-cycle>, 2016. Online; Accessed 06 February 2016.
- [134] J. Śliwerski, T. Zimmermann, and A. Zeller. When do changes induce fixes? 30:1–5, 2005.
- [135] Socorro: Mozilla’s crash reporting system. <https://blog.mozilla.org/webdev/2010/05/19/socorro-mozilla-crash-reports/>, 2016. Online; Accessed March 31st, 2016.
- [136] 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.

- [137] C. Sun, D. Lo, S.-C. Khoo, and J. Jiang. Towards more accurate retrieval of duplicate bug reports. In *Proceedings of the 26th International Conference on Automated Software Engineering (ASE 2011)*, pages 253–262. IEEE/ACM, 2011.
- [138] 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 International Conference on Software Engineering (ICSE 2010)*, volume 1, pages 45–54. ACM/IEEE, 2010.
- [139] Super-review policy. <https://www.mozilla.org/en-US/about/governance/policies/reviewers/>, 2016. Online; Accessed March 31st, 2016.
- [140] L. Tan, C. Liu, Z. Li, X. Wang, Y. Zhou, and C. Zhai. Bug characteristics in open source software. *Empirical Software Engineering*, 19(6):1665–1705, 2014.
- [141] The Bugzilla guide. <https://www.bugzilla.org/docs/2.20/html/bugreports.html>, 2017. Online; Accessed May 20th, 2018.
- [142] 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 *Proceedings of the 12th Working Conference on Mining Software Repositories (MSR 2015)*, pages 168–179, 2015.
- [143] Y. Tian, C. Sun, and D. Lo. Improved duplicate bug report identification. In *Proceedings of the 16th European Conference on Software Maintenance and Reengineering (CSMR 2012)*, pages 385–390. IEEE, 2012.
- [144] P. Tourani and B. Adams. The impact of human discussions on just-in-time quality assurance: an empirical study on OpenStack and Eclipse. In *Proceedings of the 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER 2016)*, volume 1, pages 189–200. IEEE, 2016.
- [145] Q. Tu et al. Evolution in open source software: a case study. In *Proceedings of the International Conference on Software Maintenance (ICSM 2000)*, pages 131–142. IEEE, 2000.
- [146] 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 (ECSA-Companion 2010)*, pages 127–134. ACM, 2010.

- [147] Jira. <https://jira.atlassian.com/>, 2017. Accessed March 30th, 2017.
- [148] S. Wang, F. Khomh, and Y. Zou. Improving bug management using correlations in crash reports. *Empirical Software Engineering*, pages 1–31, 2014.
- [149] 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 *Proceedings of the 30th International Conference on Software Engineering (ICSE 2008)*, pages 461–470. ACM/IEEE, 2008.
- [150] Bugzilla@Mozilla. <https://bugzilla.mozilla.org>, 2017. Online; Accessed April 12th, 2018.
- [151] M. Wermelinger and Y. Yu. Analyzing the evolution of Eclipse plugins. In *Proceedings of the 5th International Working Conference on Mining Software Repositories (MSR 2008)*, pages 133–136. ACM, 2008.
- [152] Code injection. https://en.wikipedia.org/wiki/Code_injection, 2018. Online; Accessed April 12th, 2018.
- [153] DLL injection. https://en.wikipedia.org/wiki/DLL_injection, 2018. Online; Accessed April 12th, 2018.
- [154] Okapi BM25. https://en.wikipedia.org/wiki/Okapi_BM25, 2018. Online; Accessed May 20th, 2018.
- [155] Windows data types. [https://msdn.microsoft.com/en-us/library/windows/desktop/aa383751\(v=vs.85\).aspx](https://msdn.microsoft.com/en-us/library/windows/desktop/aa383751(v=vs.85).aspx), 2018. Online; Accessed April 12th, 2018.
- [156] R. Wu, M. Wen, S.-C. Cheung, and H. Zhang. Changelocator: locate crash-inducing changes based on crash reports. *Empirical Software Engineering*, pages 1–35, 2017.
- [157] R. K. Yin. *Case Study Research: Design and Methods - Third Edition*. SAGE Publications, 3rd edition, 2002.
- [158] Keynote of the 2014 release engineering conference. <https://www.youtube.com/watch?v=Nffzkkdq7GM>, 2014. Online; Accessed March 30th, 2017.
- [159] X. Yu, S. Han, D. Zhang, and T. Xie. Comprehending performance from real-world execution traces: a device-driver case. In *Proceedings of the 19th International*

Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2014), pages 193–206. ACM, 2014.

- [160] M. Zaharia, M. Chowdhury, M. Franklin, S. Shenker, and I. Stoica. Spark: cluster computing with working sets. In *Proceedings of the 2nd Conference on Hot Topics in Cloud Computing (HotCloud 2010)*, pages 10–10. USENIX, 2010.