Coordination Practices for Software Quality Assurance Activities in Open-source Software Ecosystems

by

Jiahuei Lin

A thesis submitted to the Graduate Program in Computing

in conformity with the requirements for the

Degree of Doctor of Philosophy

Queen's University

Kingston, Ontario, Canada

June 2023

Copyright © Jiahuei Lin, 2023

Abstract

OPEN source software ecosystems continue to gain popularity and significant importance. A software ecosystem consists of tens of thousands of software projects with complex relations among them. Users can install these projects in any combination. Due to the complex relations among the projects and the diversity of combined installation of such projects, coordination between developers is necessary to ensure the quality of both their own projects and the entire ecosystem. For coordination for software quality assurance activities, a software ecosystem usually does not have guidelines for developers to ensure the quality of each software project. In this thesis, we leverage data from large-scale software ecosystems, i.e., Linux and WordPress, in an effort to gain a better understanding of the current coordination practices for software quality assurance. In particular, we examine four areas of coordination activities for software quality assurance in software ecosystems: upstream bug coordination, vulnerability coordination, vulnerability

fixing and disclosure coordination, and release coordination of co-evolving software projects, all within an ecosystem. In this thesis, we discuss the motivation and approach to study these four areas of coordination activities and perform empirical studies on the software ecosystems. Our results suggest the need of automated tools to track upstream bug coordination to facilitate in-depth investigation. Developers across software ecosystems coordinate to develop a vulnerability fix but work parallelly afterward. Co-evolving software projects need coordination mechanisms as they interfere with each other due to shared resources.

Acknowledgments

Thanks a million to Prof. Ahmed E. Hassan for his continuous guidance and insightful advice throughout these four years. All this work would not have been possible without his vision and support. It is my pleasure to work with Prof. Ahmed Hassan, who encouraged me to pursue the research topic of mining software repositories and taught me the art of research in empirical software engineering.

Very special thanks to Prof. Bram Adam, Prof. Mohammed Sayagh, and Dr. Haoxiang Zhang for the great collaborations over the past years. Thanks a lot for your fruitful discussions and inspiring comments on my work over countless times. I am looking forward to our next collaborations.

Thanks to all members of the Software Analysis and Intelligence Lab (SAIL) at Queen's University, including all the postdoc and students with whom I worked. It was a great time in Kingston to work with everyone at SAIL.

Dedication

This thesis is dedicated to my parents for their love, kindness, and support.

Co-authorship

FOR the chapters and related publications in the thesis, my contributions include proposing the initial research idea, investigating background knowledge and related work, proposing research methods, conducting experiments and collecting the data, analyzing the data and verifying the hypotheses, and writing and polishing the manuscript. My co-contributors supported me in refining the initial ideas, providing suggestions for potential research methods, providing feedback on experimental results and earlier manuscript drafts, and providing advice for polishing the writing.

The work presented in the thesis is published as listed below:

- Jiahuei Lin, Haoxiang Zhang, Bram Adams, and Ahmed E. Hassan. 2022. Upstream Bug Management in Linux Distributions. Empirical Software Engineering (EMSE). In press. This work is described in Chapter 3.

- Jiahuei Lin, Haoxiang Zhang, Bram Adams, and Ahmed E. Hassan. 2023. Vulnerability Management in Linux Distributions. Empirical Software Engineering (EMSE). In press. This work is described in Chapter 4.

- Jiahuei Lin, Bram Adams, and Ahmed E. Hassan. 2022. On the Coordination of Vulnerability Fixes. Empirical Software Engineering (EMSE). Major revision. This work is described in Chapter 5.

- Jiahuei Lin, Mohammed Sayagh, and Ahmed E. Hassan. 2022. The Co-evolution of the WordPress Platform and its Plugins. ACM Transactions on Software Engineering and Methodology (TOSEM). In press. This work is described in Chapter 6.

Table of Contents

Abstract i

Acknowledgments iii

Dedication iv

Co-authorship v

List of Tables ix

List of Figures xi

1 Introduction 1
- 1.1 Research Hypothesis . . . 4
- 1.2 Thesis Overview . . . 7
- 1.3 Thesis Contributions . . . 11

2 Literature Review 12
- 2.1 Literature selection . . . 12
- 2.2 Coordination within software projects . . . 14
- 2.3 Coordination across software projects . . . 18

3 Upstream bug coordination 22
- 3.1 Introduction . . . 24
- 3.2 Background and Related Work . . . 28
- 3.3 Study Design . . . 36
- 3.4 Results . . . 41
- 3.5 Discussion . . . 77
- 3.6 Threats to Validity . . . 83
- 3.7 Chapter Summary . . . 86

4 Vulnerability coordination 88

4.1 Introduction . . . 90
4.2 Background . . . 95
4.3 Related Work . . . 100
4.4 Study Design . . . 103
4.5 Research Questions . . . 114
4.6 Discussion . . . 136
4.7 Threats to Validity . . . 138
4.8 Chapter Summary . . . 141

5 Multi-party coordination **143**
5.1 Introduction . . . 145
5.2 Background . . . 150
5.3 Related Work . . . 156
5.4 Study Design . . . 159
5.5 Results . . . 165
5.6 Discussion . . . 187
5.7 Threats to Validity . . . 194
5.8 Conclusion . . . 195

6 Release coordination **197**
6.1 Introduction . . . 199
6.2 Background and Related Work . . . 202
6.3 (Preliminary Question) What is the proportion of urgent releases that are due to incompatibility issues? . . . 206
6.4 Understanding the Relation between WordPress and Plugin Releases . . . 211
6.5 Threats to Validity . . . 233
6.6 Chapter Summary . . . 235

7 Conclusion and future work **237**
7.1 Thesis Contributions . . . 238
7.2 Future Research . . . 239

List of Tables

1.1 The difference between the studied software ecosystems along three dimensions. . . . 4
1.2 Types of the relations defined by Kraut and Streeter (1995). . . . 5

2.1 Venues from which an initial list of papers was obtained for the literature review. . . . 13

3.1 Basic statistics about the studied Linux distributions from January, 2005 to the end of 2019. . . . 38
3.2 Statistics of high-severity **upstream** bugs in each distribution. . . . 46
3.3 The dimensions of features in our models. The feature names in bold are new features with respect to the relation between upstream and distribution fixing bugs. . . . 68
3.4 Statistics of our regression models. The goodness of fit for the metrics of interest ($\Delta LR\chi^2$) is shown in a proportion to the overall $LR\chi^2$. . . . 74

4.1 The software systems and mailing lists where we extract the three kinds of repositories in the studied distributions. . . . 106
4.2 The descriptive statistics of common security vulnerabilities (CSVs) along with their related information. Note that a vulnerability report may map to several bug reports, and vice versa. . . . 123

5.1 The top 20 CNAs with the most valid CVE IDs assigned. Note that the three RQs study the bold CNAs, which are both CNAs and software vendors. Note that a vulnerability may affect several CNAs' products, i.e., both self-assigned and delegated vulnerabilities, so that the sum of the 4th and 5th columns is greater than the unique number of the studied vulnerabilities that affected the 13 CNAs, i.e., 39,409. . . . 161
5.2 The number of self-assigned and delegated vulnerabilities that affect the **application** type of product for the 5 duration (days) categories of the $Diff_{pd}$ delay for the studied CNAs. For each row, the bold number indicates the category with the largest number of respective vulnerabilities among the 5 duration categories. . . . 173

5.3 The number of self-assigned and delegated vulnerabilities that affect the **operating-system** type of product for the 5 duration (days) categories of the Diff_{pd} delay for the studied CNAs. For each row, the bold number indicates the category with the largest number of respective vulnerabilities among the 5 duration categories. 175
5.4 The comparison of the delay Diff_{pd} of patch releases between single-affected vs. multi-affected and delegated vs. self-assigned vulnerabilities for the studied CNAs. For each row, the bold number indicates the category with the largest number of vulnerabilities among the 5 duration categories. 182
5.5 The top 10 common weaknesses (CWEs) in the multi-affected vulnerabilities. 184
5.6 Comparison of the Diff_{pd} delay with prior work. 190

6.1 Causes for urgent releases based on extended Swanson categories of changes. Note that one release can address issues across several categories. Incompatibility is the 3^{rd} most common cause for urgent releases. 209
6.2 The types of relations that exist between a pair of related plugins. Note that a relation can have multiple reasons. 230
6.3 The types of resources that are shared between each pair of plugins that show a conflict. 231

List of Figures

1.1 An example of components, actors, and relations among them in a software ecosystem. 2
1.2 An example of relations between activities in software projects in a software ecosystem. Table 1.2 lists the definition of each type of the relations. 6

3.1 Simplified illustration of the three major roles involved in the maintenance of upstream packages in Linux distributions. The maintainer role takes the responsibility of fixing bugs in upstream packages, interacting with the upstream developers, and integrating upstream fixes into a distribution. While the "maintainer" role is distribution-specific, the person having that role in a distribution could be part of the upstream project or external to it. The developer role develops in-house packages and addresses bugs in these packages. 30
3.2 An illustration of the bug fixing process for upstream packages. The three large blue boxes with the blue-background text indicate the three stages in which a maintainer in a distribution, according to the `freedesktop.org` guidelines, collaborates with upstream developers to fix a bug in their package. The blue arrow lines indicate the common steps. The green arrow lines represent the alternative reaction of a maintainer (e.g., upstream is discontinued or severe bugs). The dashed arrow lines and boxes represent optional steps. The yellow dashed boxes represent the instances/roles/actions inside a Linux distribution or an upstream project. 31
3.3 An overview of our study approach. The solid arrow lines indicate the data sources used for all RQs. The dashed arrow lines indicate the particular data sources used for each specific RQ. We will describe the data sources in more details in the corresponding sections (RQ2 and RQ3). . . 36
3.4 An example of how to determine an upstream package. The *gedit* package is an upstream package since the red box indicates the source project, i.e., GNOME. 40
3.5 Two examples of bug fixes in the changelogs of upstream bug fixes in a package release in Debian and Fedora. 45

3.6 The normalized counts of upstream bugs (relative to package count) **that are recorded to be fixed** locally (left) and upstream (right) have dropped in Fedora over time. Both trends continue to fluctuate in Debian. 51
3.7 Comparison between the evolution of the normalized count of **reported bugs** normalized by the package count for both in-house and upstream packages. 52
3.8 The distribution of top 10 buggy categories of upstream packages. Fedora and Debian share 9 of 10 common buggy categories with different ranking orders. The categories that do not match are the **mail** category in Debian and the **graphics** category in Fedora. 53
3.9 Upstream bugs that are fixed locally (the yellow boxplots) take a longer time compared to in-house bugs (the green boxplots) across the studied distributions. In Debian, upstream bugs that are fixed locally (the yellow boxplot) take a shorter time compared to those that are fixed upstream (the orange boxplot). 58
3.10 The number of participants per day to fix an upstream bug locally (yellow) is equal to the number of participants per day to fix an in-house bug (green) across the studied distributions. Note that these boxplots remove the top 1% of data points for better visualization. 60
3.11 The number of comments per day in an upstream bug is similar to the number for an in-house bug. Note that these boxplots remove the top 1% of data points for better visualization. 61
3.12 The number of reference links per comment in an upstream bug that is fixed locally (yellow) is larger than the number for an in-house bug (green) in Debian, though they are similar in Fedora. Note that these boxplots remove the top 1% of data points for better visualization. 62
3.13 The number of patches per upstream package in one release of a given distribution is smaller than the number of patches per in-house package in Debian, while the number is equal for upstream and in-house packages in Fedora, though there are more outliers for upstream packages. . . 63
3.14 The code churn and survival time of each patch file for the upstream and in-house packages in each Fedora and Debian release. Across the studied distributions, the maintenance effort for the upstream packages is lower than for the in-house packages in (a), (b) and (c). 64
3.15 Overview of RQ3 model construction process. 65
3.16 The model building and validation process of the models used in RQ3. . . 71

4.1 An illustration of events in a vulnerability lifecycle started by the initial discovery of a vulnerability. Note that exploits can happen at any time during the lifecycle. 90

4.2 Overview of vulnerability management in a given distribution. The black solid and grey dashed arrow lines represent the periods of the five phases of tracking a vulnerability from the maintainer and user perspectives, respectively. To fix a vulnerability in the distribution, the required time (*i.e.*, the fixing time) is the sum of the integration and test periods (the black dotted arrow line). Note that the disclosure time of security advisories for end users (*i.e.*, when the user unawareness phase ends) could happen at any time during the maintainer awareness/integration/test periods. 97
4.3 Overview of our study approach. 105
4.4 An example of a comment in a Fedora bug report that mentions that a vulnerability fix has been pushed into the test repository. 106
4.5 The flow chart of the potential strategies that are used in Debian and Fedora for dealing with vulnerabilities. The bold numbers represent the number and proportion of CSVs in each strategy in Debian and the normal numbers are for Fedora. The two grey dotted diamonds and dotted lines represent the strategies related to Red Hat's embargo policy. The three grey dashed boxes group the strategies based on their purpose. The three boxes with grey background represent the quantitative analyses to measure the degree of potential collaboration in the three groups. 112
4.6 The time taken in the three phases from the maintainers' perspective. . . 115
4.7 Cumulative density plot of the duration of the integration phase of a security vulnerability. Debian and Fedora maintainers take a median of 6.1 and 5.3 days, respectively, to integrate fixes for security vulnerabilities after their bug reports were created. 117
4.8 Cumulative density plot for the testing phase of a security vulnerability. Debian maintainers test the fixes of 75% of security vulnerabilities in 17 days, while Fedora maintainers take 44 days for testing. 60% of vulnerabilities in Debian have a 0-day test phase. 118
4.9 The number of vulnerabilities during the two phases from the user perspective. 121
4.10 The (a) high-level CVSS 2.0 scores for CSVs, and a breakdown for (b) exploitability and (c) impact scores. 125
4.11 The duration of the three phases for CSVs from the maintainers' perspective. 126
4.12 The comparison of the length of each phase from the maintainers' perspective for CSVs. The awareness period is shorter than the integration and test phases. 127
4.13 The duration of the two phases from the user perspective for CSVs. 127

4.14 The distributions of the difference (in log-scale days) between the creation times of fixed CSV reports with the same CVE-ID in Debian and Fedora. The black lines indicate the median values of the two distributions. . . . 130
4.15 The comparison between the embargoed and non-embargoed CSVs in Fedora. (a) shows the distribution of the base, impact and exploitability scores. (b), (c), and (d) reflect the three phases from the maintainer perspective. (e) and (f) reflect the two phases from the user perspective. 133
4.16 Debian and Fedora maintainers do not speed up the integration of vulnerability fixes (in log-scale days) when peer fixes are available before the integration. . . . 135

5.1 Overview of multi-party coordination on fixing vulnerabilities. The dashed lines indicate possible coordination between the parties involved in fixing a vulnerability. The coordination activities include collecting information of a vulnerability, investigating root causes, discussing and developing a fix, and testing the fix. Note that software vendors can be a CNA and/or the discoverer and vice versa. . . . 154
5.2 Overview of our study approach. . . . 160
5.3 Illustration of the measurement of Diff_{pd}. The Diff_{pd} of a vulnerability is calculated by subtracting the disclosure date from the patch (available) date. A negative value (Diff_{pd} <0) indicates the patch is available before the disclosure date. . . . 164
5.4 Illustration of the measurement of Diff_{pp} (patch coherence). . . . 165
5.5 The number and percentage of self-assigned vs. delegated vulnerabilities in the 13 CNAs. . . . 167
5.6 Heat map of the assignor-assignee pairs based on the number of delegated vulnerabilities. Note that Mitre is the #1 assignor but is not included in the studied CNAs. . . . 168
5.7 The number and percentage of self-assigned vs. delegated vulnerabilities in the 13 CNAs, based on the three product types, i.e., application, hardware and operating-system. . . . 169
5.8 For the application type of products, delegated vulnerabilities have a higher CVSS 2.0 score than self-assigned vulnerabilities. . . . 170
5.9 The comparison of the delay (Diff_{pd}) of releasing patches between the severity levels (i.e., high, medium and low), self-assigned vs delegated, and the product types across the CNAs. . . . 176
5.10 The number and percentage of single-affected and multi-affected vulnerabilities in each of the studied CNAs. . . . 180

5.11 The three metrics of CVSS scores for single-affected and multi-affected vulnerabilities that affect one (i.e., either application or operating-system) or two types (i.e., both) of products. 181
5.12 Comparison of the delay ($Diff_{pp}$) between when the first and last patches are available across the studied CNAs, based on affected product types and the number of affected variants. A variant represents a flavour of a product designed for particular conditions (e.g., x84 or x64). 183
5.13 The comparison of the delay ($Diff_{pp}$) between when the first and last patch dates, base on the top 10 vulnerability weaknesses (CWEs) in the multi-affected vulnerabilities. Table 5.5 reflects the details of the top 10 CWEs. 184
5.14 Comparison of the delay ($Diff_{pp}$) between the first and last patches. "AGMM" represents any pair of vendors from Adobe, Google, Microsoft, and Mozilla. "AOR" represents any pair of vendors from Apple, Oracle, and Red Hat. 185
5.15 The number of CNAs in the CVE program. 192

6.1 An example of a readme file and the web page displaying the associated information in the readme file. 203
6.2 The distribution of plugins with their respective number of urgent releases. 497 out of the studied plugins have at least one urgent release, while three do not have any urgent release. 208
6.3 An illustration of the **"off-sync period"**, which consists of the time between the release of a new version of WordPress and when a plugin supports that new version. We refer to the release that supports a new WordPress version by **"support release"**. During the off-sync period, a plugin can release an **"off-sync release"**, which is shipped by a plugin before it declares support to the latest available version of WordPress. We refer to any release that is shipped after the support of the latest WordPress version by **"in-sync"** release. 213
6.4 The distribution of plugins shipping a support release for a new WordPress version. Early spikes (i.e., early and fast adopters) appear across all minor versions of WordPress version 4. The vertical dashed lines indicate when WordPress released a version. Note that this plot considers just version 4 and we observe similar results for the other WordPress versions (i.e., 2, 3, and 5). 214
6.5 The comparison of the distribution of urgent release ratio for support releases based on their adoption speed. Slow and normal adopters are more likely to be followed by an urgent release. 216

6.6 The comparison of the percentage of releases that change a WordPress API among in-sync, off-sync and support releases. Plugin developers are more likely to make changes related to the associated WordPress APIs in in-sync releases compared to support releases. 217
6.7 The comparison of the distribution of urgent release ratio among in-sync, off-sync and support releases. Off-sync releases are more likely to be followed by an urgent release compared to support and in-sync releases. 219
6.8 The number of plugins that have a certain percentage of its support releases that are urgent releases. For example, 102 plugins have 10-20% of their support releases that are urgent releases. 221
6.9 The comparison of urgent release ratio for support releases with large v.s. few changes. Support releases with a large amount of code changes are more likely to be followed by an urgent release. 222
6.10 The comparison of urgent release ratio for releases with a change related to WordPress APIs v.s. ordinary releases (no changes related to WordPress APIs). Releases with changes related to WordPress APIs are more likely to be followed by an urgent release compared to ordinary releases (that not have changes related to WordPress APIs). 223
6.11 An example of a pair of plugins, the *All-In-One-Wp-Security-and-Firewall* plugin and the *Woocommerce* plugin, based on the release note of the the *All-In-One-Wp-Security-and-Firewall* plugin. 225
6.12 The distribution of the number of related plugins for a given plugin. The numbers at the top of each bin indicate the unique number of related plugins and the percentage of these related plugins that are in the top 500 most-released plugins. 227
6.13 The number of plugins that have a certain percentage of its peer-triggered releases that are urgent releases. For example, 53 plugins have 10-20% of its peer-triggered releases that are urgent releases. 228
6.14 The comparison of urgent release ratio for ordinary v.s. peer-triggered releases. Peer-triggered releases are less likely to be followed by an urgent release. 229
6.15 The incompatibility last (lower-bound estimation) between pairs of plugins with explicit or implicit relations. The blue number above each box-plot represents the number of pairs of plugins are considered in each relation. 232
6.16 The explanation for the examples of sharing resources between plugins in Table 6.3. 233

CHAPTER 1

Introduction

A software ecosystem consists of thousands of projects. Software projects within a software ecosystem are inter-connected with activities in one project impacting activities in other projects. For example, one project imports another project in the ecosystem as a library to reduce the development effort. The activity of (importing) constructs a relation between the two projects.

Figure 1.1 illustrates many complex relations among software projects in a software ecosystem. These software projects in the ecosystem not only have the relations (orange arrow lines) among themselves but also have relations with the ecosystem, e.g., leveraging the authentication service offered by the ecosystem. In some cases, a software ecosystem (e.g., the Linux ecosystem) contains several distributions that package part of these software projects to meet the needs of a particular group of users.

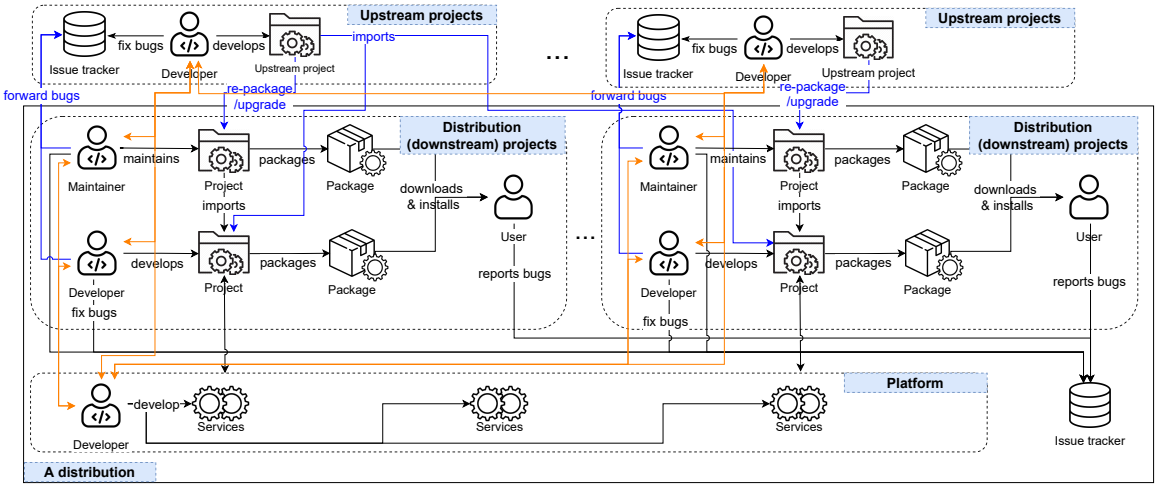


Figure 1.1: An example of components, actors, and relations among them in a software ecosystem.

Coordination is the management of relations between activities, including classifying different types of relations and identifying the process that can be used to manage these relations (Malone and Crowston, 1994). In the context of software ecosystems, the developers of a project within a software ecosystem face many challenges when they deal with the interrelations. For example, fixing a bug in an upstream project that was imported as a library. Intuitively, the developers report the bug to the upstream project and wait for the fix. However, coordination between the upstream project and the developers might not be as easy as foreseen. For example, the upstream project might be unresponsive or have resource constraints, or the upstream does not consider a bug to be an actual bug. In these cases, the downstream developers are compelled to develop their own fix and deviate from the source code of the upstream project.

The coordination between developers is one of the major root causes of problems in software development (Kraut and Streeter, 1995). Changes in a software project are inevitable and thus impact its associated projects where their developers have to react

to the changes. Several researchers have empirically studied professional software development teams to gain an understanding of coordination mechanisms (e.g., knowledge transfer and task assignment processes) (Reed and Knight, 2010; Desouza et al., 2006; Guzzo and Shea, 1992). For example, knowledge transfer is an important process for developing a common understanding at all stages of software development (Desouza et al., 2006) and insufficient knowledge transfer increases the likelihood of potential risks in software development (Reed and Knight, 2010).

However, only a few prior studies focused on coordination between developers in software ecosystems and most of these studies focused on the software ecosystems that emerge due to programming languages (e.g., Python (Ma et al., 2020a), R (Decan et al., 2016)). For example, Ma et al. (2020a) studied the impact of poor coordination in 22 popular projects in the scientific Python ecosystem. The authors identified 1,132 buggy functions in the 22 projects that were introduced by respective upstream projects. Most (96%) of these bugs were not reported in the 22 projects (when the authors performed the study). Decan et al. (2016) reported that failures in the upstream projects caused issues in the downstream projects in the R ecosystem. For example, 41% of R packages were broken due to the incompatible changes in the upstream projects.

In this thesis, we empirically study developer coordination practices for software quality assurance within a software ecosystem. We investigate coordination practices in two large-scale software ecosystems: **Linux** and **WordPress**. In particular, we study the coordination practices related to: (1) upstream bug coordination, (2) vulnerability coordination, (3) vulnerability fixing and disclosure coordination, and (4) release coordination of co-evolving software projects.

1.1 Research Hypothesis

Research hypothesis: A good understanding of coordination activities and associated best practices around such activities in large-scale open source ecosystems is essential to define best practices and challenges that can be shared widely across practitioners and academics.

This thesis aims to study coordination practices for quality assurance activities in large-scale software ecosystems. Developers rely on a software ecosystem to facilitate the development of their own projects and easily distribute these projects to users. Therefore, there exists sophisticated relations among the projects in the software ecosystem. A deeper understanding of how developers within a software ecosystem coordinate to ensure the quality of their projects as well as the software ecosystem as a whole is essential to identify best practices and common challenges.

Table 1.1: The difference between the studied software ecosystems along three dimensions.

Software ecosystems	**Size**	**Control**	**Communication**
Linux	Consists of 600+ active distributions[1] and each distribution usually has 50K+ packages	Each distribution has its own control platforms, including code repositories, issue trackers, CI/CD systems, and developer tools	Email, web platform (e.g., issue trackers), IRC channel
WordPress	60K+ active packages[2]	A code repository and issue tracker	Web platform (i.e., the issue tracker)

[1] https://truelist.co/blog/linux-statistics
[2] https://blog.hubspot.com/website/wordpress-stats

Software ecosystems are formed for different reasons and therefore vary across

different dimensions. Table 1.1 contrasts the studied ecosystems along three dimensions: size, control and communication. In terms of size, the Linux ecosystem consists of more than 600 active distributions[1] and each of them is at the center of a sub-ecosystem with its own developers and projects, while the WordPress ecosystem does not have sub-ecosystems. As sub-ecosystems exist in the Linux ecosystem, each Linux distribution (sub-ecosystem) builds its own control platforms (e.g., code repositories and issue trackers) for its own software projects. These control platforms might not have been aggregated by an identifier that developers could leverage to track the coordination activities. For example, tracing a bug after it is created, fixed in one (or more) code repository, tested, and deployed on production. As a result, the information about particular coordination activities are commonly scattered across these control platforms. In addition, due to the large number of developers, the variety of communication channels (e.g., email, IRC) in the Linux ecosystem increases the complexity of coordination activities for quality assurance.

Table 1.2: Types of the relations defined by Kraut and Streeter (1995).

Types of the relations	Description
Shared resources	Whenever some activities share limited resources, a resource allocation process is needed to manage the interdependencies among these activities.
Producer and consumer	One activity produces something that is used by another activity.
Co-occurrence constraints	Whenever multiple activities need to co-occur.
Task decomposition	A group of activities is needed to achieve a collective goal.

A large amount of coordination activities for quality assurance occur within a software ecosystem. Kraut and Streeter (1995) defined several types of relations that we can leverage to study coordination in software ecosystems, as shown in Table 1.2. Coordination activities occur *within a software project* (e.g., code review) as well as *across*

[1] https://truelist.co/blog/linux-statistics

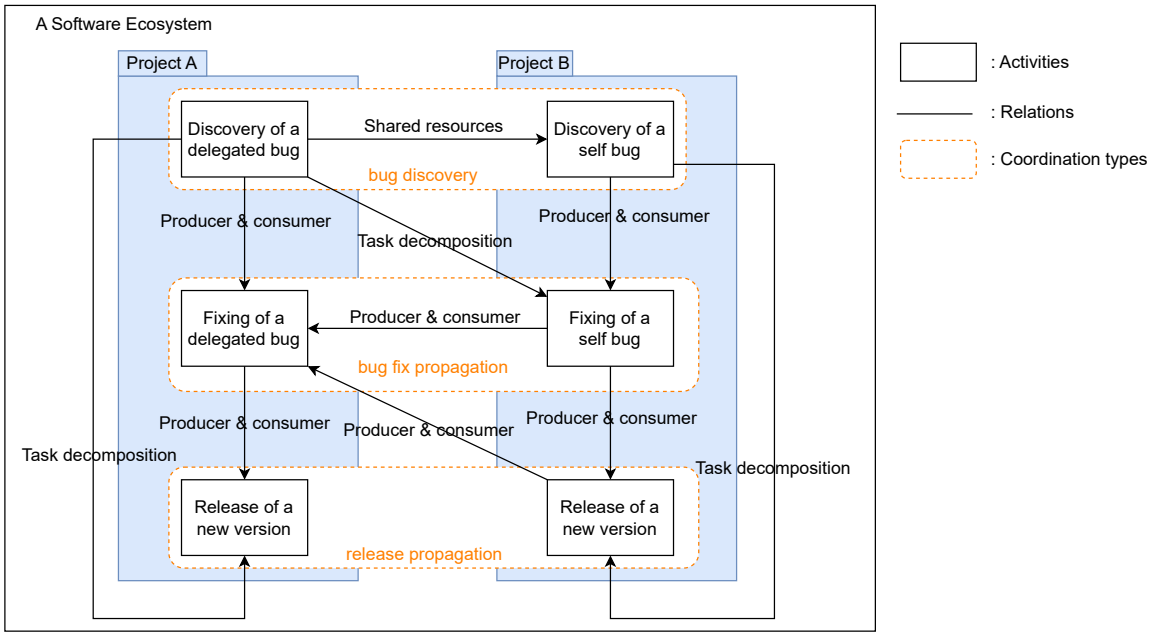


Figure 1.2: An example of relations between activities in software projects in a software ecosystem. Table 1.2 lists the definition of each type of the relations.

software projects (e.g., due to cross project dependencies). Figure 1.2 shows an example of activities and the relations among these activities in the case where one project imported another project as a library in a software ecosystem. A delegated bug in a project (Project A) indicates that a bug was introduced by a dependent project (Project B), as opposed to a self bug. There is a shared-resource relation between the activity of a delegated bug found (in Project A) and the activity of the discovery of the self bug in Project B. After these two activities occur, two corresponding activities, i.e., the fixing of a delegated bug and the fixing of a self bug, occur in Project A and B, respectively. After a self bug is fixed, a new version of Project B will be released. Similarly, Project A will release a new version after the fixing of a delegated bug. Therefore, there is a producer-consumer relation between the fixing of delegated/self bug and the release

of a new version in Project A/B. In addition, the discovery of a delegated bug, the fixing of a delegated bug and the release of a new version of Project A occur to achieve a collective goal, i.e., fixing the delegated bug. Hence, there is a task-decomposition relation among aforementioned activities.

1.2 Thesis Overview

Chapter 2 describes a literature survey regarding empirical studies of software ecosystems. Chapter 3 describes an empirical study of upstream bug fixing in Linux distributions. Chapter 4 describes an empirical study of vulnerability management in Linux distributions. Chapter 5 presents an empirical study of multi-party coordination for vulnerability fixing and disclosure. Chapter 6 presents an empirical study of the co-evolution coordination of software projects in the WordPress ecosystem. Below, we summarize each of these four empirical studies:

1.2.1 Upstream bug fixing in Linux distributions (Chapter 3)

A Linux distribution consists of thousands of packages that are either developed by in-house developers (in-house packages) or by external projects (upstream packages). Leveraging upstream packages speeds up development and improves productivity, yet bugs might slip through into the packaged code and end up propagating into downstream Linux distributions. Maintainers, who integrate upstream projects into their distribution, typically lack the expertise of the upstream projects. Hence, they could try either to propagate the bug report upstream and wait for a fix, or fix the bug locally and maintain the fix until it is incorporated upstream. Both of these outcomes come

at a cost, however, little is known about how high-severity bugs are fixed in upstream packages. We show that a similar proportion of high-severity upstream bugs are explicitly reported as being fixed by upstream (at most 14%) and by the distributions (13%), while the majority (at least 73%) of the high-severity upstream bugs do not explicitly record the source of their fix. Upstream fixed bugs make users wait for a longer time to get fixes and require additional information compared to fixing upstream bugs locally by the distribution.

1.2.2 Vulnerability management in Linux distributions (Chapter 4)

Since the coordination history of ordinary bugs (even for high-severity bugs studied in Chapter 3) is less likely to be explicitly recorded in an issue tracking system, we investigate coordination activities for the urgent release of high severity bugs, i.e., vulnerabilities, that have a more crucial impact, such as leading to the losses of revenue, reputation and trust. In the Linux ecosystem, upstream projects usually do not approach users directly, instead, they rely on Linux distributions to integrate and distribute the upstream projects to millions of end users. To this end, an upstream project might be packaged into several Linux distributions so that a vulnerability can propagate across multiple distributions. In this chapter, we investigate the lifecycle of fixing vulnerabilities, analyze how fast it takes for a vulnerability to go through each phase of its lifecycle, characterize the commonly occurring vulnerabilities that affect both distributions, and identify the practices that developers use to fix vulnerabilities. We observe that the vulnerability testing period (i.e., the period from when a vulnerability fix is committed for testing to when the vulnerability fix is released) accounts for the largest number of days in the lifecycle of fixing vulnerabilities. The majority of vulnerabilities are common to

both studied distributions.

1.2.3 Multi-party coordination for vulnerability fixing and disclosure (Chapter 5)

Given that vulnerabilities are high-severity and have a large impact on software systems, we expand our study to investigate the coordination activities for vulnerability fixing and disclosure across CVE Numbering Authorities (CNAs). The Common Vulnerabilities and Exposures (CVE) program[2] is dedicated to analyzing vulnerabilities, then to assigning a unique ID to them and disclosing the vulnerabilities to affected software vendors. A CNA is a key partner in the CVE program responsible for assigning an official ID to a CVE and registering a description of the vulnerability in order to communicate it to the other CNAs and the affected software vendors. In addition, each CNA could be a software vendor that has its own products. To avoid the disclosure of vulnerabilities before the deployment of a fix, the CNAs and the affected vendors need to coordinate a proper schedule for the disclosure of vulnerabilities and the release of their fixes, indicating **multi-party coordination**. Our results show that the studied CNAs discover and assign (CVE IDs) for the majority of vulnerabilities that affect their own products, which we refer to as *self-assigned vulnerabilities*. While the vulnerabilities that are assigned for other CNAs' products, which we refer to as *delegated vulnerabilities*, tend to be more risky than the self-assigned vulnerabilities, (median CVSS score of 7.5), we observe that their fixes are released at a slower pace.

[2] https://cve.mitre.org/

1.2.4 Coordination of software projects co-evolution on WordPress (Chapter 6)

WordPress is a plugin-based software ecosystem that is different from traditional software ecosystems. WordPress plugins have a high coupling relation with the WordPress platform. WordPress has more than 54k plugins that are available to end-users. The richness of these plugins makes WordPress as one of the most popular content management platforms, which is used by 61.8% of all the websites.[3] Due to the richness of plugins, the freedom to intermix plugins, and their varying release cycles, the users of WordPress often must cope with incompatibility issues, which discourage users from upgrading their WordPress and plugins.[4] WordPress and its plugins must coordinate as they co-evolve. The high coupling between a plugin and its platform and other plugins causes incompatibility issues that occur during the co-evolution of a plugin and its platform as well as other plugins. In fact, incompatibility issues represent a major challenge when upgrading WordPress or its plugins. According to our study of the top 500 most-released WordPress plugins, we observe that incompatibility issues represent the 3^{rd} major cause for bad releases, which are rapidly (within the next 24 hours) fixed via urgent releases. 32% of these incompatibilities are between a plugin and WordPress while 19% are between peer plugins.

[3] https://w3techs.com/technologies/overview/content_management
[4] https://wordpress.org/news/2009/10/plugin-compatibility-beta/

1.3 Thesis Contributions

The empirical results that are presented and discussed in this thesis demonstrate the variety of coordination practices in software ecosystems and provide actionable information to improve quality assurance practices in software ecosystems. The thesis makes the following key contributions:

- The lack of tracking of the bug fixing process across upstream and distributions (in Chapter 3) suggests the need of developing automated tools to record the traceability information and enabling more effective coordination.

- Our results that are associated with fixing vulnerabilities in Linux distributions (in Chapter 4) show that distribution maintainers collaborate closely across distribution boundaries before vulnerability fixes are available, but less afterward, i.e., integrating vulnerability fixes into the distribution.

- We uncover the inconsistency of releasing vulnerability fixes for shared vulnerabilities in the multi-party coordination (in Chapter 5). We provide dimensions related to the inconsistency to help developers better manage vulnerabilities.

- The release coordination practices of co-evolving software projects in a high coupling software ecosystem (in Chapter 6) indicate that it is important to develop mechanisms to manage the interference of resources that are shared by peer projects.

CHAPTER 2

Literature Review

THIS thesis studies coordination practices for software quality assurance in large-scale software ecosystems. In this chapter, we first explain our literature selection process (Chapter 2.1). Then we survey the related work to coordination within software projects (Chapter 2.2) and across software projects (Chapter 2.3).

2.1 Literature selection

Software ecosystems have gained attention in the research area regarding software development in the last decade (Jansen et al., 2009). Hence, we search for papers that were published between 2010 and 2021 in the major software engineering journals and

Table 2.1: Venues from which an initial list of papers was obtained for the literature review.

Venue Type	Venue Name	Abbreviation
Journal	IEEE Transactions on Software Engineering	TSE
Journal	Empirical Software Engineering	EMSE
Journal	ACM Transactions on Software Engineering and Methodology	TOSEM
Journal	Journal of Systems and Software	JSS
Conference	International Conference on Software Engineering	ICSE
Conference	ACM SIGSOFT Symposium on the Foundation of Software Engineering/ European Software Engineering Conference	FSE/ESEC
Conference	International Conference on Automated Software Engineering	ASE
Conference	International Conference on Software Maintenance	ICSM
Conference	International Conference on Software Maintenance and Evolution	ICSME
Conference	International Conference on Software Analysis, Evolution, and Reengineering	SANER
Conference	International Conference on Mining Software Repositories	MSR
Conference	International Conference on Cooperative and Human Aspects of Software Engineering	CHASE
Conference	International Conference on Open Source Systems	OSS
Conference	International Conference on Global Software Engineering	ICGSE
Conference	International Conference on Program Comprehension	ICPC

conferences. Table 2.1 shows the research venues from which we started our literature review process.

First, we select the initial list of papers using the following three sets of keywords: (1) *software quality assurance activities*, (2) *social interactions* and *software quality*, and (3) one of the following keyword pairs: *upstream bugs*, *bug fix propagation*, or *upstream release propagation*. Then, we check the citations of each selected papers to include relevant papers not in the initial list. We read the abstract and introduction of the citations and remove papers that do not relate to our topics of interest.

2.2 Coordination within software projects

Maintaining large software projects involves the effort of multiple developer teams. Teams of developers divide a large software project into manageable components with interrelations, i.e., internal coordination. These teams communicate and coordinate in effect to maintain the whole project. In this process, the coordination (social) interactions are crucial to ensure the quality of the whole project. Datta (2018) extracted characteristics of developer coordination activities in distributed software development teams. The authors reported that a higher level of developer connections (e.g., knowledge sharing) in a team correlated with a higher defect count, whereas more clustering (i.e., developers share a common task or responsibility) related to fewer defects. Bird (2011) reported that core developers have a higher importance in coordination between distributed developer teams in a project. Wagstrom and Datta (2014) reported that temporal separation has a larger impact on the time to respond in developer coordination than geographic distance in a large commercial software project.

In the context of distributed development, congruence between coordination activities and technical requirements in a software project shortens development time (Cataldo et al., 2006). Several researchers evaluated the alignment of coordination (social) activities and technical requirements, i.e., socio-technical congruence (Cataldo et al., 2008), with respect to productivity and quality. Meneely and Williams (2011) performed a survey from 124 developers in three popular open source projects to corroborate metrics for studying the gap between socio-technical congruence align with developer perceptions. Kwan et al. (2011) observed that the gap (mean values) between socio-technical congruence correlated inversely with the likelihood of software build success in the IBM Rational Team Concert project. Cataldo and Herbsleb (2012) also

reported that the gap between socio-technical congruence carried out by the developers significantly increased software defect-proness in two large-scale projects.

Another line of research focused on coordination issues and their impact in distributed software development. Developers work independently on their own task by forking a copy of the source repository and editing changes locally. Developers then make pull requests to merge their local changes to the source repository. This allows developers to work on the same source repository parallelly to meet the schedule but also increases the complexity of code changes. As a consequence, a conflict arises when two developers make inconsistent changes to the same part of the source repository. Brindescu et al. (2020) observed that 20% of merges caused conflicts in 143 open-source projects. The code associated with a merge conflict is more likely to have a bug. As developers might defer resolving conflicts due to the complexity of merge conflicts, it can lead to increased complexity of merge conflicts and end up with throwing the changes and re-implementation (Nelson et al., 2019). In addition, Bird and Zimmermann (2012) observed that merge conflicts lead to delay of development and developer frustration. Brun et al. (2011); Guimarães and Silva (2012); Brun et al. (2013) proposed techniques or tools to detect merge conflicts as early as possible. Maddila et al. (2021) developed a concurrent edit tool to address conflicts in 234 repositories with 26k pull requests at Microsoft. The tool made 775 recommendations and 70% of them were rated as useful by 48 users.

Coordination is necessary to ensure software quality in distributed software development. However, software assurance activities (e.g., code review and source code inspection) are usually time-consuming and costly. To reduce the involved effort in these activities, many prior studies focus on improving the performance of defect prediction

models. For example, Kamei et al. (2010) studied the performance of defect prediction models in teams of the needed costs to perform quality assurance activities (e.g., the number of code changes needed testing). The results showed that metrics related to the coordination process (e.g., the number of revisions) outperformed code metrics (e.g., LOC). Ulan et al. (2021) revealed that calculating the weights of software metrics in an unsupervised learning approach when a ground truth is lacking improved the involved effort of quality assurance activities by bug prediction models. For example, the involved effort to inspect a large file containing many bugs could be smaller than the involved effort to inspect multiple small files containing small bugs in each of them, due to a higher density. Jiarpakdee et al. (2020) evaluated model-agnostic techniques for defect prediction models and reported that instance explanations of the prediction were necessary and useful by the majority of the participants. Bettenburg and Hassan (2013) observed that metrics related to issue discussions had similar explanatory power as traditional models built by code-based metrics in the Eclipse and Mozilla projects. For instance, the length of discussions in bug reports correlated with an increased risk of post-release bugs. Tourani and Adams (2016) observed that metrics related to discussions in bug reports and code review process improved precision and recall of the explanatory models up to 10% in OpenStack and Eclipse. The results revealed that the review time and issue discussion lag were two of the most important metrics to the likelihood of defect proneness. Ryu et al. (2016) proposed using common knowledge (e.g., defect-proneness code patterns) in different projects (knowledge transferring) to improve the performance of cross-project bug prediction.

Several prior studies proposed just-in-time defect prediction models to detect a defect-inducing commit at the development phase instead of the testing phase to

reduce coordination activities at the testing phase of software development. Kamei et al. (2012) constructed models to predict defect-inducing software changes when they were made (Just-In-Time) to reduce substantial time used in coordination for quality assurance activities. The authors observed that the average time interval since the previous change was a risk-decreasing factor. Tabassum et al. (2020) proposed a learning approach to improve the performance of Just-In-Time defect prediction models. The approach leveraged the coordination data of quality assurance activities from other projects at the early stage of the model construction and afterward updating the model using the data within the project. Kamei et al. (2016) reported that the performance of Just-In-Time models trained by all available coordination data in other projects is better than by a selected set of coordination data from similar projects.

Coordination activities are usually triggered by core developers in mature software projects (Steinmacher et al., 2016; Mendez et al., 2018; Steinmacher et al., 2014). For example, Gharehyazie et al. (2015) studied how developers become a committer of a project based on coordination activities in 6 Apache open-source projects. The results revealed that developers' early activities (the first three months) were an important factor to the likelihood of becoming a committer. However, the power of using coordination activities as an indicator decreased when a project became mature, implying that it is unlikely to become a new committer when the project has a group of trustworthy developers. Steinmacher et al. (2014) reported 38 barriers in 7 categories that hinder developers' contribution to an open source project. For example, how to start the project from scratch and make it work properly, which due to lack of documentation (another of the 38 barriers). In addition, Ashraf et al. (2021) reported that developers

maintained more coordination activities with developers outside their own project in 10 open-source projects. This kind of coordination activities might be due to the need of remaining robust against the disruption of leaving developers and quick onboarding of new members.

Many prior studies focused on coordination in distributed software development and reducing the limited resources in quality assurance activities by factoring coordination activities for quality assurance to improve bug prediction models. The majority of these activities are associated with internal coordination within a project.

2.3 Coordination across software projects

Software projects establish a relation when they leverage other projects (i.e., upstream projects) to facilitate the speed of software development. Developers have to maintain the updates of these upstream projects to ensure the security and quality of their own projects, i.e., external coordination. Canfora et al. (2011) identified the prevalence of bug fixes between the FreeBSD and OpenBSD kernels. To fix bugs that occur between projects, developers in the two projects must communicate with each other to share the information of bugs. Kula et al. (2018a) studied the lag of API updates over 4,600 GitHub software projects containing over 2,700 library dependencies. The authors reported that 81.5% of the studied projects kept their outdated dependencies because upgrading dependencies required additional workload and responsibility.

A few prior works studied the fix propagation in shared code snippets. Thummalapenta et al. (2010) reported that only 16% of shared code snippets with late

propagation in four different Java and C software projects. Barbour et al. (2011) defined 8 types of late propagation in shared code snippets and 2 out of the 8 types were riskier than the others in terms of fault-proneness. For example, in a pair of clones, when both clones are modified, causing a divergence and then both are modified to re-synchronize the clone pair.

The npm ecosystem is a striving ecosystem with accelerating growth of packages and increasing dependencies between them (Wittern et al., 2016). However, a minority of packages account for most of the dependency updates and a higher number of transitive dependencies is an indicator of the fragility of a package (Decan et al., 2018a). Several researchers conducted studies on the lag of fixes propagation from the package dependency network. Decan et al. (2017a) observed that upstream projects are updated more frequently than downstream projects. Chinthanet et al. (2021) identified the inconsistent release of a vulnerability fix between the vulnerable project and its downstream projects. When the vulnerable project released a fix in a patch version, the downstream projects released 36% of their releases in minor versions and 22% in major versions, compared to 21% in patch versions. Decan et al. (2018b) reported that it took a considerable long time before a vulnerability was fixed in the downstream (dependent) projects in a 6-year period of the npm dependency network. For example, it took about 52 months before 50% of all low-severity vulnerabilities got fixed, 20 months for 50% of all medium-severity vulnerabilities, and 32 months for 50% of all high-severity vulnerabilities. Bogart et al. (2016) conducted a manual study to understand the cost of downstream developers when facing breaking changes in dependencies. Although the interviewed downstream developers maintained a broad awareness of changes in upstream projects through various social means, most of these developers employed

a reactive strategy where they waited for notifications from others. On the other hand, Cogo et al. (2019) reported that downstream developers might downgrade their dependencies when facing breaking changes, though the speed of dependency downgrades was slower than typical releases.

Despite the npm ecosystem, some researchers focused on coordination of fix propagation in other ecosystems that emerge due to programming languages (e.g., Python (Ma et al., 2017, 2020b), Java (Brito et al., 2016, 2018), R (Decan et al., 2016)). For example, Ma et al. (2017) identified the difficulty for downstream developers to resolve upstream bugs in the scientific Python ecosystem, and observed that workarounds (i.e., temporary and local solutions) are common practices to reduce the long-lasting impact of upstream bugs. Ma et al. (2020b) studied the impact of upstream bugs in the Python ecosystem, such as buggy code hidden in downstream projects until the failure inducing preconditions are triggered. The authors proposed a technique to identify the hidden buggy code in the downstream projects. The authors identified 1,132 downstream modules using 31 buggy upstream functions but only 47 out of these 1,132 modules have ever reported the buggy functionality in the past.

Some researchers focused on coordination activities in software product lines. A software product line indicates a set of software products that are offered by a company or a brand, share similar features or functions, and target a similar market segment or mission, i.e., business-oriented (Van der Linden et al., 2007). Building software product lines improves the development process of a product and reduces the time of a product entering into a particular market, i.e., cost-effective. For example, fixing a bug in the variants of a particular product. However, such a product line is typically built and used by certain corporations where open-source software communities are less likely

to be involved.

Most prior studies related to coordination focused on software ecosystems that emerge due to programming languages (e.g., Python, npm). However, software ecosystems that emerge due to operating systems (e.g., Linux) or frameworks (e.g., WordPress) are less explored.

CHAPTER 3

Upstream bug coordination in the Linux distributions

Vulnerabilities in software systems not only lead to loss of revenue, but also to loss of reputation and trust. To avoid this, software projects strive to remedy vulnerabilities rapidly for their customers. However, in open-source development, the projects do not always control the distribution of their software themselves, but instead typically rely on Linux distributions to integrate and distribute upstream projects to millions of end users, which increases the difficulty of vulnerability management. In addition, an upstream project is usually packaged into several Linux distributions so that a vulnerability can propagate across multiple distributions via the upstream project. In this work, we empirically investigate a large number of vulnerabilities registered with the Common Vulnerabilities and Exposures (CVE) program in two popular Linux distributions, i.e., Debian (21,752 CVE-IDs) and Fedora (17,434 CVE-IDs), to study the practices of vulnerability management in such ecosystems. We investigate the lifecycle of fixing vulnerabilities, analyze how fast it takes for a vulnerability to go through each phase of its lifecycle, characterize the commonly occurring vulnerabilities that affect both distributions, and identify the practices that developers use to fix vulnerabilities. Our results suggest that the vulnerability testing period (i.e., the period from when the vulnerability fix is committed for testing to when the vulnerability fix is released) accounts for the largest number of days (median of 15 days) in Fedora. 74% (i.e., 16,070) and 92% (i.e., 16,070) of the vulnerabilities in Debian and Fedora, respectively, occur in both Linux distributions, which we refer to as common security vulnerabilities (CSVs). This result is impacted by the package selection and customization of

the distributions. Finally, on a representative sample of 345 fixed CSVs, we find that upstream projects were responsible for fixing 303 (85%) and 267 (76%) out of the 345 CSVs in Debian and Fedora, respectively, with distribution maintainers integrating those fixes. Our work aims to gain a deeper understanding of the current practices in the vulnerability management of Linux distributions, and propose suggestions to distribution maintainers to better mitigate the risks of vulnerabilities.

An earlier version of this chapter is published in the Empirical Software Engineering Journal (EMSE) (Lin et al., 2022b).

3.1 Introduction

A software ecosystem consists of many software projects with relationships among them and a shared platform (Van Den Berk et al., 2010). Linux distributions (e.g., Debian) are one of the most well-known software ecosystems. In such distributions, software projects developed internally (in the distribution) or externally (upstream, potentially customized in the distribution) are integrated with each other into a coherent product. To achieve this, maintainers of the distribution package up all projects in a standard format using the distribution's package manager, recording any metadata such as dependencies, licenses, etc. The package manager is able to enforce the metadata for the end users, guaranteeing that each installed package has all the dependencies it needs.

Being the middlemen between the upstream projects and the end users, the distribution maintainers spend a large amount of effort and resources maintaining upstream packages, for example, updating to a new version, applying and reporting bug fixes, etc (Adams et al., 2016; Anbalagan and Vouk, 2009; Storey et al., 2016). Prior studies investigated the integration challenges of upstream packages, such as estimating the effort to select the proper packages or ensuring the security of the selected packages (Stol et al., 2011; Li et al., 2005; Merilinna and Matinlassi, 2006). Prior studies also surveyed the practices of fixing bugs that are introduced by such integration (Ma et al., 2017; Ding et al., 2017) and the associated effect to developers during integration (Ma et al., 2020b). However, prior studies did not investigate (1) to what extent bug fixing effort is spread across upstream developers and distribution maintainers, and (2) the cost going into upstream package management for distribution maintainers.

One effort to reduce the distribution maintainers' required maintenance effort of upstream packages are freedesktop.org's *distribution-neutral packaging guidelines*[1]. These guidelines suggest that distribution maintainers should follow the development practices of upstream projects, e.g., sending changes to upstream. Nevertheless, the guidelines list some exceptions for distribution maintainers to make a local fix, e.g., security issues, unresponsive upstream developers, or discontinued projects.

In such exceptional cases, distribution maintainers need to step up and develop their own patches, i.e., bypassing the upstream bugs, to protect their users from potential risks of upstream code (Ma et al., 2017). However, this practice is not straightforward since distribution maintainers are not familiar with the upstream projects to propose the most appropriate fix, or they could even introduce new bugs in such patches (Hauge et al., 2010; Li et al., 2005; Merilinna and Matinlassi, 2006). Furthermore, while distribution maintainers can create patches, they also need to maintain those patches and integrate them with the subsequent releases of the upstream package, until it incorporates the fixes (Ma et al., 2017). Even then, the typical lack of documentation for changes in upstream releases hinders the ability of distribution maintainers to understand which upstream release incorporates their needed fixes (Stol et al., 2011; Ma et al., 2017).

In this chapter, we measure how upstream packages in a distribution are managed in terms of fixing bugs and compare the costs of locally fixing bugs of upstream packages by distribution maintainers to the costs of upstream fixes to such bugs, as well

[1] https://www.freedesktop.org/wiki/Distributions/Packaging/WhyUpstream/

as locally fixing bugs of in-house packages (i.e., the packages developed by the distribution itself). We investigate 143,362 high-severity (i.e., high, urgent, critical) bugs[2] across two popular Linux distributions, i.e., Debian and Fedora, that use two different types of package managers and different bug management strategies. Debian is a purely open-source distribution, while Fedora forms the core of the commercial Red Hat distribution. We focus on high-severity bugs since they typically represent fatal errors or crashes that are important for developers to fix (Menzies and Marcus, 2008; Lamkanfi et al., 2010).

We structure our study by answering the following research questions, yielding the following key findings:

RQ1: What percentage of upstream bugs are fixed upstream versus locally?

This first RQ analyzes the prevalence of high-severity bugs in upstream vs. in-house packages. High-severity bugs are dominant in the upstream packages of the studied distributions, yet for 27% of them we found explicit traces of where they were fixed (14% fixed upstream, 13% fixed in the distribution), while for the remaining 73% it was unclear where they were fixed in Debian. The proportion of high-severity bugs per upstream package that are fixed by upstream has dropped over time in Fedora and has been fluctuating in Debian. Similarly, the proportion of opened bugs per upstream package has also dropped over time in the studied distributions. Moreover, Debian and Fedora share 9 out of the top 10 categories of buggy upstream packages.

RQ2: What is the time and cost for fixing high-severity upstream bugs locally in distributions and maintaining such distribution fixes? We observe that bugs in

[2]Unless noted otherwise, the remainder of this chapter refers to high-severity bugs as "bugs".

upstream packages take 52% and 49% more time to be fixed locally in Debian and Fedora, respectively, compared to bugs in in-house packages, but 39% less time than upstream fixed bugs in Debian. Moreover, in the bug comment discussions, maintainers mention more links to reference documents of a distribution (e.g., design docs, bug reports) when fixing upstream bugs locally than when fixing bugs in in-house packages in Debian, but fewer links than upstream bugs that are fixed upstream. In terms of the number of participants per day and the number of comments per day, we observe that the cost of fixing upstream bugs, both upstream and locally, is similar to fixing bugs in in-house packages. In addition, distribution maintainers maintain fewer local patches (fixes) per upstream package within one distribution release than developers do for in-house packages.

RQ3: Which high-severity bugs are most likely to be fixed upstream?

Since fixing upstream bugs locally requires a longer time than fixing in-house bugs, this RQ investigates the factors that are associated with a bug being fixed by upstream developers, by leveraging logistic regression models. In particular, we extract 27 features in 6 dimensions to estimate the association between these features and the likelihood of a bug being fixed upstream. Our models achieve a median AUC of 0.87 and 0.73 for Fedora and Debian, respectively. For the performance across the studied distributions, the model trained on Fedora achieves an AUC of 0.63 for Debian and the model trained on Debian achieves an AUC of 0.81 for Fedora. Our results show that the features related to bug history (e.g., reported bugs in upstream) are correlated the most with a bug being fixed upstream. Moreover, the more similar to a previously reported upstream bug, the

more likely a bug is fixed upstream.

Since only the minority of upstream bugs are explicitly recorded to be fixed upstream or locally, distributions and upstream projects should establish a process to explicitly track the fixing process of bugs in upstream packages. In addition, even though the cost of fixing a bug in upstream packages locally at the fine-grained level (i.e., in terms of the number of participants per day and comments per day) is similar to in-house bugs, users wait a longer time to get fixes and maintainers need more information for fixing upstream bugs. Based on our models, the features in the bug history dimension have a relatively large explanatory power regarding the likelihood of a bug being fixed upstream. We suggest that maintainers should check upstream projects or peer-distributions for similar bug reports, especially those with potential fixes, beyond the complex relation between upstream projects and distributions that carry them.

The chapter is organized as follows. Section 3.2 provides the background of upstream package management in Linux and prior related work to our study. Section 3.3 describes how we design our study and obtain our studied dataset. Section 3.4 answers our research questions. Section 3.5 discusses the implications of our study. Section 3.6 discusses the threats to the validity of our study. Finally, Section 3.7 concludes the work.

3.2 Background and Related Work

In this section, we briefly introduce upstream package management in Linux distributions and how to fix bugs in upstream packages. We also discuss prior studies that are related to our study. We discuss related work on empirical studies of (1) upstream and

downstream coordination and (2) bug fixing cost.

3.2.1 Upstream Package Management in Linux Distributions

A Linux distribution contains the kernel, a set of packages, and a package management system. The package manager facilitates the installation of software from scratch by searching the available packages in specific repositories, while ensuring that specified version constraints are followed. As shown in Figure 3.1, upstream developers are members of upstream projects in charge of developing those projects. The integration (and customization) of such upstream projects into upstream packages is done by (distribution) maintainers. While the role of "maintainer" is specific to a distribution, the person having that role could be part of the upstream project (e.g., Fedora) or external to it (e.g., Debian). Finally, distributions also feature in-house packages developed by (in-house) developers.

Given that distributions distribute both upstream and in-house packages to their user base, they also play a major role in the management of bugs reported for those packages. While in-house package bugs obviously are the responsibility of distributions themselves, upstream bugs should be reported to and resolved by the upstream projects in order to propagate the fixes to downstream distributions. Freedesktop.org, a website that offers interoperability specifications on the software development management (e.g., packager policy, naming conflicts between files in two upstream packages) for Linux distributions, offers an official guideline on how to manage upstream package bugs in Linux distributions[3], encouraging developers not to deviate from the software development practices of the upstream projects. The

[3] https://www.freedesktop.org/wiki/Distributions/Packaging/WhyUpstream/

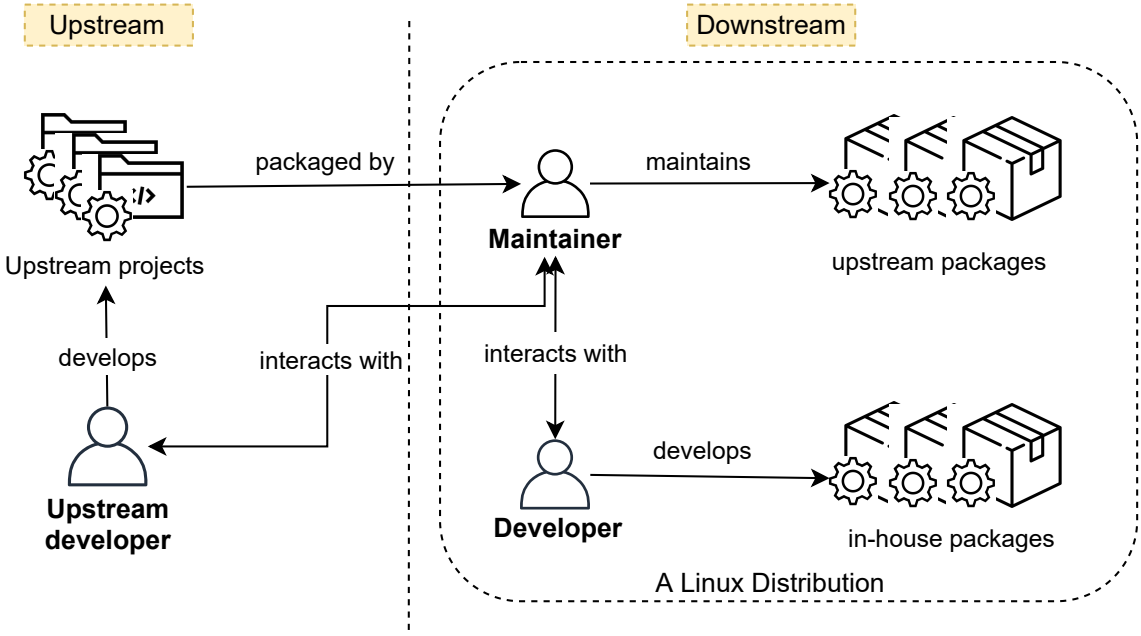


Figure 3.1: Simplified illustration of the three major roles involved in the maintenance of upstream packages in Linux distributions. The maintainer role takes the responsibility of fixing bugs in upstream packages, interacting with the upstream developers, and integrating upstream fixes into a distribution. While the "maintainer" role is distribution-specific, the person having that role in a distribution could be part of the upstream project or external to it. The developer role develops in-house packages and addresses bugs in these packages.

guideline highlights the benefits and shortcomings of upstream package management, such as the benefit of incremental feature improvements, and the shortcoming that upstream projects can be unresponsive.

However, in practice, some upstream bugs are resolved by distribution maintainers. This could indicate an important challenge as maintainers often lack the expertise of upstream package developers to investigate the root causes of those bugs (Ma et al., 2017). Distribution maintainers can feel compelled to provide quick workarounds for the upstream bugs in the form of local bug fixes (Ding et al., 2017). While providing

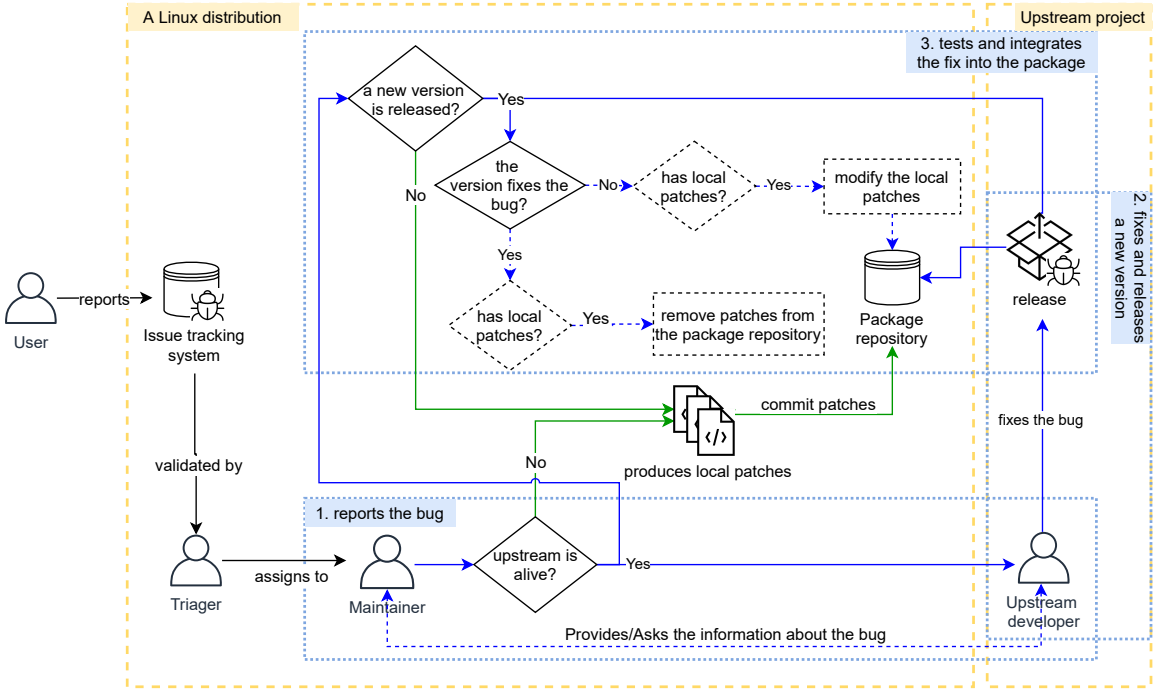


Figure 3.2: An illustration of the bug fixing process for upstream packages. The three large blue boxes with the blue-background text indicate the three stages in which a maintainer in a distribution, according to the freedesktop.org guidelines, collaborates with upstream developers to fix a bug in their package. The blue arrow lines indicate the common steps. The green arrow lines represent the alternative reaction of a maintainer (e.g., upstream is discontinued or severe bugs). The dashed arrow lines and boxes represent optional steps. The yellow dashed boxes represent the instances/roles/actions inside a Linux distribution or an upstream project.

quick workarounds can avoid delays due to communication with the upstream, it conflicts with the official guideline on freedesktop.org.[4] Therefore, in our study, we are interested in how maintainers manage upstream packages in practice.

Figure 3.2 indicates the typical bug fixing process for upstream packages. We derive the process from the official documents in Debian (Debian.org, 2021d) and Fedora (Fedora.org, 2021c) by an open-coding approach. The 1st author derived the initial version

[4] https://www.freedesktop.org/wiki/Distributions/Packaging/WhyUpstream/

of the process, discussed this version with the 2nd and 3rd authors, then iteratively resolved any disagreements. A triager validates an upstream bug report and assigns it to the maintainer of the upstream package after the upstream bug was reported by a user. Unlike fixing bugs in in-house packages, fixing bugs in upstream packages includes additional stages: (1) the maintainer reports the bug to upstream developers if they are accessible, (2) the upstream developers fix the bug and release a new version, and (3) the maintainer tests the fix and integrates it into the package repository of her/his distribution.

In practice, however, the process is more complex. If the upstream project is discontinued or not responsive, maintainers need to fix the bug themselves by producing *local patches* (Adams et al., 2016). Furthermore, upstream developers might take a long time to fix the bug, while maintainers also produce a local patch to reduce the impact of the bug. When maintainers produce the patches to fix a bug, they need to maintain the patches until the upstream project ships a release incorporating the fix. In other words, maintainers have the additional patch maintenance effort in keeping the patches compatible with the subsequent distribution releases until the bug is fixed upstream.

3.2.2 Related Work on Upstream and Downstream (Distribution) Coordination

Prior studies investigated the potential problems and concerns regarding the coordination between upstream and downstream developers to fix bugs in upstream projects. For instance, Canfora et al. (2011) identified the prevalence of bug fixes between the FreeBSD and OpenBSD kernels. To fix bugs that occur between projects, developers in

the two projects must communicate with each other to share the information of bugs. Ma et al. (2017) identified the difficulty for downstream developers to resolve upstream bugs in the scientific Python ecosystem, and observed that workarounds (i.e., temporary and local solutions) are common practices to reduce the long-lasting impact of upstream bugs. Our study investigates the cost of fixing bugs in upstream packages in terms of the number of developers, time, and local patches.

Ding et al. (2017) summarized four common workaround patterns that are used in practice to, on the one hand, require as few code changes as possible while, at the same time, being easy to remove such code changes afterwards. For example, replacing the buggy upstream method by using a similar method in the downstream project. We focus on the amount of effort for downstream developers to produce local patches to fix upstream bugs.

Zhang et al. (2018) observed that upstream bugs are often linked with longer developer discussions in the Ruby on Rails ecosystem. We also aim to understand the communication between upstream and distribution (downstream) developers by leveraging machine learning models to explain how collaborative activities affect the fixing of bugs in upstream packages. Ma et al. (2020b) studied the impact of upstream bugs, such as buggy code hidden in downstream projects until the failure inducing preconditions are satisfied. They proposed a technique to identify such impact in the downstream projects. They identified 1,132 downstream modules using 31 buggy upstream functions but only 47 out of these 1,132 modules have ever reported the buggy functionality in the past.

Another line of research related to upstream and downstream coordination is to

port functions or code snippets in software projects from one to another. Porting software projects is close to our study since it is similar to the 3rd stage in Figure 3.2, i.e., integrating and testing upstream fixes into a distribution. For instance, Ray and Kim (2012) studied the extent and characteristics of duplicate work happening in forked projects when cross-system porting from three BSD projects. They observed that porting happens periodically (at least 10 months) and the rate of porting does not reduce over time. At least 26% of active developers are involved in porting patches from other projects in the three BSD projects. Ray et al. (2013) characterized 5 categories (e.g., inconsistent variable renaming, porting code in a wrong operation) of porting errors from 113 and 182 porting errors in FreeBSD and the Linux kernel, respectively. Barr et al. (2015) developed an algorithm using genetic programming to transplant features in one software project into another automatically. The authors applied the algorithm to extract a feature from the donor project (source) and transplant it into the host project (target) through several experiments, including the VLC media player. However, our study focuses on the entire bug fixing process, including the coordination between upstream and distributions to fix high-severity bugs in upstream projects.

Different from the aforementioned studies, we study the prevalence of bugs in upstream packages within Linux distributions, which contain thousands of packages to better understand how maintainers manage upstream packages.

3.2.3 Related Work on Bug Fixing

Bug fixing is a complex, iterative, and costly process in distributed software development (Crowston and Scozzi, 2008). Prior work investigated the amount of needed time

and developers to fix bugs. Kim and Whitehead Jr (2006) observed that it takes a median of 200 days to resolve a bug in the ArgoUML and PostgreSQL projects. Duc et al. (2011) showed that, in three firm-involved open-source projects, firm-paid developers contribute higher to fix bugs and have a higher workload on average compared to volunteer developers.

Another popular line of work in this area is the analysis of the influential factors in the bug-fixing process. Weiss et al. (2007) observed that similar bugs take similar effort for developers to fix them in the JBoss project. Guo et al. (2010) studied two Microsoft products and found that the reputation of bug reporters affects the bug-fixing time. However, in five open-source projects, Bhattacharya and Neamtiu (2011) showed that the reputation of bug reporters has little influence on the bug-fixing time. Zhang et al. (2013) discovered that bugs with higher severity and priority are fixed faster, but the differences of bug fixing time between each of these bugs are large in three industrial projects.

The aforementioned studies extracted bug reports from a project and studied the bug fixing activities within the project, while our work extracts bug reports within a software ecosystem (i.e., Linux distribution). In other words, in our case, bugs exist in in-house and/or upstream projects in the Linux distribution. We also compare the cost of fixing bugs for maintainers and developers between upstream and in-house projects. To the best of our knowledge, no prior studies have examined the time for maintainers/developers to fix upstream bugs and investigated the corresponding cost.

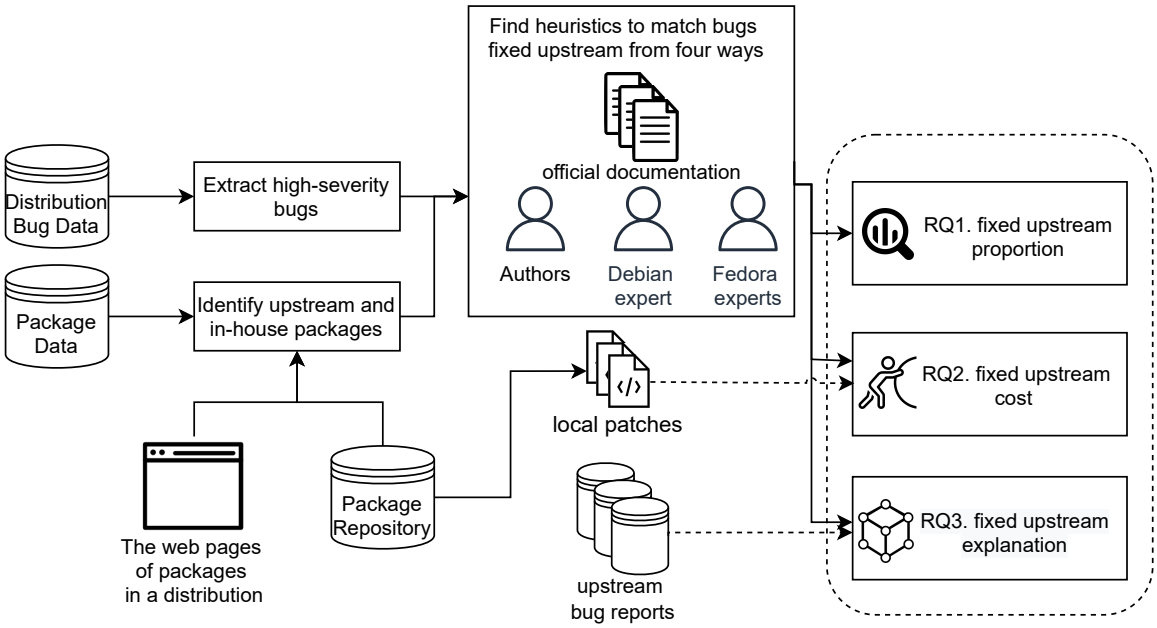


Figure 3.3: An overview of our study approach. The solid arrow lines indicate the data sources used for all RQs. The dashed arrow lines indicate the particular data sources used for each specific RQ. We will describe the data sources in more details in the corresponding sections (RQ2 and RQ3).

3.3 Study Design

This section presents the design of our empirical study addressing the three research questions that we mentioned in the introduction. RQ1 studies the proportion and evolution of high-severity bugs in upstream and in-house packages that are fixed. RQ2 investigates the maintenance cost for upstream bugs that are fixed by distributions. Finally, to understand why bugs are fixed upstream, RQ3 examines the influential factors for a high-severity bug to be fixed upstream. Note that the specific methodology for each RQ is explained in Section 3.4.

3.3.1 Selection of Subject Systems

We select two Linux distributions, i.e., Fedora and Debian, based on their popularity and the ease of access to links between bugs and upstream for these two distributions. They leverage two of the most popular package managers in Linux. Fedora leverages *rpm* as its package manager, and Debian uses the *dpkg* package manager. According to the 2020 popularity ranking on distrowatch.com[5], Debian (6th) and Fedora (8th) are the two highest ranked root distributions, i.e., the source distributions from which derivative distributions (like Ubuntu, Mint, etc.) are forked to develop their own features independently. The upstream packages in a derivative distribution are either the packages that are inherited from their parent distribution or those that are packaged directly from external projects (i.e., the source projects where their parent packaged from). The derivative distribution gets the fix of an upstream bug either from their parent or the external project. As the derivative distribution may switch the fix sources due to certain reasons (e.g., critical bugs), and there is no data recording where the source of a fix for an upstream bug is in a derivative distribution, we exclude out the derivative distributions.

In addition, Debian and Fedora follow different strategies of upstream bug management. Fedora is the upstream distribution of the commercial Red Hat distribution, which is its primary sponsor[6]. Therefore, Fedora follows the "upstream first" policy[7] from Red Hat to do software development (e.g., forwarding bugs to upstream, fixing bugs) side-by-side with upstream developers in the upstream projects. Only in specific cases (e.g., unresponsive upstream or release schedules), Fedora will fix bugs directly in

[5] https://distrowatch.com/dwres.php?resource=popularity
[6] https://getfedora.org/en/sponsors/
[7] https://www.redhat.com/en/blog/what-open-source-upstream

Table 3.1: Basic statistics about the studied Linux distributions from January, 2005 to the end of 2019.

Distribution	ITS	# high-severity bugs	# upstream packages[3]	# in-house packages[3]
Fedora	Bugzilla[1]	37,205	35,139 (76%)	11,332 (24%)
Debian	Debbugs[2]	106,157	33,044 (86%)	5,366 (14%)

[1] https://bugzilla.redhat.com/
[2] https://www.debian.org/Bugs/
[3] The percentage is the # in the cell divided by the sum of the # of upstream and in-house packages.

the distribution and try to get patches accepted upstream to avoid future maintenance on those patches. By contrast, Debian depends on maintainers in its own community to maintain local patch repositories for packaged upstream packages, sending patches upstream whenever possible.

3.3.2 Data Collection

Figure 3.3 presents an overview of our study approach. We leverage two data sources for each of the studied distributions, i.e., the bug reports and the packages. Our studied period is from 2005 to 2019 to exclude the insufficient data of the early stage in each distribution and the latest data of 2020 (time of starting our study). The first release of Fedora was created from Red Hat in 2003[8], while the first release of Debian was made in 1996[9]. We archived our scripts and uploaded them to GitHub[10]

[8] https://fedoraproject.org/wiki/Releases/HistoricalSchedules
[9] https://wiki.debian.org/DebianReleases
[10] https://github.com/SAILResearch/suppmaterial-20-justina-upstream_bug_linux

Extract High-severity Bugs

To obtain our dataset, we first extract the bug reports from the corresponding issue tracker of each studied distribution, as shown in Table 3.1, by either the relevant APIs that are offered by the issue trackers or custom crawlers. We only consider the high-severity (e.g., high, urgent, critical) bugs because these bugs have a higher impact on users and are hence more important to fix (Menzies and Marcus, 2008; Lamkanfi et al., 2010). We also identify duplicate bugs, similar to prior work (Bettenburg et al., 2008; Boisselle and Adams, 2015), and remove them to avoid over-estimating the number of bugs and analyses in our work. Our dataset contains a total of 143,362 high-severity bugs across the studied distributions. The details of each distribution are presented in Table 3.1.

Note that even though an issue tracking system can also be used for feature requests, our dataset does not include the latter since Linux distributions apply a low-severity label or use other platforms for feature requests. In particular, Debian uses the *wishlist* value, which is a low-severity label, to indicate that a bug report is a feature request.[11] Fedora does not track feature requests in its issue tracking system.[12]

Identify Upstream and In-house Packages

We then extract the list of packages and their relevant information (e.g., name and category) for each release of a given distribution. We also classify packages into two groups based on the sources of packages: upstream (obtained directly from an upstream project) or in-house (developed by in-house teams).

For each package in a given distribution, we leverage the relevant web pages within

[11] https://www.debian.org/Bugs/Developer
[12] https://docs.fedoraproject.org/en-US/fesco/

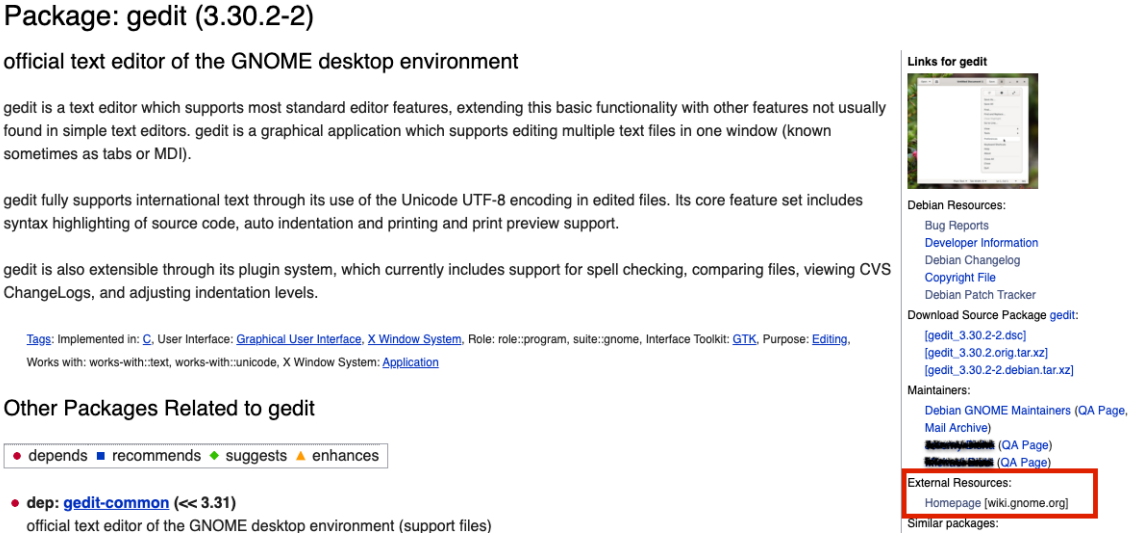


Figure 3.4: An example of how to determine an upstream package. The *gedit* package is an upstream package since the red box indicates the source project, i.e., GNOME.

the distribution or package repositories, when available, to identify whether there is an upstream project link. For example, Figure 3.4 shows that *gedit* is an upstream package in Debian[13]. We identify 35,139 and 33,044 upstream packages in Fedora and Debian, respectively. After identifying upstream packages, we filter out 13,996 and 33,230 of the remaining packages in Fedora and Debian, respectively, for which we cannot find available web pages or repositories. We then assign the remaining packages as in-house packages.

To evaluate the precision of the above heuristic method that we use to determine upstream and in-house packages, we manually investigate 50 packages by randomly selecting 25 upstream packages and 25 in-house packages. For the 25 sample upstream packages, we leverage the upstream project link in each package to check whether the upstream developer team is active or not. We observe that all the upstream projects are available at the time when we write this chapter, which indicates no false positives

[13] https://packages.debian.org/buster/gedit

in our sample set.

For the 25 sample in-house packages, we search their package names to investigate whether there is an active upstream project somewhere on the Internet (e.g., GitHub), in which case our heuristic would not be correct. For each package, we investigate the metadata (e.g., description, contributors, folder structure) of the potential upstream projects carefully to identify the actual upstream project the package could belong to. After that, we determine that an upstream project is alive by its activities (e.g., commits, comments). We observe that 2 out of the 25 sample in-house packages were originally packaged from upstream projects. One of these two upstream projects is discontinued and thus distribution maintainers took ownership of that package, becoming an in-house package. The other one was customized by distribution maintainers, indicating it is an in-house package. Hence, our analysis of 50 packages indicates that the heuristic method achieves an accuracy of 100%.

3.4 Results

RQ1: What percentage of upstream bugs are fixed upstream versus locally?

Motivation: Packages in a Linux distribution are either developed by in-house developers or packaged from the upstream projects, e.g., GNOME, Firefox, and the Linux kernel. In an ideal world, upstream developers provide new features and fix bugs, while distribution maintainers integrate new package versions into their distribution, making them interact seamlessly with the other packaged projects. Hence, the goal of this research question is to characterize the bugs from upstream packages. In particular,

we investigate the evolution of reported and fixed high-severity bugs over time and the proportion of high-severity bugs that are fixed upstream.

Approach: To study high-severity bugs that are fixed upstream, we analyze the proportion of these bugs that are fixed upstream and those that are fixed locally in the two studied distributions, the evolution of reported and fixed bug reports in a distribution over time, and which application domains of packages are the most buggy. We compare the above analysis between the studied distributions.

Unfortunately, there is no unique way to identify whether a bug reported to a distribution has been fixed upstream or locally. For that reason, we have leveraged several heuristics, combined with manual validation of the heuristics by the authors, an actual Debian maintainer and three Fedora maintainers. Our interviewed maintainers are experienced individuals. We conducted the interview by asking four maintainers to avoid potential bias. More specifically, the Debian maintainer has been working in Debian for almost 10 years and currently maintains at least 35 packages. One of the three Fedora maintainers has been working in Fedora since its first release (i.e., 2007) and keeps contributing to upstream projects. This maintainer is an expert in several areas of package maintenance, having contributed to and being involved in the maintenance of dozens of packages. The second Fedora maintainer worked at Fedora for more than 6 years and is an expert in Python and auto deployment. The third Fedora maintainer has worked at Fedora for 4 years, specializing in the synchronization of package releases between upstream and Fedora.

Heuristics for identifying bugs that are fixed upstream

We contacted the maintainers to figure out the potential sources that record the information of bugs fixed upstream. For example, the Debian maintainer suggested that *"the Debian BTS has features to track whether or not a bug is in the upstream portion of a package, and whether or not it is fixed upstream. See, in particular, the 'upstream' and 'fixed-upstream' tags on bugs."* and "*to see if [a bug] was fixed by a patch in the packaging.".* One of the Fedora maintainers suggested that *"BZ [numbers] help to track the fixed changes [in the changelog]."* Eventually, we settled on leveraging the tags in bug reports, the changelogs of packages, and the metadata of patches.

First, we leveraged the "fixed-upstream" tag of bug reports to identify a set of bugs that are fixed upstream in Debian (Debian.org, 2020c), while in Fedora we look for bugs with the "UPSTREAM" label (Fedora.org, 2020b). Using these heuristics, we identified 3,726 and 556 fixed-upstream bugs in Debian and Fedora, respectively. Since the "UPSTREAM" label in Fedora indicates that a bug has been reported to upstream and it will be integrated back into Fedora after the upstream fixes it, we manually investigate whether bugs with the label were fixed upstream on a representative sample set of 227 fixed-upstream bugs (confidence level = 95%, confidence interval = 5%). We observe that 208 (92%) out of the 227 bugs were fixed upstream and 19 are unknown. The upstream bug reports of 10 bugs are not available, 4 bugs were reported to upstream developers by email, and 5 bugs do not contain the links to the respective upstream bug reports. Hence, we conclude that the heuristic has a high accuracy (at least 92%) to identify fixed-upstream bugs.

Second, we leverage the official guidelines maintainers should follow to compose

the changelogs of a new package version. Specifically, Debian states that *"It is conventional that the changelog entry of a package that contains a new upstream version of the software looks like this: *New upstream release"* (Debian.org, 2020d) and adds the fixed bugs following a certain format (i.e., #id) (Debian.org, 2020d). Fedora also has a similar policy for maintainers to record updates to a new version and fixed bugs in a changelog (Fedora.org, 2020d).

Based on these documented guidelines to identify fixed-upstream bugs, we inspect the changelogs (i.e., a total of 918 changelog entries) from a total of 50 randomly selected upstream packages (i.e., 25 packages in Debian and 25 packages in Fedora) to find potential patterns. We also randomly investigate the changelogs of the top 5 buggy upstream packages in both studied distributions (i.e., a total of 10 packages) to figure out additional patterns.

Finally, we derive the following regular expression:
`"new upstream [\w ]*(?=release|version|fix)"` in Debian to find references to bugs fixed upstream, while in Fedora we would look for the keyword "upstream". Bug #194851 in Debian (Figure 3.5a) and Bug #742349 in Fedora (Figure 3.5b) illustrate these heuristics. Note that the bug-id in a changelog entry might indicate a low-/medium-severity bug or the reference bug-id to the upstream project (that may not be reported to the distribution) for traceability, which we filter out afterward from our dataset. Based on the heuristic, we identified a total of 5,180 (4,007 additional) fixed-upstream bugs in Debian and 68 additional fixed-upstream bugs in Fedora.

To validate this second heuristic, we manually investigate the comments of 75 randomly selected bugs from three popular packages (i.e., perl, apache2, and samba) in Debian. We observe that 20 out of the 75 bugs that are marked as fixed-upstream bugs

```
6tunnel (0.10rc2-1) unstable; urgency=high

  * New upstream release (closes: #194851)
  * Bumped Standards-Version to 3.5.10
```

(a) An example of the changelogs for bug 194851 in Debian

```
* Tue Nov  1 2011 Tim Waugh <twaugh@redhat.com> 9.04-5
- Applied upstream fix for skipping "cached" outline glyphs (bug #742349).
```

(b) An example of the changelogs for bug 742349 in Fedora

Figure 3.5: Two examples of bug fixes in the changelogs of upstream bug fixes in a package release in Debian and Fedora.

by the second heuristic are the bugs that are reported to be fixed upstream, which are true positives. However, we found 3 out of the 75 selected bugs that are not marked as fixed-upstream bugs that should be classified into fixed-upstream bugs, suggesting a high precision ($72/75 = 96\%$) and recall ($20/23 = 87\%$) of the heuristic. The changelogs of these 3 bugs do not match any pattern in the heuristic. Moreover, we also note that the participants in 4 out of the 75 selected bugs were not able to determine whether the upstream fixed them or not. For instance, for those bugs no developer tested the fix of a bug or there were no comments for a long time (e.g., a few months or years) before the bug was closed and marked as fixed. Since we cannot classify bugs for which distribution maintainers themselves are not able to track the source of the fix, we do not consider these to be misclassifications of our heuristics.

On the other hand, since the heuristic only identifies a small number (i.e., 68) of fixed-upstream bugs in Fedora, we investigate whether maintainers follow the guidelines in practice. Although we observe 808 changelog entries containing bug-ids using the specific format, only 68 of them also contain the keyword “upstream”. Furthermore, out of the 808 changelog entries in our dataset, only 121 refer to high-severity bugs.

Table 3.2: Statistics of high-severity **upstream** bugs in each distribution.

Distribution	# reported upstream bugs[1]	# fixed bugs[2]	# fixed upstream bugs[2]	# fixed locally bugs[3]
Fedora	32,530 (87.4%)	12,009 (36.9%)	624 (5.1%)	529 (4.4%)
Debian	88,300 (83.2%)	55,481 (62.8%)	7,733 (13.9%)	7,386 (13.3%)

[1] The percentage presents the percentage out of all high-severity bugs.
[2] The percentage is computed relative to the cell on its left.
[3] The percentage is computed relative to the # of fixed bugs.

Third, we also leverage the metadata in patches where maintainers tag the source (e.g., upstream, Debian) of a patch to identify fixed-upstream bugs. Debian suggests their maintainers follow the DEP-3 tagging guideline[14] and to use the *Origin* label to indicate the origin of the patch in the header. However, we observe that only 9.3% of patches in Debian used the DEP-3 format, in which we identify a total of 652 (133 additional) fixed-upstream bugs in Debian. Unfortunately, we cannot find a similar guideline in Fedora. Eventually, we obtain a total of 7,733 (13.9%) and 624 (5.1%) fixed-upstream bugs in Debian and Fedora, respectively (see Table 3.2).

Heuristics for identifying bugs that are fixed locally

To identify bugs that are fixed locally, we derive heuristics based on bug reports and the changelogs of packages. First, we select all distribution bug reports that contain attached patch files, i.e., the attachments with one of the following extensions: `.debdiff`, `.diff`, or `.patch` in Debian. In Fedora, we leverage a similar heuristic based on the structured field (i.e., `is_patch`) of an attachment in Bugzilla. Using these heuristics, we identified 7,386 and 416 bugs that are fixed locally in Debian and

[14] https://dep-team.pages.debian.net/deps/dep3/

Fedora, respectively. Second, we select changelog entries that contain at least one of the following keywords: work-around, workaround, temporar*, or patch and that do not contain the "upstream" keyword in Fedora. We identified 113 additional bugs that are fixed locally. Eventually, we obtain a total of 7,386 and 529 bugs that are fixed locally in Debian and Fedora, respectively (see Table 3.2).

We validate these two heuristics by manual investigation on the set of identified bugs. For the first heuristic, we investigate the comments of 25 randomly selected bugs in Debian and 25 randomly selected bugs in Fedora. We observe that 1 out of the 50 selected bugs was fixed upstream instead of locally, suggesting a high accuracy of 98% (49/50). Similar to the aforementioned cases of fixed-upstream bugs, for 2 out of the 50 selected bugs we were not able to determine whether distribution maintainers fixed it or not. For the 2nd heuristic, we investigate the comments of 25 randomly selected bugs in Fedora and observe that in 1 out of the 25 bugs maintainers were not able to determine who fixed it. Due to the lack of precise information in the bug reports, we do not consider such cases to be misclassifications of our heuristic.

Bug Analysis

To analyze the evolution of the reported bugs in in-house or upstream packages by year, we calculate the number of reported bugs for either group of packages divided by the total number of reported bugs (i.e., both in-house and upstream packages) in each year. To avoid the impact of the number of packages, we normalize the proportion of reported bugs by the number of active packages (that have identified bugs by our aforementioned heuristics) in each year. For example, when distribution maintainers integrate new upstream packages into their system, the number of reported

bugs in upstream packages would exhibit a sudden spike, then drop afterwards. On the other hand, the actual count of reported bugs drops suddenly when maintainers remove packages along with their dependent ones.

Therefore, we calculate the normalized count of bugs (NCB) using the following equation to characterize the evolution of the proportion of bugs reported to maintainers per package and compare the evolution between in-house and upstream packages.

$$\text{NCB}_{\text{report}}(y) = \frac{\text{Bug Count}_{\text{report}}(y)}{\text{Bug Count}_{\text{reportall}}(y) * \text{Package Count}(y)} \tag{3.1}$$

where:

y = year of interest

$\text{Bug Count}_{\text{report}}$ = the number of bugs reported (in-house or upstream) in year y

$\text{Bug Count}_{\text{reportall}}$ = the number of bugs in year y

Package Count = the number of active packages in year y

We also compare the evolution of upstream bugs that are recorded to be fixed by in-house teams and those that are recorded to be fixed by upstream developers. Similar to the reported bugs, we calculate the normalized count of fixed bugs (NCFB) using the following equation to capture the evolution of fixed bugs at the package level.

$$\text{NCFB}_{\text{fixed}}(y) = \frac{\text{Bug Count}_{\text{fixed}}(y)}{\text{Bug Count}_{\text{fixedall}}(y) * \text{Package Count}(y)} \tag{3.2}$$

where:

y = year of interest

$\text{Bug Count}_{\text{fixed}}$ = the number of upstream bugs fixed (locally or upstream) in year y

$\text{Bug Count}_{\text{fixedall}}$ = the number of upstream bugs in year y

Package Count = the number of active packages in year y

Finally, we identify the categories of packages that more commonly contain high-severity bugs. Since Debian has a list of 59 package categories (Debian.org, 2020a) and Fedora does not, we determine the package category for the upstream packages in Fedora using the following steps:

- Assign the package category of a package in Fedora when the name of the package matches exactly the name of a package in Debian.
- Assign the package category of a package in Fedora when the upstream source project of the package matches exactly that of a package in Debian.
- Based on our observations, we derive several heuristic rules from the naming convention of packages in Debian. For example, the names of packages in the localization package category contain the keyword "i18n" or "l10n". The names of packages related to the KDE desktop start with the prefix "kde-". We apply such heuristics to identify the package categories. The full list of our heuristics can be found in Appendix A.

Results: 13.9% and 5.1% of upstream bugs in the studied distributions are explicitly recorded to be fixed upstream in Debian and Fedora, respectively, and 13.3% and 4.4% of upstream bugs are recorded to be fixed locally. The number of upstream bugs

that are fixed upstream and those that are fixed locally are similar in both studied distributions (Fedora: 624 vs. 529, Debian: 7,733 vs. 7,386), while the percentages for Debian (13.9% and 13.3%) are much higher than for Fedora (5.1% and 4.4%).

As a corollary of our findings, surprisingly, for 73% and 90% of upstream bugs in Debian and Fedora, respectively, we are not able to automatically determine whether upstream developers or distribution maintainers produced the fixes. While distribution maintainers usually have this information in their minds, the fact that it is not explicitly recorded in bug reports or changelogs could complicate upstream bug management in the long run. Maintainers are not able to link several changes made for a particular bug. Furthermore, in the context of the software supply chain, missing such information exacerbates the risks of resolving security issues. For example, Heartbleed (i.e., CVE-2014-0160) was a severe security bug in the OpenSSL package. Debian maintainers cherry-picked a commit[15] made by two external contributors into their OpenSSL package repository on April 5, 2014, without the corresponding bug id. The commit directly changed source code in files (e.g. ssl/d1_both.c) instead of being recorded as explicit patches. Bug[16] #743883 (for Heartbleed) in Debian's issue tracker was closed on April 9. Shortly afterwards, Debian maintainers incorporated a new upstream release[17] on July 23, 2014, again without reference to the bug id. Since there is no corresponding bug id in these two commits, the only way for Debian maintainers to link both patches (direct fix and new upstream release) to bug #743883 would be searching for the common CVE-ID (CVE-2014-0160).

Note that in our analysis a bug that is fixed upstream refers to a bug that was fixed

[15] https://salsa.debian.org/debian/openssl/-/commit/96db9023b881d7cd9f379b0c\154650d6c108e9a3

[16] https://bugs.debian.org/cgi-bin/bugreport.cgi?bug=743883

[17] https://salsa.debian.org/debian/openssl/-/commit/2f21d2da895a465e151f4cb8\f100040b897b5c1c

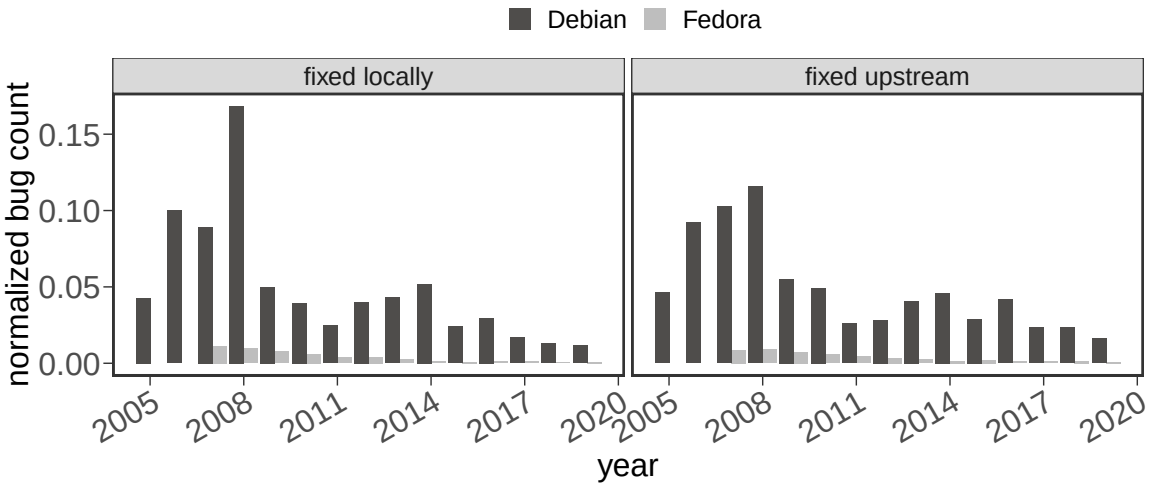


Figure 3.6: The normalized counts of upstream bugs (relative to package count) **that are recorded to be fixed** locally (left) and upstream (right) have dropped in Fedora over time. Both trends continue to fluctuate in Debian.

upstream first, before propagating the fix to the distributions, while a locally fixed bug refers to a bug that first was fixed locally in a distribution, before potentially propagating the fix upstream. The propagation process of an upstream fixed bug is automatic (as soon as the next release is sent to the package maintainers, the distribution will have access to the fix), while the propagation process of a locally fixed bug from a distribution to an upstream project is not guaranteed (e.g., in the case of distribution-specific modifications).

The normalized count of upstream bugs (NCFB) that are recorded to be fixed upstream has dropped over the years in Fedora, while the trend has been fluctuating in Debian. The proportion of bugs reported to be fixed upstream per year has been steady in both Fedora and Debian. For each year, a median of 5% and 14% of bugs across upstream packages are recorded to be fixed upstream with a standard deviation of 1.9 and 2.4 in Fedora and Debian, respectively.

Given this stable trend of the proportion of bugs that are fixed upstream, we further

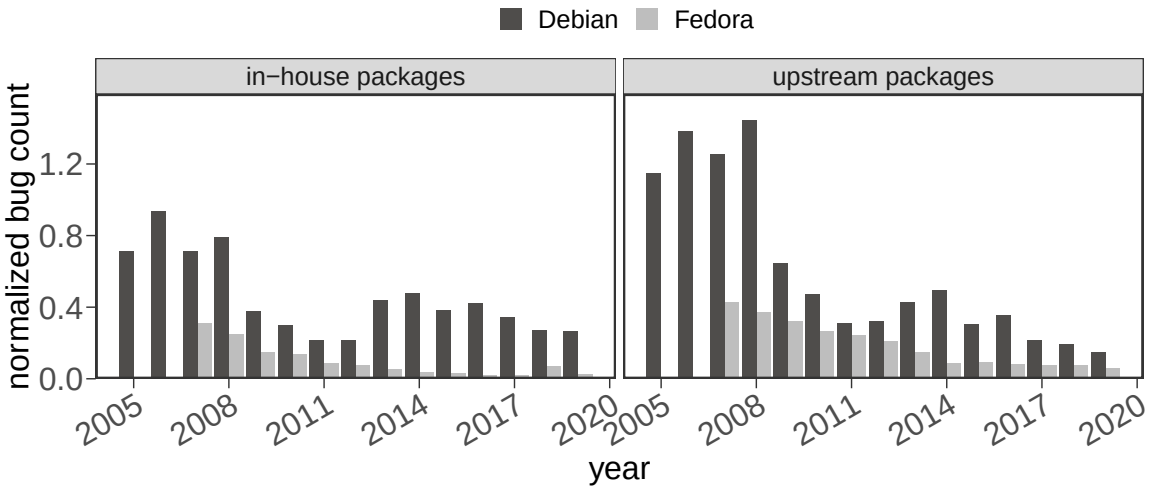


Figure 3.7: Comparison between the evolution of the normalized count of **reported bugs** normalized by the package count for both in-house and upstream packages.

analyze the trend at the package level using our normalized bug count (NCFB). Figure 3.6 shows the evolution of the normalized count of upstream bugs that are fixed upstream and locally over time, respectively. In both studied Linux distributions, these trends are similar since the proportion of bugs that are fixed upstream has been steady over time. However, Fedora has fewer bugs per upstream package that are fixed, compared to Debian, for both upstream and locally fixed bugs.

The normalized count of reported bugs (NCB) in upstream packages has also dropped over the years, with fluctuations in the studied distributions. Figure 3.7 shows the evolution of the normalized count of reported bugs in upstream and in-house packages across the studied distributions. The trend of the reported bugs in upstream packages (the right plot in Figure 3.7) is similar to the corresponding trends for fixed bugs in Figure 3.6, indicating that the proportion of the fixed bugs in upstream packages is consistent over the years. For example, from 2011 to 2014, both the trends of bugs that are fixed upstream and locally continue to increase in Debian (Figure 3.6),

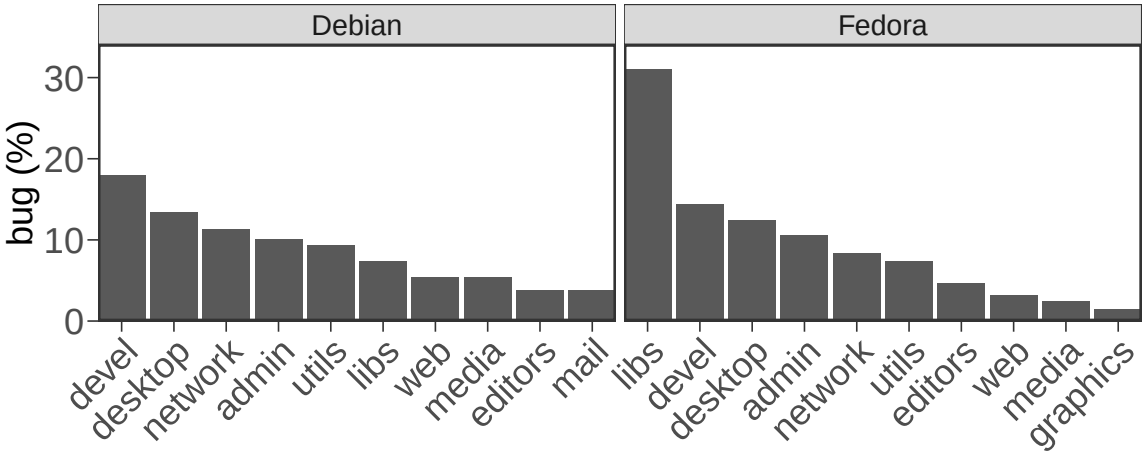


Figure 3.8: The distribution of top 10 buggy categories of upstream packages. Fedora and Debian share 9 of 10 common buggy categories with different ranking orders. The categories that do not match are the **mail** category in Debian and the **graphics** category in Fedora.

which coincided with a similar trend for reported bugs in upstream packages (the right plot in Figure 3.7). We also observe that the NCFB (Figure 3.6) and NCB (Figure 3.7) in Debian are larger than in Fedora. One possible reason for this is that they follow different release cycles of approximately 3 years[18] vs. 13 months[19], respectively. As such, the period for users to report bugs to a particular version of a package in a Debian release is longer than in a Fedora release.

Debian and Fedora have a similar catalog of buggy upstream packages. Figure 3.8 shows that 9 out of the top 10 buggy categories of upstream packages are common in the studied distributions. These 9 common categories of upstream packages account for 84% and 94% of upstream bugs in Debian and Fedora, respectively. However, the

[18] https://www.debian.org/releases/
[19] https://fedoraproject.org/wiki/Fedora_Release_Life_Cycle

ranking of these categories of upstream packages is different between Debian and Fedora. For example, the libs category is the most buggy category in Fedora, while it is the 6th ranked category in Debian.

Since packages in the libs category are common packages that are shared by many packages, Fedora ensures the quality of these packages to reduce the impact on other packages. One of the interviewed Fedora maintainers stressed "*the responsibility of the SRE's (Site Reliability Engineers) to make sure [the bugs in libs] are fixed for Fedora.*" On the other hand, bugs in packages that are not in the libs category are reported to the corresponding upstream projects. Therefore, the fixes are tracked and developed upstream rather than in Fedora.

A minority (at most 14% and 13%) of upstream bugs are explicitly recorded as fixed by upstream developers or distribution maintainers, respectively, whereas for at least 73% of upstream bugs the origin of their fix is not explicitly recorded. The evolution of reported bug count and that of the number of bugs that are reported to be fixed upstream per upstream package has dropped over time in Fedora and Debian. In both studied distributions, the most buggy categories are libs and devel.

RQ2: What is the time and cost for fixing high-severity upstream bugs locally in distributions and maintaining such distribution fixes?

Motivation: The results in RQ1 show that only less than 14% of upstream bugs are explicitly recorded to be fixed upstream. However, distribution maintainers take responsibility for at least 13% of the remaining upstream bugs (with the 73% remaining cases

in Debian not having explicit information w.r.t. who fixed them) even though they are not necessarily familiar with the upstream packages. For example, for maintainers who adopt an upstream project, Ubuntu states that *"You should have some experience with the upstream project - it really helps if you're working on something you know well."*[20] Lacking this experience increases the difficulty to come up with good fixes for the upstream bugs. As a consequence, downstream developers often propose workarounds (e.g., local patches) to reduce the impact of such bugs (Ma et al., 2017).

When a bug is successfully fixed by a distribution maintainer, the fix still needs to be submitted to the upstream project (Freedesktop.org, 2013). However, the fix might only be relevant to the maintainer's own distribution, or it might take time before the upstream project is able to validate the submitted fix. In such a case, distribution maintainers need to hold on to their bug fix changes as separate patches, i.e., diffs between the official upstream release and their modified version. Whenever upstream releases a new version of their project, the maintainer now needs to check if it incorporates his/her bug fix changes and, if not, whether the existing bug fix patch still applies cleanly (Ma et al., 2017). If not, the patch has to be fixed, and re-submitted online for (possible) inclusion in the next project release. Therefore, this is an additional cost for distribution maintainers to maintain the patches and keep them compatible with the following releases until distribution maintainers integrate the upstream releases including the bug fixes.

Considering the above challenges and the large number (i.e., 67,490) of fixed bugs in upstream packages (see Table 3.2), we are interested in gaining a deeper understanding of the involved costs of fixing upstream bugs locally.

[20] https://wiki.ubuntu.com/Upstream/Adopt

Approach: As in Section 3.2.1 (Figure 3.2), the bug fixing process for upstream packages includes collaboration between maintainers and upstream developers. We focus on the bugs that our heuristics in RQ1 are able to identify as fixed upstream or locally to estimate the cost of fixing an upstream bug. We leverage the following metrics that prior studies have widely used to study bug fixing: needed time for users waiting for fixes (Zhang et al., 2013; Kim and Whitehead Jr, 2006; Bhattacharya and Neamtiu, 2011), the number of participants in the bug fixing process (Bhattacharya and Neamtiu, 2011; Canfora et al., 2020), the number of comments (Giger et al., 2010; Zhou et al., 2015; Zhang et al., 2012), the number of reference links (e.g., URLs to design documents or other bug reports) mentioned in comments (Canfora et al., 2020), and the characteristics of local patches (Weimer, 2006; Bhattacharya and Neamtiu, 2011). We then compare the difference of the costs among upstream bugs that are fixed upstream vs. locally, and in-house bugs that are fixed locally.

We compare the needed time for users waiting for fixes among three types of bugs, i.e., bugs in upstream packages that are fixed upstream vs. locally, and bugs in in-house packages. However, fixing upstream bugs includes several different processes in certain circumstances, as shown in Figure 3.2. We cannot measure the time from when upstream developers start fixing a bug to when maintainers know about the fix (although they might not yet have access to it). Therefore, similar to prior work (Zhang et al., 2013; Kim and Whitehead Jr, 2006; Bhattacharya and Neamtiu, 2011), we use the duration between when a bug is opened and when it is marked as fixed in the bug report of a given distribution.

We normalize three of our selected metrics to reduce the correlation between these metrics. In particular, we normalize the number of participants and the number of

comments by the needed time to fix. The number of reference links mentioned in comments is normalized by the number of comments.

Additionally, maintainers produce local patches when these patches (fixes) are not immediately accepted upstream (Adams et al., 2016). Hence, maintainers maintain these patches locally for the subsequent distribution releases until the upstream fixes that particular bug. Local patches can be managed by different tools and stored in dedicated patch repositories. We extract the patches for each package from the corresponding package repository in the studied distributions.

We analyze how many patches are maintained in one distribution release, how long a patch file survives in terms of the number of distribution releases (i.e., a patch is removed when the upstream incorporates the fix), and the code churn of each patch file. Specifically, Debian maintainers use *Quilt* to control the series of patches related to the upstream source (Debian.org, 2021b). In contrast to regular version control systems like git, tools like Quilt do not physically merge patches into a code base, but treat patches as first-class entities. In other words, patches can easily be applied, removed, swapped with other patches, etc. Fedora maintainers leverage the *git diff* command to track the changes of the upstream source for a bug as a patch (Fedora.org, 2021a).

Results: In Debian, locally fixed upstream bugs make users wait for 39% less time than upstream bugs that are fixed upstream, but require users to wait for 52% more time than locally fixed in-house bugs. Figure 3.9 compares the user-waiting time for in-house bugs, upstream bugs that are fixed locally, and upstream bugs that are fixed upstream. In Debian, bugs that originate from upstream packages take longer to be fixed by the upstream developers themselves than being fixed locally. This inefficiency is mostly due to the communication overhead with the upstream projects.

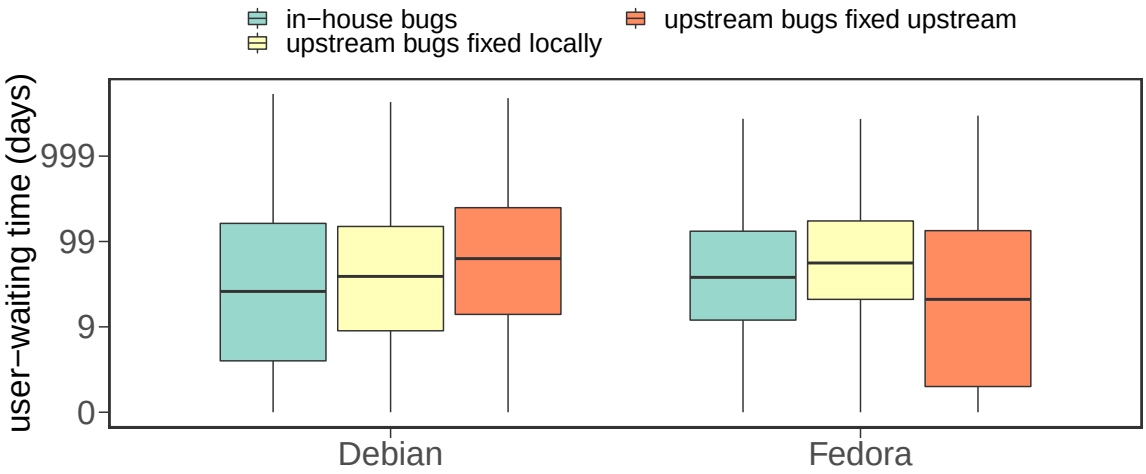


Figure 3.9: Upstream bugs that are fixed locally (the yellow boxplots) take a longer time compared to in-house bugs (the green boxplots) across the studied distributions. In Debian, upstream bugs that are fixed locally (the yellow boxplot) take a shorter time compared to those that are fixed upstream (the orange boxplot).

Since the upstream packages are widely used through Linux distributions (those "distribute" open source projects to end users), bugs from upstream packages typically are reported through them before being propagated upstream (Davies et al., 2010). After reporting to the upstream, maintainers need to wait for the upstream developers to respond to and fix the bug.

More specifically, in Debian, users have to wait a median of 38 days for local fixes of the upstream bugs, while they wait a median of 62 days for fixes that are from upstream and being integrated into Debian. The result of the Wilcoxon signed-rank test shows that the difference between bugs that are fixed upstream and those that are fixed in-house is statistically significant (i.e., p-value $\ll 0.01$, $\alpha = 0.01$), with a small effect size (0.22). On the other hand, users also wait longer to get local fixes for upstream bugs (i.e., a median of 38 days) than in-house bugs (i.e., a median of 25 days), with p-value $\ll 0.01$ ($\alpha = 0.01$).

In Fedora, locally fixed upstream bugs make users wait for 49% and 175% more time than locally fixed in-house bugs and upstream bugs that are fixed upstream, respectively. The reason why users wait a relatively shorter time for upstream bug fixes that are from upstream than local fixes is not that those bugs were actually fixed faster, but because maintainers mark the bug report as fixed once they report the bug to upstream or know that upstream developers have published the fix for the bug. Maintainers do not need to wait for upstream developers releasing a new version and integrating the version into their distribution manually before marking a bug as fixed. Antiya (Fedora.org, 2020a), a service to monitor the upstream releases in Fedora, will automatically integrate the new upstream releases of packages into Fedora (Fedora.org, 2020e). Of course, users would still need to wait the entire time, suggesting that the results for Fedora might be more akin to those in Debian.

For example, a bug report related to taking screenshots when the screen was locked in Fedora, was closed by a maintainer who stated *"Fixed upstream, the 3.14.1 release in next week will bring the fix to Fedora."*[21]. In the corresponding upstream bug report[22], before posting the comment in Fedora, the maintainer posted 13 comments to help resolve the bug, such as testing the patches committed by other developers, pointing out the drawbacks of those patches, and proposing solutions and changes to fix the bug. Finally, the bug was resolved according to his suggestions and changes.

The number of participants per day and the number of comments per day when fixing upstream bugs locally are similar to when fixing in-house bugs and fixing upstream bugs upstream. Figure 3.10 and Figure 3.11 indicate that fixing an upstream

[21] https://bugzilla.redhat.com/show_bug.cgi?id=1150444
[22] https://bugzilla.gnome.org/show_bug.cgi?id=737456

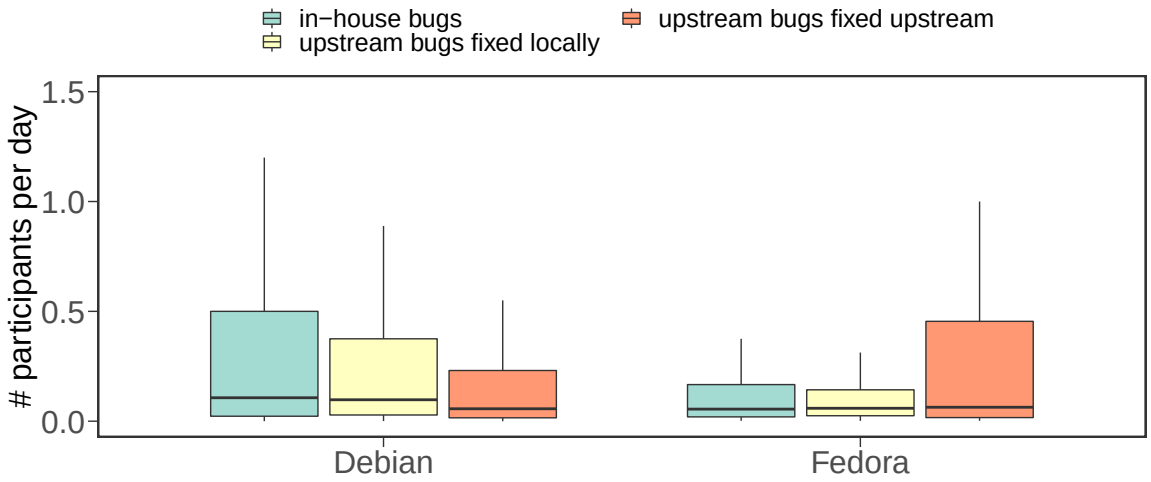


Figure 3.10: The number of participants per day to fix an upstream bug locally (yellow) is equal to the number of participants per day to fix an in-house bug (green) across the studied distributions. Note that these boxplots remove the top 1% of data points for better visualization.

bug locally does not require more participants per day, nor more discussion (comments) per day, compared to fixing in-house bugs and fixing upstream bugs upstream. One possible explanation is that upstream bugs that are fixed locally occur due to certain builds or modifications in a given distribution. For example, a Debian maintainer mentioned *"Actually we had to put some handlers back in after upstream removed them because the keys were not generated in the Debian kernel."*, which made the keystroke events be reported twice on certain platforms[23].

In Debian, distribution maintainers record more links to reference documents when fixing upstream bugs locally than when fixing in-house bugs, but fewer links than for upstream bugs that are fixed upstream. In Fedora, the number of reference links per comment is similar among all three types of bugs. The reference links in comments indicate whether fixing a bug needs more additional information (e.g.,

[23] https://bugs.debian.org/cgi-bin/bugreport.cgi?bug=586924

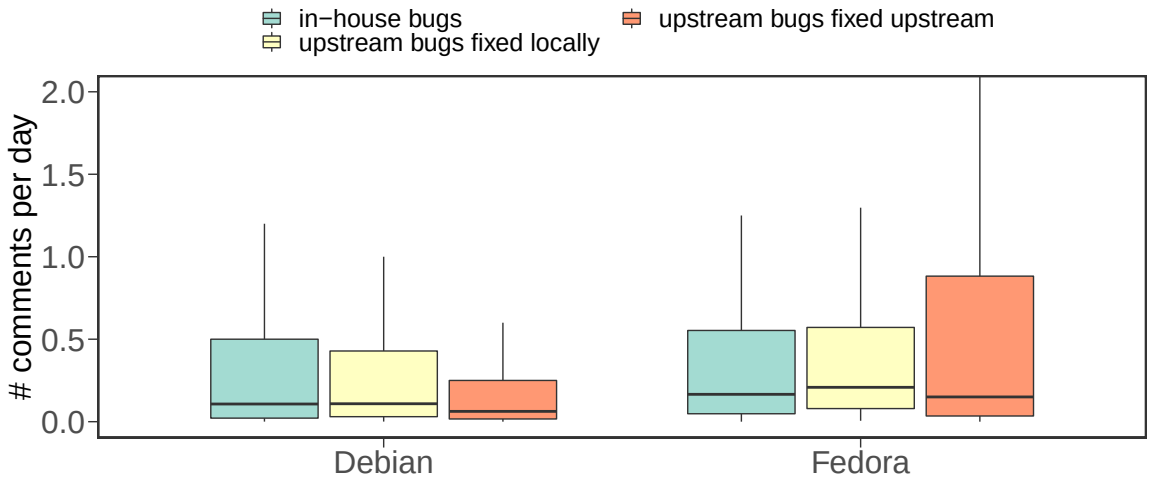


Figure 3.11: The number of comments per day in an upstream bug is similar to the number for an in-house bug. Note that these boxplots remove the top 1% of data points for better visualization.

design documents, requirements). Fedora follows the "upstream first" policy[24] that encourages its maintainers to discuss solutions of a bug in the upstream bug tracker, where the larger community can vet the solutions, as discussed in Section 3.3. As such, Fedora maintainers tend to comment on and discuss bugs in the upstream bug tracker rather than the distribution bug tracker. Figure 3.12 shows that the number of links to reference documents when fixing upstream bugs locally is larger than when fixing in-house bugs in Debian (Wilcoxon test: p-value $\ll 0.01$, $\alpha = 0.01$ and the negligible effect size). For example, an upstream bug[25] that is fixed locally and an in-house bugs[26] have 1.33 (4 links among 3 comments) and 0.43 (3 links among 7 comments) reference links per comment, respectively.

Maintainers maintain equal or fewer local patches per package for upstream packages within one distribution release compared to in-house packages. More

[24] https://www.redhat.com/en/blog/what-open-source-upstream
[25] https://bugs.debian.org/cgi-bin/bugreport.cgi?bug=489553
[26] https://bugs.debian.org/cgi-bin/bugreport.cgi?bug=424456

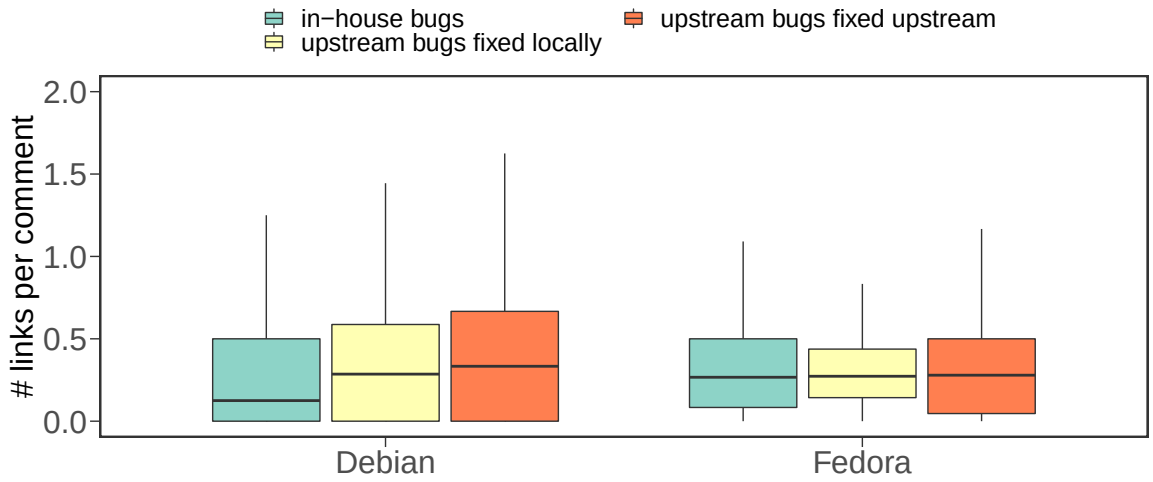


Figure 3.12: The number of reference links per comment in an upstream bug that is fixed locally (yellow) is larger than the number for an in-house bug (green) in Debian, though they are similar in Fedora. Note that these boxplots remove the top 1% of data points for better visualization.

specially, both Debian and Fedora maintain a median of 2 patches per upstream package, while they maintain a median of 3 and 2 patches per in-house package, respectively. As shown in Figure 3.13, 930 upstream packages (outliers) require a higher number of local patches than in-house packages. According to the guidelines of upstream package management in Linux distributions (Freedesktop.org, 2013), when the upstream bugs are critical (e.g., security issues or severe impacts), the distribution maintainers can produce local patches for such bugs to reduce their effects. For such a short-term fix, distribution maintainers potentially face a long-term maintenance effort. They need to check the compatibility of their patches with each subsequent release of upstream packages, until the upstream projects release a new version containing the bug fixes. On the other hand, in-house packages are not necessarily developed by in-house developers from scratch, they can take ownership of an orphaned upstream package or customize a package themselves as an in-house

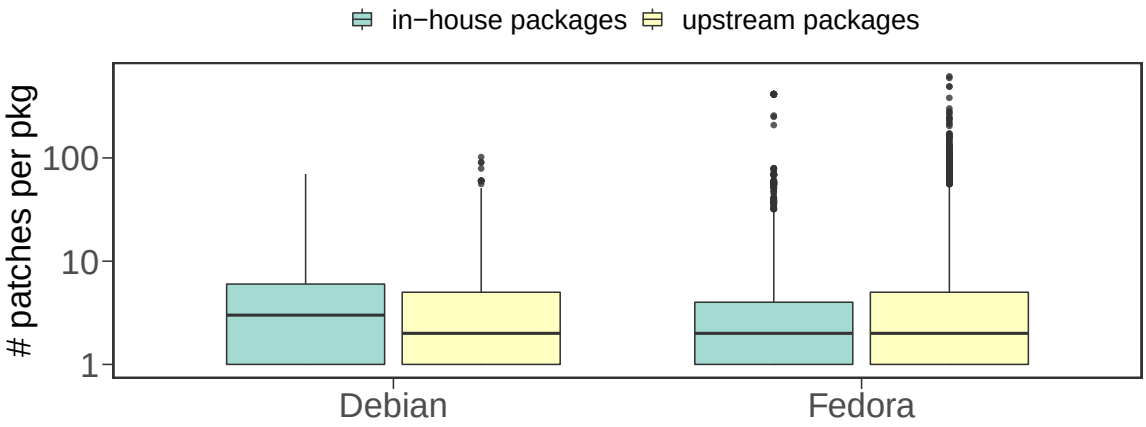
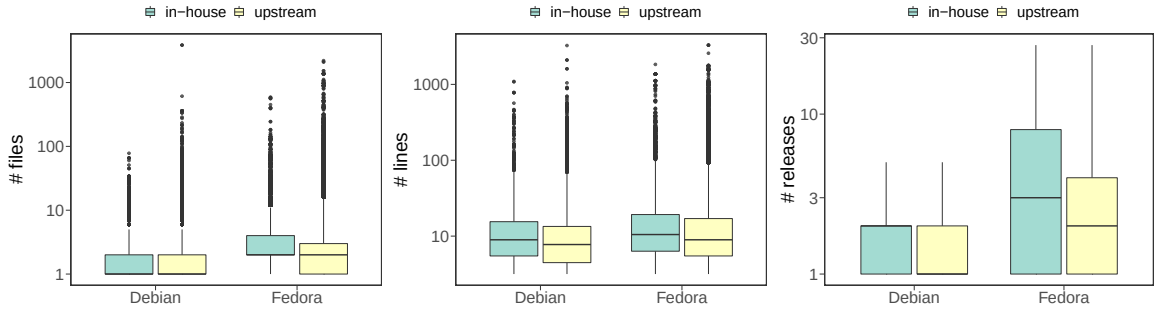


Figure 3.13: The number of patches per upstream package in one release of a given distribution is smaller than the number of patches per in-house package in Debian, while the number is equal for upstream and in-house packages in Fedora, though there are more outliers for upstream packages.

package (as mentioned in Section 3.3.2). Hence, in-house developers can also produce patches for bugs in in-house packages.

However, surprisingly we observe that the maintenance cost of the patches for upstream packages is slightly lower than for in-house packages in terms of the number of patches per package, with a small effect size (Wilcoxon test: p-value $\ll 0.01$, $\alpha = 0.01$).

Maintainers remove local patches for upstream packages faster than for in-house packages in terms of the number of distribution releases. Figure 3.14 shows the distribution of code churn and survival time of each patch file in one distribution release. The difference in survival time of patch files between upstream and in-house packages is statistically significant in both Fedora and Debian, with a p-value $\ll 0.01$ (Wilcoxon test: $\alpha = 0.01$). The Cliff's Delta effect size indicates that the magnitude of the difference is small (0.34) and median (0.45) in Fedora and Debian, respectively. As patches are produced to fix bugs and ensure the stability of a distribution, these



(a) #changed files (b) #changed lines (c) Survival time

Figure 3.14: The code churn and survival time of each patch file for the upstream and in-house packages in each Fedora and Debian release. Across the studied distributions, the maintenance effort for the upstream packages is lower than for the in-house packages in (a), (b) and (c).

patches will be removed afterwards, i.e., the survival time is shorter when the upstream projects release new bug fixes. In contrast, local patches for in-house packages are maintained much longer in terms of the number of distribution releases.

Although the cost of fixing upstream bugs upstream is similar to the cost of fixing them locally in terms of the number of participants per day and the number of comments per day, the cost of fixing upstream bugs upstream is larger than the cost of fixing them locally, in terms of the time users have to wait for bug fixes and the number of reference links per comment. Debian maintainers mention more links to references per upstream fixed bug than per locally fixed bug. Moreover, when maintainers produce local patches for bugs, the cost of producing such patches per upstream package is lower than per in-house package in terms of code churn.

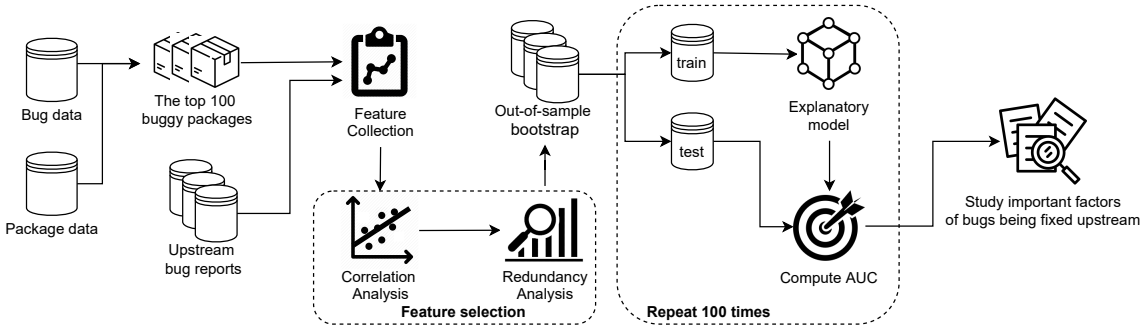


Figure 3.15: Overview of RQ3 model construction process.

RQ3: Which high-severity bugs are most likely to be fixed upstream?

Motivation: While observations in RQ2 indicate that distribution maintainers need less information to fix upstream bugs locally than upstream, they still need more information than to fix bugs of in-house packages. Similarly, users wait longer for local fixes to upstream bugs than for in-house packages. While compared to bugs fixed upstream, maintainers need less information to fix upstream bugs locally. Therefore, for a distribution as a whole, it is important to understand the characteristics of fixing bugs upstream. Specifically, we aim to understand what types of bugs are fixed upstream and whether the interaction between upstream and downstream influences bugs being fixed upstream. Therefore, we can offer suggestions for maintainers to reduce the maintenance burden and identify additional opportunities for assigning bugs to upstream.

Approach: We are interested in the key factors that influence the likelihood that a bug will be fixed upstream. Figure 3.15 illustrates an overview of the process to build our explanatory models. First, we select a dataset that consists of a balanced number of bugs that are fixed upstream and those that are fixed locally, with one dataset for Debian and one for Fedora. In particular, we randomly select 500 bugs that are fixed upstream

and 500 bugs that are fixed locally from the dataset in Table 3.2, obtaining two datasets of 1,000 high-severity bugs (**Model Dataset**).

Feature Collection

Then, we extract a total of 27 features in 6 dimensions regarding bugs, packages, and collaboration between upstream and distributions. Similar to prior work (Shihab et al., 2013; Ohira et al., 2012; Xia et al., 2014), we extract the common factors (e.g., description, bug-fixed time) of a bug report to characterize bugs (Shihab et al., 2013; Zimmermann et al., 2010; Xuan et al., 2012), reporters (Fan et al., 2018; Just et al., 2008; Xuan et al., 2012; Marks et al., 2011), and bug fixing activities (Mockus et al., 2002; Jeong et al., 2009). For example, Fan et al. (2018) found that reporter's experience is an important indicator that determines a valid bug or not. To understand the relation between upstream and downstream, we add several novel dimensions along with several metrics and highlight them in bold in Table 3.3. We now describe the dimensions with novel metrics in more detail.

Bug Fixing History. Since upstream projects are the source of upstream packages, maintainers report a bug to the upstream project before adding the reference link of the upstream bug report in the bug report. We leverage the number of reference links to the upstream bug reports to indicate such behavior. Furthermore, an upstream bug can affect many distributions and their users in the Linux ecosystem. Therefore, a similar upstream bug can already be reported in the issue tracking systems of different distributions. We add two metrics regarding the similarity of previous bug reports either between upstream and distributions, or between peer distributions (see Table 3.3).

Additionally, since upstream packages may affect in-house packages or vice versa,

we add the number of reference links to the related in-house bug reports to indicate the interference between upstream and in-house packages. Furthermore, bug triage is an essential step in the bug fixing process but a time-consuming task (Hu et al., 2014). We add one novel metric, i.e., the number of bugs that are fixed upstream before the current bug report in a distribution, to measure the performance of the package maintainers with respect to upstream bugs.

Internal Collaboration Activities. The number of comments in a bug report is related to its fixing time (Hooimeijer and Weimer, 2007). Hence, we further extract the URLs in the comments as an indicator to reflect the maintenance effort of maintainers to search the relevant information (e.g., design documents, bug reports in other bug trackers). Then, we classify these URLs into internal links (referring to internal documents) and external links (referring to external documents). When maintainers discuss the fix for a bug, they may post the links regarding the relevant information (e.g., design doc, similar bugs or solutions) in the comments of the bug report. Therefore, we use the number of external and internal links to measure the communication activities. In addition, according to RQ1, bugs in upstream packages might be addressed by maintainers instead of upstream developers. We measure the number of maintainers involved with fixing upstream bugs based on the comments of a bug report. We identify those maintainers via their email address, which contains the domain name of their distribution (e.g., debian.org) (Debian.org, 2020b; Fedora.org, 2020c).

External Collaboration Activities. In general, an upstream bug is first reported by a user of a distribution before maintainers forward the upstream bug to upstream developers. While upstream developers are fixing the bug, they often lack the information

Table 3.3: The dimensions of features in our models. The feature names in bold are new features with respect to the relation between upstream and distribution fixing bugs.

Feature Name	Type	Description
Bug Characteristics		
#Description	Numeric	The number of words in the description of the bug
Severity	String	The severity of the bug
Priority	String	The priority of the bug
Reporter Characteristics		
Is internal reporter	Boolean	States whether the reporter is an in-house developer. In-house developers are determined based on the developer's email that contains the domain name of a distribution (e.g., debian.org) Debian.org (2020b); Fedora.org (2020c)
Reporter experience	Numeric	The number of bugs reported by the same developer before this bug
Reporter experience to the package	Numeric	The number of bugs reported to the same package by the same developer before this bug
Bug Fixing Activities		
Fixed time	Numeric	The time duration in the number of days between when the bug was reported and when the bug is fixed
Triaged time	Numeric	The time duration in the number of days between when the bug was reported and when developers started to fix the bug
Bug Fixing History regarding the current bug		
#Related in-house bug reports	Numeric	The number of related in-house bug reports that were reported before the current bug
#Referred upstream links	Numeric	The number of reference links to the upstream bug reports
Peer similarity score	Numeric	The cosine similarity between the bug report and peer-distributions' bug reports that were reported before the current bug
Similarity score[1]	Numeric	The cosine similarity between the current bug and the corresponding upstream bug reports
Package category	String	The application domain of the package
#Fixed bugs	Numeric	The number of fixed bugs in the package before the current bug report
#Fixed-upstream bugs	Numeric	The number of bugs that are fixed upstream in the package before the current bug report
#Participants	Numeric	The number of participants across the bug report history before the current bug report
Internal Collaboration Activities		
#Comments	Numeric	The number of comments in the bug report

#External links in comments	Numeric	The number of unique external reference links that developers posted in the comments
#Internal links in comments	Numeric	The number of unique internal reference links that developers posted in the comments
%Internal comments	Numeric	The proportion of comments added by distribution maintainers or developers in the bug. Distribution maintainers and developers are determined based on the developer's email that contains the domain name of a distribution (e.g., debian.org) Debian.org (2020b); Fedora.org (2020c)
#Maintainers	Numeric	The unique number of maintainers contributing to the bug by posting comments
External Collaboration Activities		
Has upstream developers	Boolean	States whether upstream developers contributed to the bug by posting comments
#External developers	Numeric	The unique number of external developers contributing to the bug by posting comments
%External comments	Numeric	The proportion of comments added by external developers in the bug. When the email of a developer does not contain the domain name of a distribution, we determine the developer is an external developer Debian.org (2020b); Fedora.org (2020c)
Has distribution maintainers[1]	Boolean	States whether distribution maintainers attached comments to the upstream bug report
#Comments from distribution maintainers[1]	Numeric	The number of comments posted by distribution maintainers in the upstream bug report
#Comments upstream[1]	Numeric	The number of comments in the upstream bug report

[1] The features are computed from the corresponding upstream bug report if it is available.

and context of a specific bug occurring downstream (Ma et al., 2017) and require additional information (e.g., logs from the user, tests in certain circumstances). Maintainers coordinate with them to provide the required information, even from the user who reported the bug. We use four metrics extracted from the comments of the local and upstream bug reports to reflect the interaction between upstream and downstream (see Table 3.3). Due to the variety of upstream bug trackers (e.g., GitHub, Bugzilla), it is a time-consuming task to collect all the bug reports of upstream projects. Hence,

we analyze the distribution of upstream bug trackers and only consider the upstream projects using Bugzilla and Debbugs (i.e., Debian bugs) as their bug trackers. In the selected 100 upstream projects, 73% of the upstream reference links in bug reports link to such bug trackers.

Model Construction and Performance Assessment

Finally, we build logistic regression models to understand the relation between the studied metrics and the possibility of getting a bug fixed upstream (see Figure 3.15). Our models classify upstream bugs into being fixed upstream or in-house. Since the results in RQ1 show that each Linux distribution has separate characteristics, we build a model for each Linux distribution instead of one model for all the studied distributions.

Feature Selection. After extracting the metrics, we apply correlation and redundancy analysis, similar to prior work (Lee et al., 2020; Rajbahadur et al., 2021; da Costa et al., 2018), before constructing our models. The correlation analysis (Lee et al., 2020; Rajbahadur et al., 2021) step removes highly correlated variables (whose Spearman correlation is $> 80\%$), since correlated variables can affect the interpretation of the model. While the correlation analysis removes correlated variables, it does not detect all the redundant variables, i.e., variables that can be predicted with more than one other independent variable. Therefore, we then conduct a redundancy analysis (McIntosh et al., 2016; da Costa et al., 2018) that identifies for each metric whether it can be predicted by other metrics. The redundancy analysis is performed by the `redun` function[27] from the `Hmisc` package in R. The discarded metrics are marked with the dagger symbol (†)

[27] https://www.rdocumentation.org/packages/Hmisc/versions/4.2-0/topics/redun

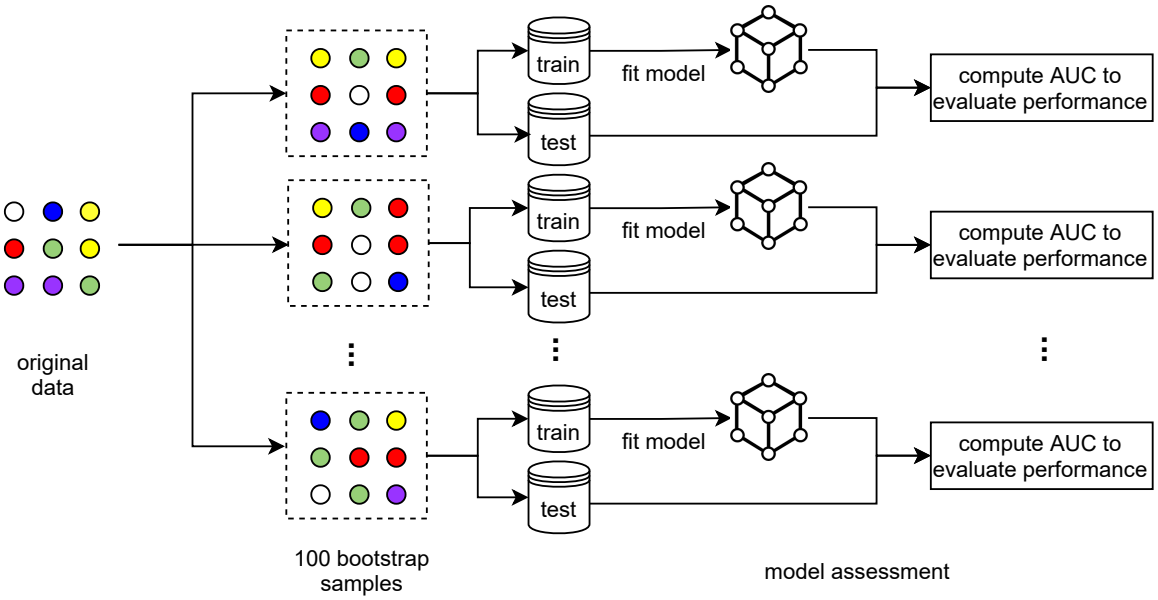


Figure 3.16: The model building and validation process of the models used in RQ3.

in Table 3.4.

AUC. We assess the performance of our models using the area under the curve (AUC), which is calculated by 100 out-of-sample bootstrap iterations with replacement (Thongtanunam et al., 2016; McIntosh et al., 2016). AUC is a better measurement for evaluating the performance of a classification algorithm (Ling et al., 2003), since, unlike accuracy, AUC is not impacted by class distributions. Moreover, AUC evaluates performance across a range of thresholds instead of requiring one specific threshold like precision, recall, and F-Measure do. In each bootstrap iteration, as shown in Figure 3.16, we draw a bootstrap sample set from the **Model Dataset** with replacement for each of the studied distributions. The train set is re-sampled with replacement from the bootstrap sample, while the test set consists of the data points from the bootstrap sample that are not in the train set. The AUC of our models is calculated by the test set, which enables to compute true positive rates and false positive rates on

unseen data.

To assess the performance of our models across the studied distributions, we train a model on one dataset and assess its performance on the other dataset. For example, we train a model on the Fedora dataset and test this model on the Debian dataset. Note that we remove the features that do not have the corresponding ones across the studied distributions. We use the commonly occurred features which are the majority of our extracted features.

Log-likelihood ratio tests. In addition, we use a log-likelihood ratio (LR) test (Huelsenbeck and Crandall, 1997) to evaluate the performance of our regression models fit to the data at the dimension level. Similar to prior work (Thongtanunam and Hassan, 2020; Lee et al., 2020), we compute the overall LRχ^2 to represent the performance of fit for the model using all the studied metrics (i.e., the full model) against the null model (i.e., a model without any metrics). We then compute the ΔLRχ^2 to indicate the performance of fit for the model with the metrics of interest against the model without using the metrics of interest.

Association analysis. We use Wald statistics to analyze the explanatory power (Wald χ^2) of each studied metric to the fit of the regression models. The larger the Wald χ^2 value, the higher the explanatory power of a metric to the model (Harrell Jr et al., 1984). In addition, we use the regression coefficients of the regression model to examine the direction of the association of the studied metrics.

Results: Our logistic regression models reach a median AUC of 0.87 and 0.73 to explain the likelihood that a bug will be fixed upstream in Fedora and Debian, respectively. We note that our models for the two studied distributions are remarkably good compared to a random model (AUC of 0.5). The high AUC values in modeling Fedora

and Debian suggest that our models have a high explanatory power when studying bugs being fixed upstream, hence, we analyze our models' features in more details.

The Fedora model achieves an AUC of 0.63 to explain the likelihood that a bug will be fixed upstream for Debian, and the Debian model achieves an AUC of 0.81 for Fedora. These AUC values indicate the generalizability of our models for understanding bugs being fixed upstream. An upstream project can be part of multiple distributions, suggesting that a potential issue can occur in a common set of upstream packages across different distributions. The generalizability of our models across the studied distributions indicates that bug fixes from upstream projects can propagate across distributions, thus sharing common bug fixing patterns.

Dimension Level. **The dimension of bug fixing history has the strongest association with the likelihood that a bug will be fixed upstream in Debian, while the dimension ranks 2nd in Fedora.** Table 3.4 shows that 4 out of 5 metrics and 2 out of 4 metrics that capture the history of a fixed bug contribute a significant amount of explanatory power to our Fedora and Debian models, respectively. Moreover, Table 3.4 shows that the $\Delta\text{LR}\chi^2$ values for our studied metrics account for 15% and 44% of the overall $\text{LR}\chi^2$ in the Fedora and Debian models, respectively. This result suggests that a bug being fixed upstream tends to be related most to its fixing history in the ecosystem. For example, the number of bugs that are fixed upstream before the current bug reflects the extent of support from upstream developers to resolve their bugs in distributions.

On the other hand, **the dimension of internal collaboration activities between upstream projects and distribution maintainers accounts for the highest percentage (i.e., 38%) of the overall $\text{LR}\chi^2$ in the Fedora model, while the dimension**

Table 3.4: Statistics of our regression models. The goodness of fit for the metrics of interest (ΔLRχ^2) is shown in a proportion to the overall LRχ^2.

		Fedora	**Debian**
Overall LRχ^2		526***	211***
AUC		0.87	0.73
AUC (across distributions)		0.63	0.81
Bug Characteristics	ΔLRχ^2	31***(6%)	17***(8%)
#Description	χ^2	5.27* (+)	0.86^o (+)
Severity	χ^2	0.09^o	**15.22****
Priority	χ^2	**22.79****	-
Reporter Characteristics	ΔLRχ^2	1.1(0.2%)	12***(6%)
Is Internal reporter	χ^2	0.79^o (-)	5.25 * (-)
Reporter experience	χ^2	0.01^o (-)	4.05* (-)
Reporter experience to the package	χ^2	0.15^o (-)	†
Bug Fixing Activities	ΔLRχ^2	17***(3%)	1(0.5%)
Fixed time	χ^2	**16.41***** (+)	0.97^o (+)
Triaged time	χ^2	0.09^o (+)	-
Bug Fixing History	ΔLRχ^2	81***(15%)	92***(44%)
#Related in-house bug reports	χ^2	1.91^o (+)	†
#Referred upstream links	χ^2	†	†
Peer similarity score	χ^2	4.25* (–)	1.35^o (-)
Similarity score	χ^2	**21.88***** (+)	**54.44***** (+)
Package category	χ^2	†	†
#Fixed bugs	χ^2	†	†
#Fixed-upstream bugs	χ^2	**19.24***** (+)	8.17** (+)
#Participants	χ^2	**13.90***** (-)	0.34^o (-)
Internal Collaboration Activities	ΔLRχ^2	201***(38%)	23***(11%)
#Comments	χ^2	7.21** (-)	†
#External links in comments	χ^2	6.77** (+)	7.84** (+)
#Internal links in comments	χ^2	**46.14***** (-)	**9.32**** (-)
%Internal comments	χ^2	0.01^o (+)	5.25* (-)
#Downstream developers	χ^2	**16.68***** (–)	4.12^o (+)
External Collaboration Activities	ΔLRχ^2	8*(2%)	15**(7%)
Has upstream developers	χ^2	2.67^o (+)	0.11^o (+)
Has distribution contributors	χ^2	†	†
#External developers	χ^2	†	**13.49***** (-)
%External comments	χ^2	5.06* (-)	3.96* (+)
#Comments from distribution contributors	χ^2	†	†
#Comments upstream	χ^2	†	†

Statistical significance: *** $p < 0.001$, ** $p < 0.01$, * $p < 0.05$, ^{o} $p > 0.05$
† Discarded due to the correlation or redundancy analysis.
- Metrics are not available in the distribution.
Bold numbers indicate that correlation is significant, taking into account Bonferroni correction.

accounts for 11% (2nd ranked) in the Debian model. Since developers are geographically distributed, they have several communication channels (e.g., emails, Internet Relay Chat (IRC) channels) (Panichella et al., 2014) to discuss and make decisions in software development other than the bug report comments. Therefore, future research may study whether collaboration activities occur in another communication channel. In addition, the strategy of upstream package management in Fedora is to mainly develop and fix bugs upstream, as mentioned in Section 3.3. Internal collaboration activities could be a negative indicator of a bug being fixed upstream.

Feature Level. **The similarity score between the upstream and distribution bug reports is highly associated with the likelihood that a bug will be fixed upstream (p-value < 0.001).** Table 3.4 indicates that there is a significant positive correlation between the cosine similarity of the upstream and distribution bug reports and the likelihood that a bug will be fixed upstream. In particular, the similarity score of the upstream and distribution bug reports contributes a relatively large amount of explanatory power in the Fedora (10%) and Debian (38%) models. This result indicates that the more likely a similar bug was reported in the past, the more likely this bug will be fixed upstream. This result suggests that maintainers should take into account identifying/reusing similar bug reports upstream before requesting an upstream bug, which could save unnecessary effort. This result is also in line with the observation of the guidelines for working with upstream in Debian that *"If the package has an upstream bug tracker then searching it for similar reports can be useful"*.[28]

The number of bugs that are previously fixed upstream in one upstream package is significantly associated with a bug being fixed upstream (p-value < 0.01). Table 3.4

[28] https://wiki.debian.org/BugTriage

shows that the number of previously fixed bugs from upstream packages has a positive association with the possibility of a bug being fixed upstream. This result indicates that some upstream projects might be used to interacting more closely with distributions, being more likely to continue such interactions. On the other hand, it might not be likely to expect upstream projects without such interactions to suddenly start doing this by themselves.

The number of internal links in comments is significantly associated with a bug being fixed upstream (p-value < 0.01). Table 3.4 shows that the number of internal links in comments has a negative association with the possibility of a bug being fixed upstream. This result suggests that maintainers discuss how to resolve a bug by posting additional internal references in comments, the less likely the bug will be fixed upstream. Similarly, the number of external links in comments has a positive association with the possibility of a bug being fixed upstream. The more maintainers refer to external links to discuss the fix for a bug, the more likely the bug will be fixed upstream.

Additionally, there is a gap in the explanatory power of the number of internal links in comments between Debian (6%) and Fedora (21%). One possible explanation is the difference in strategy of Debian and Fedora to manage upstream packages, as discussed in Section 3.3. Fedora prefers to vet solutions in upstream projects rather than distribution bug reports. Therefore, the number of internal links in comments has a larger amount of explanatory power, and a negative association with a bug being fixed upstream in the Fedora model. In other words, the more internal reference links in a Fedora's bug report, the more likely the bug is not related to the source code of the upstream project. By contrast, the number of external links in comments has a similar amount of explanatory power in both models.

Our regression models achieve a median AUC of 0.87 and 0.73 for the Fedora and Debian datasets, respectively. The dimension of bug fixing history has a significant association with the likelihood that a bug will be fixed upstream in the Debian (44%) model, while the dimension of internal collaboration activities is the largest in the Fedora (38%) model. The similarity score between upstream and distribution bug reports, the number of internal reference links, and the number of bugs that previously have been fixed upstream in a package share a significant association with the likelihood that a bug will be fixed upstream (p-value < 0.01).

3.5 Discussion

This section discusses the implications of our findings. We focus on the interaction across distributions and upstream projects (Section 3.5.1) and distribution maintainers of upstream packages (Section 3.5.2).

3.5.1 Implications for the interaction across distributions and upstream projects

There is a need for tool support to explicitly track the bug fixing process across upstream and distributions. In RQ1, we observe that for the majority (at least 73%) of upstream bugs it is not clear who fixed them. The two popular and established Linux distributions that we studied use a mixture of different practices (e.g., certain fields such as labels or tags in a bug, but do not enforce these practices) to record that a bug is related to upstream and has been fixed upstream. Since such a recording mechanism is not a mandatory feature in the studied distributions, maintainers might not record

the information of bug fixing before closing a bug. Missing such information increases the difficulty of maintainers to trace the progress of fixing an upstream bug and the difficulty of new maintainers to learn the history of managing a package. Furthermore, in case of vulnerabilities, it is important to have full traceability information.

Therefore, distributions should establish a means to unambiguously record this information across upstream projects and their distribution, even though upstream packages are packaged and distributed by multiple Linux distributions, corresponding to an N-to-N relation of traceability links between bug reports of packages and distributions. Trying to establish a centralized database, while the most intuitive solution, might not be feasible in practice, since it is unclear where to locate the database and who should be responsible for maintaining it. Furthermore, an upstream project usually focuses on bugs reported in its own issue tracker. Since a given upstream project can be packaged and distributed by several distributions, the upstream developers need substantial additional effort to record the corresponding distribution bug ids in their issue tracker and maintain this information. This is especially true since each distribution has its own bug repository technology (making automation less effective) and upstream projects usually lack human resources and are relatively small.

On the other hand, the issue of software supply chain attacks has led to increasing adoption of different metadata formats for specifying a project's Software Bill of Materials (SBOM[29]). Technologies like SPDX (Software Package Data Exchange) allow to (semi-)automatically generate the SBOM information for a given project or distribution, including lists of components, dependencies, etc. This SBOM information is shipped in a given release of a software product. Some SBOM technologies even allow to specify which patches were installed, which bug fixes (for which bugs), etc., which

[29] https://www.ntia.gov/SBOM

seems to be a promising means for both upstream projects and distribution alike to formally specify the traceability information relative to their product in a scalable manner. Of course, this does not solve the issue of recovering traceability links for historical releases.

Active collaboration between upstream developers and distribution maintainers should be encouraged in order to fix bugs and better distribute fixes to the needed users. In RQ3, our explanatory models indicate that the similarity score between upstream and distribution bug reports and the number of previously fixed bugs in upstream projects are associated with a higher likelihood of a bug being fixed upstream. While upstream is the best place to perform quality assurance (Freedesktop.org, 2013), sending more bugs to upstream and fixing them there requires contributions from both sides (i.e., upstream and distributions).

First, an upstream project might not have an explicit bug tracker or different bug tracking technology than the distribution where distribution maintainers report their bugs to and search for the old bugs of the upstream project. In such a case, the collaboration activities between upstream and distributions happen in other channels (e.g., emails), which is difficult for people who are less involved with tracking or reviewing the fixing history.

Second, distribution maintainers require providing sufficient information (e.g., logs, configurations) of a bug for upstream developers when they validate whether the bug is valid in the upstream project or specific to a given distribution.

Furthermore, even if the upstream project would be interested in fixing a bug, the fix typically only applies directly to a new version of the upstream project. The results

in RQ2 indicate that both Debian and Fedora maintain a median of 2 patches per upstream package. Since distributions offer their users a certain period of support, distribution maintainers need a backport fix for the old versions, especially for security fixes. For example, Debian maintainers had to backport the fix for a security bug[30] in the *glib2.0* package from the upstream release 2.60.0 to the old version 2.58.3 in Debian.

In addition, it would be essential for distributions to collaborate together more systematically in terms of exchanging fixes to common bugs. When bug fixes are made by one distribution instead of the upstream project, these fixes will not be shared with the other distributions until the upstream project integrates the fix in its next release, unless distribution maintainers proactively search for peer fixes in other distributions. For example, a bug[31] in the *augeas* package was resolved in Fedora more than one month earlier than when a similar bug[32] was reported in Debian. At the same time, the bugs in an upstream package might affect several distributions without them knowing this, hence precious time could be saved by closer collaboration between peer distributions.

3.5.2 Implications for distribution maintainers

There is a need for tool support to scan existing bug reports in the ITS of the upstream project or peer distributions for similar reports that potentially were fixed upstream. Since an upstream package can be part of several distributions, a bug in

[30] https://bugs.debian.org/cgi-bin/bugreport.cgi?bug=931234
[31] https://bugzilla.redhat.com/show_bug.cgi?id=772257
[32] https://bugs.debian.org/cgi-bin/bugreport.cgi?bug=731132

the upstream package in a given distribution might have been reported by the maintainers in other distributions, and potentially already have a fix. Our models suggest that the similarity score between a previously reported upstream bug report and a distribution bug report is an important factor for a bug being fixed upstream (Table 3.4). In our dataset, there are 9% and 6% of the high-severity bugs in Debian and Fedora, respectively, with a median similarity score of 0.87 and 0.88 to their correlated upstream bug report. These numbers suggest a high similarity score of bug reports between upstream projects and distributions, even though the actual proportion of bugs with such high similarity is low (<10%). Since we only computed the similarity score for bugs that contain a reference link to their respective upstream bug report, there might be a larger set of bugs with such a high similarity score to an upstream report, unknown to maintainers, and hence, without such information being recorded explicitly. As such, using techniques of duplicate bug report detection could help find such cases.

Although there are some existing techniques to detect duplicate bug reports (Boisselle and Adams, 2015; Alipour et al., 2013), these techniques typically focus on detection between individual software systems in a 1-to-1 relation. In Linux distributions, such tools might not be able to detect similar bug reports between upstream projects and distributions, which is an N-to-N relation (N upstream projects vs. N distributions carrying them). In addition, after an upstream project ships a release, the package version of the upstream release is usually changed by maintainers to meet their distribution versioning guideline, i.e., adding the version number used in a given distribution. The upstream version of the release reflects several different version numbers in several distributions. On the other hand, a distribution might skip a certain package version of the upstream release. Hence, the complexity of mapping package versions

from distributions to upstream projects increases the difficulty of identifying similar bug reports across distributions and upstream.

There is a need for tool support to integrate the web context of the reference links in the comments of a bug report and provide a summary of these links to speed up fixing upstream bugs. Having more external (less internal) links in bug comments is associated with the possibility of a bug being fixed upstream (Table 3.4). Since the maintainers of an upstream package can leave or join over time, a new maintainer needs to learn how to fix bugs in the package and might take care of an unfixed bug from other retired maintainers. In addition, anyone who is interested in fixing bugs can contribute to the comments of a bug. The links in the comments provide an additional channel for gathering information in the bug fixing process, which can be a source of additional context about a bug. The results in RQ2 indicate that Debian maintainers need more links to reference documents when fixing upstream bugs locally than when fixing in-house bugs (Figure 3.12). An external link can refer to information related to the bug (e.g., the configuration setting is disabled by default) in the upstream package. In contrast, the internal links in the comments of a bug report indicate that the upstream bug might be related to other packages in a given distribution (e.g., conflicts with other packages or the distribution-specific process). To increase the ability of fixing upstream bugs, the need of tool support to retrieve and summarize the links in comments helps maintainers to have comprehensive information related to a bug.

3.6 Threats to Validity

3.6.1 Construct Validity

Although we only study the high-severity bugs in two studied distributions, our dataset contains a large number of bugs in total (i.e., 143,362). High-severity bugs typically indicate fatal errors and even crashes, while low-severity bugs represent the effect of such bugs is low on the functionality of a software system (Lamkanfi et al., 2010). Although the severity field of a bug might be filled out randomly (Lamkanfi et al., 2010), our findings from high-severity bugs reveal the lower-bound of involved cost of fixing upstream bugs in-house, since lower-severity bugs also need to be addressed.

In addition, developers might assign the severity label of a bug based on their own knowledge. However, Debian and Fedora have been in the top 10 popular Linux distributions over the past decade, according to the `distrowatch.com` rankings. They have derived and documented a mature policy of assigning severity labels (Debian.org, 2020c; Fedora.org, 2020b) for their maintainers to fix bugs and maintain their popularity. Note that a bug could be tagged as severe initially but later converted into a low severity bug, or vice versa. Since our work aims at studying bugs that really were severe, the final status of a bug, captured the latest bug characteristics, suffices for us.

We classify packages in a distribution into upstream and in-house packages based on the reference link to the source upstream project in related web pages of a package or package repositories, when available. Since an in-house package can be packaged from an upstream project a long time ago and the upstream project is no longer maintained, the link to the upstream project could have been removed. It is challenging to

derive a heuristic to discriminate between such a type of package and in-house packages even though such a type of package differs from in-house packages that are developed from scratch. In our manual study sample in Section 3.3.2, we observe that 1 (4%) out of the 25 in-house packages was such a type of package. Future work can study such a type of package and evaluate its impact.

Although we aim to identify bugs that are fixed upstream or locally in a comprehensive manner, like any heuristic-based approach, we may miss other scenarios of upstream fixed bugs and local fixed bugs. For instance, a bug labeled as fixed upstream may have a root cause related to the different packaging processes between upstream and the distribution, where maintainers need to customize the upstream project (e.g., modify locations for configuration files) (Adams et al., 2016). When maintainers cannot customize the upstream project by themselves, they need to ask the upstream to adjust its packaging process (e.g., add configurations, switch to a new build system) to fix bugs in the packaging process. Since the bugs in the packing process only happen when maintainers package the upstream project to fit their distribution policy, we do not consider the case of customization for upstream packages and do not investigate the logs of the packaging process.

Since the studied distributions provide their maintainers flexibility on how to deal with packages, maintainers decide whether or not to reference relevant commits in their distribution's bug reports and changelogs, or to include bug ids in the commit messages of their version control system. For example, Debian's *openssh* repository[33] contains 8,492 commits between 2005 and 2019 (i.e., our studied period), 769 (9%) of which included a total of 212 Debian bug ids, with 28 out of these 212 bugs being high-severity bugs. Since Debian has 180 high-severity bug reports for *openssh* in its bug

[33] https://salsa.debian.org/ssh-team/openssh/-/commits/master

repository, this means that only 28 (16%) out of 180 high-severity bugs were explicitly recorded in commit messages. Upstream developers have the same freedom w.r.t. including bug ids in the commit messages of their version control system.

Furthermore, a bug fix can be made by a maintainer of another distribution, which is neither a local fix nor an upstream fix. For example, bug 851052[34] was a flickering issue while inputting characters and happened in several distributions (e.g., Ubuntu). A Ubuntu contributor made a commit directly to the Debian repository to fix the bug.

3.6.2 Internal Validity

For the measures of issue discussions (i.e., comments) and user waiting time for fixes, we only examine the number of comments per day, the number of links per comment and user waiting time between upstream bugs and in-house bugs. The experience of issue commenters and issue difficulty are not considered in our study. Nevertheless, we focus on high-severity bugs to partially mitigate these issues and provide the overall insight of the costs of fixing upstream bugs for distribution maintainers.

In RQ3, the regression models present the relation between the possibility of a bug that is fixed upstream and a set of software metrics. We build our explanatory models to gain an understanding of how distribution maintainers submit bugs to upstream and how these bugs are fixed upstream successfully. Although we do not use a mixed-effect model, our models reach high AUC values, i.e., 0.87 and 0.73 for Fedora and Debian, respectively. There might be other factors that influence a bug that is fixed upstream. Hence, we encourage future research to add more features to improve the explanatory power of the model and conduct qualitative studies to understand how upstream and

[34] https://bugs.debian.org/cgi-bin/bugreport.cgi?bug=851052

downstream developers collaborate together to fix a bug.

3.6.3 External Validity

We perform a case study on two Linux distributions. Although Debian is commonly used as a case study of prior research (Herraiz et al., 2011; Claes et al., 2015; Boisselle and Adams, 2015; Davies et al., 2010) and Fedora is one of the popular Linux distributions[35], the results may not generalize to all Linux distributions. Moreover, Fedora and Debian represent the roots of their families. Since derivative distributions can inherit packages from Fedora and Debian, maintainers in derivative distributions can also report bugs in those packages to Fedora and Debian. Maintainers fix such bugs that slip through into derivative distributions for downstream developers in their families. Hence, additional replication studies are required in order to characterize the practices and challenges of upstream package management in the Linux ecosystem.

3.7 Chapter Summary

In this chapter, we investigate 143,362 high-severity bugs across two Linux distributions to understand the prevalence of upstream bugs and the cost of fixing them for distribution maintainers in practice.

Our analysis shows that upstream bugs are dominant across the studied distributions, yet only less than 14% of these bugs are explicitly recorded fixed by upstream and at least 13% of these bugs are recorded fixed by the distribution. Our findings suggest that the vast majority (at least 73%) of upstream bugs are unknown who fixed them. In

[35] https://distrowatch.com/index.php?dataspan=2020

other words, there is a lack of traceability between high-severity bugs and their fixes, potentially impacting maintenance activities by other contributors than a package's maintainer.

Our measurements for upstream bugs indicate that the cost of fixing an upstream bug locally requires a longer needed time for users to wait for fixes and more reference links per comment, compared to bugs in in-house packages, but less than for upstream bugs fixed upstream. However, the number of participants per day and the number of comments per day are similar. In addition, the patches made for the upstream packages are simpler than the patches for the in-house packages in terms of code churn.

Finally, we build regression models to understand the factors that influence bugs being fixed upstream. We suggest that maintainers search for bug fixing history in the upstream projects before starting to fix a bug. Since searching the fixing history of bugs increases the effort for maintainers, we suggest to leverage techniques of duplication bug detection. Furthermore, the similarity score of upstream and distribution bug reports, the number of reference links, and the number of bugs that have previously been fixed upstream in a package are important indicators of a bug being fixed upstream.

CHAPTER 4

Vulnerability coordination in the Linux distributions

Vulnerabilities in software systems not only lead to loss of revenue, but also to loss of reputation and trust. To avoid this, software providers strive to remedy vulnerabilities rapidly for their customers. However, in open-source development, the providers do not always control the distribution of their software themselves, but instead typically rely on Linux distributions to integrate and distribute upstream projects to millions of end users, which increases the difficulty of vulnerability management. In addition, an upstream project is usually packaged into several Linux distributions so that a vulnerability can propagate across multiple distributions via the upstream project. In this chapter, we empirically investigate a large number of vulnerabilities registered with the Common Vulnerabilities and Exposures (CVE) program in two popular Linux distributions, i.e., Debian (21,752 CVE-IDs) and Fedora (17,434 CVE-IDs), to study the practices of vulnerability management in such ecosystems. We investigate the lifecycle of fixing vulnerabilities, analyze how fast it takes for a vulnerability to go through each phase of its lifecycle, characterize the commonly occurring vulnerabilities that affect both distributions, and identify the practices that developers use to fix vulnerabilities. Our results suggest that the vulnerability testing period (i.e., the period from when the vulnerability fix is committed for testing to when the vulnerability fix is released) accounts for the largest number of days (median of 15 days) in Fedora. 74% (i.e., 16,070) and 92% (i.e., 16,070) of the vulnerabilities in Debian and Fedora, respectively, occur in both Linux distributions, which we refer to as common security vulnerabilities (CSVs). This result is impacted by the package selection and customization of

the distributions. Finally, on a representative sample of 345 fixed CSVs, we find that upstream projects were responsible for fixing 303 (85%) and 267 (76%) out of the 345 CSVs in Debian and Fedora, respectively, with distribution maintainers integrating those fixes. Our work aims to gain a deeper understanding of the current practices in the vulnerability management of Linux distributions, and propose suggestions to distribution maintainers for better mitigation of the risks of vulnerabilities.

An earlier version of this chapter is published by in the Empirical Software Engineering Journal (EMSE) Journal Lin et al. (2023).

4.1 Introduction

Attackers seek flaws or weaknesses in software systems as a weapon, also known as security vulnerabilities, and exploit them to attack or steal information from the victim systems. Given the increase of malicious attacks on software systems, developers' main strategy to reduce the exploit of vulnerable code is to try to update their software as quickly as possible with patches (fixes) for these vulnerabilities. In particular, each vulnerability has its lifecycle that consists of several phases defined by the events of its discovery, disclosure, exploitation, and patching (Shahzad et al., 2012), as shown in Figure 4.1. The highest risk in a vulnerability lifecycle happens in the phase after its disclosure (Shahzad et al., 2012) as miscreants are actively trying to make *zero-day* attacks (Anderson, 2002). Sometimes the exploitation occurs even prior to the disclosure, when a vulnerability is discovered by a Black-Hat hacker who does not report the vulnerability to the affected vendors. The lifecycle of the vulnerability ends when all victim systems have adopted the fix patch (Shahzad et al., 2012).

Ironically, in today's open-source software (OSS) ecosystems, the patch adoption phase is not as trivial as it sounds, since a software project can be deployed and used directly or be integrated into a larger system, creating a system of systems, which forms a software supply chain (Ellison et al., 2010; Al Sabbagh and Kowalski, 2015). In other

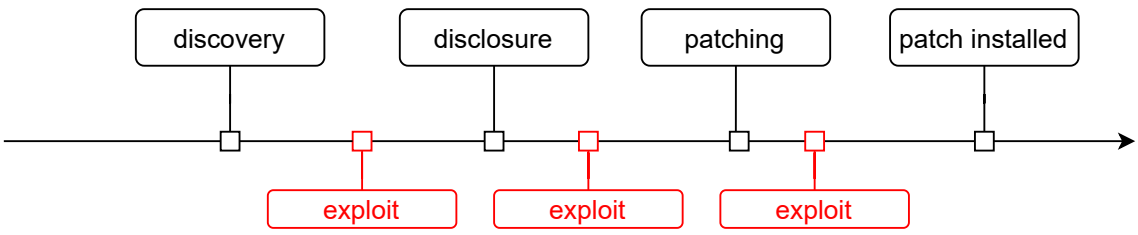


Figure 4.1: An illustration of events in a vulnerability lifecycle started by the initial discovery of a vulnerability. Note that exploits can happen at any time during the lifecycle.

words, a software product is assembled by integrating smaller software components that are developed either by developers inside or outside of the product team. Therefore, the vulnerability could propagate through OSS projects across OSS ecosystems (Ohm et al., 2020), and so must any patch for it. To this end, once a vulnerability is identified, miscreants can exploit it to attack several victim software systems at the same time.

In open-source software ecosystems, due to the emergence of software supply chains and bug bounty programs, fixing a vulnerability may require the involvement of multiple parties associated with the vulnerability, rather than just the entity who detected a vulnerability and the impacted provider. The involvement of multiple parties complicates the process of fixing vulnerabilities and increases coordination and disclosure challenges. `First.org` is one of the leading organizations in incident response and includes a variety of computer security incident teams from governments, corporations, and research organizations. To deal with the challenges of patching vulnerabilities in an ecosystem, `first.org` suggests practices for multi-party vulnerability coordination and disclosure (First.org, 2021). These practices indicate that a vulnerability should be reported to the upstream project responsible for developing the vulnerability fix, which should then propagate the fix to downstream ecosystems. The reason for this is that developers in the upstream projects are familiar with their own projects and are experts of the vulnerable source code (Ma et al., 2017), hence they should be able to fix the vulnerability in a timely manner to reduce the exposure time of the vulnerability.

While a vulnerability fix is more likely to happen upstream, downstream ecosystems or their users could also come up with fixes themselves. For example, vulnerability CVE-2014-3207 was fixed by an upstream developer who sent a fix to the oss-security community by email.[1] In order to streamline such early detection and distribution of available fixes and workarounds downstream, several distributions in the Linux ecosystem, including Debian and Fedora, share a mailing list for handling security issues (*e.g.*, reports, discussions, notifications, *etc.*) (OSS-security.org, 2021).

Prior work investigated the lifecycle of a vulnerability in software projects (Shahzad et al., 2012; Frei et al., 2006, 2008). For example, Shahzad et al. (2012) leveraged public documentation from vulnerability databases to analyze the evolution of disclosure, exploitation, and patching behaviors. Prior studies also surveyed the patch deployment process, such as the timelines of vulnerability fix patches (Li and Paxson, 2017), the adoption speed of patches from users (Nappa et al., 2015), and the reliability of vulnerability fix patches (Huang et al., 2016). However, prior work did not investigate how developers work together to fix vulnerabilities across OSS ecosystems with multi-party coordination.

In this chapter, we identify the vulnerability fixing process and divide it into three phases from the maintainer perspective (*i.e.*, awareness, vulnerability fix integration, and vulnerability fix testing) and two phases from the user perspective (*i.e.*, user unawareness and users waiting for a fix) for two popular Linux distributions (*i.e.*, Debian and Fedora). In particular, we study 21,752 and 17,434 security vulnerabilities, *i.e.*, unique CVE-IDs, in Debian and Fedora, respectively, to compare the duration of each phase in the process from both perspectives across the studied distributions. We also investigate the strategies that maintainers use to fix the common vulnerabilities

[1] https://seclists.org/oss-sec/2014/q2/225

across the studied distributions. In the following, we list our research questions along with the key results:

RQ1: How fast are security vulnerabilities fixed within the studied distributions?

This first RQ analyzes to what extent distribution maintainers react to vulnerabilities and how fast users get the information (*e.g.*, fixes, advisories) related to vulnerabilities in the vulnerability fixing process in Linux distributions. In general, maintainers in Debian and Fedora take a median of 20 and 27 days to fix vulnerabilities, respectively, from when the security team creates a vulnerability report to when the fix is released. This time includes the time taken by Debian and Fedora maintainers to integrate vulnerability fixes, which happens within a median of 1 week. From the user perspective, users are unaware of any existing vulnerability for a median of 17 days, then wait for a median of 0.2 days to get vulnerability fixes.

RQ2: What are the characteristics of common security vulnerabilities (CSVs) across the studied distributions?

Since a vulnerability can affect several distributions at the same time, the second RQ aims to characterize security vulnerabilities that have been reported in both studied distributions (CSVs). CSVs make up 74% of Debian vulnerability reports, and 92% of Fedora's. We find that the CSVs are easy to exploit (median exploitability score of 8.6), though their severity level is medium (median base score of 5.0), based on the Common Vulnerability Scoring System CVSS 2.0. Testing fixes for CSVs requires a median of 4.3 days longer than integrating fixes for CSVs.

RQ3: How do developers fix common security vulnerabilities (CSVs) across the studied distributions?

Given the prevalence of CSVs, this RQ investigates the collaborative strategies of fixing CSVs across distributions by a qualitative study on a representative random sample of 345 fixed CSVs (confidence level = 95%, confidence interval = 5%) and several quantitative analyses. From the qualitative study, we found that distribution maintainers have five main types of sources from which they become aware of vulnerabilities, *i.e.*, upstream, Debian, Fedora, upstream projects, or others, and seven main types of strategies for fixing the identified vulnerabilities, such as in-house vs. external fix development or the usage of embargoes. Furthermore, upstream projects were found to develop vulnerability fixes for the majority of CSVs in Debian (58%±5%) and Fedora (59%±5%), and to distribute the fixes to downstream distributions. Ironically, from the quantitative analyses, the availability of a peer fix before maintainers start to analyze the vulnerability does not correlate with faster integration of the fix.

Maintainers in the two studied distributions dedicate substantial effort to securing their distributions, such as integrating vulnerability fixes in a median of one week after vulnerabilities are reported. Our results suggest that the studied distributions collaborate with each other and with upstream after the discovery of vulnerabilities, while they integrate and test vulnerability fixes in parallel.

The chapter is organized as follows. Section 4.2 provides the background of vulnerability management process in Linux. Section 4.3 describes prior related work to our study. Section 4.4 describes how we design our study and obtain our studied dataset. Section 4.5 answers our research questions. Section 4.6 discusses our findings and

their implications. Section 4.7 discusses the threats to the validity of our study. Finally, Section 4.8 concludes the work.

4.2 Background

In this section, we briefly introduce the Common Vulnerabilities and Exposures (CVE) system and Common Vulnerability Scoring System (CVSS) as well as the vulnerability management process in Linux distributions.

4.2.1 Common Vulnerabilities and Exposures (CVE) and Common Vulnerability Scoring System (CVSS)

The Common Vulnerabilities and Exposures (CVE)[2] platform is one of the most popular platforms for vulnerability registration. The CVE system stores a large number of common software and hardware vulnerabilities and reports. More than 100 major software providers, security companies and research organizations form the CVE Numbering Authorities (CNAs), which are responsible for the vulnerability disclosure process. When a vulnerability is discovered, a unique identifier, *i.e.*, CVE-ID, is issued by one of the CNAs for the vulnerability. Information about the vulnerability, such as the description, symptoms and references, are then posted on the CVE website as a CVE report. In general, a CVE-ID is usually assigned several days up to months before its mitigation (*e.g.*, fix, security advisories) is made public.

The Common Vulnerability Scoring System (CVSS)[3] is an open standard for assigning scores to a vulnerability to indicate its threat level. CVSS scores are widely used by

[2] https://cve.mitre.org/
[3] https://nvd.nist.gov/vuln-metrics/cvss

many prior studies (Joh and Malaiya, 2011; Fruhwirth and Mannisto, 2009; Shahzad et al., 2012; Zhang et al., 2021) and large organizations and companies (*e.g.*, Computer Emergency Response Team (CERT), Cisco) to communicate the severity of the vulnerabilities found in their products. A CVSS score consists of scores in three groups: base, temporal and environmental, each group consisting of a set of metrics. The base score represents the innate characteristics (*e.g.*, access complexity, exploitability) of a vulnerability to indicate its severity. For instance, a base score in the range of 4.0 to 6.9 indicates medium severity. The base score of a vulnerability is computed by six metrics grouped into two aspects: exploitability, *i.e.*, how the vulnerability is exploited, and impact, *i.e.*, the extent of victim system losses if the vulnerability is exploited.

The temporal and environmental scores capture dynamics and subjective information. The temporal score represents the characteristics of a vulnerability that change over time, such as the availability of exploit code and techniques, disclosed remediation plans of a vulnerability, and confirmation of a vulnerability. The environmental score represents the characteristics of a vulnerability that are relevant to a particular environment, such as damage or potential losses to productivity, the proportion of vulnerable systems, and the extent of impact. Note that we use CVSS 2.0 as CVSS 3.0 (released in 2015) and 3.1 (released in 2019) are not available for the majority of our data. Our dataset consists of vulnerabilities whose CVE-ID was issued between Jan. 2007 and Dec. 2019 (see Section 4.4).

4.2.2 Vulnerability Management Process in Linux Distributions

Vulnerability management plays an important role in Linux distributions because they have a large user base, including individuals who use them on desktops and companies

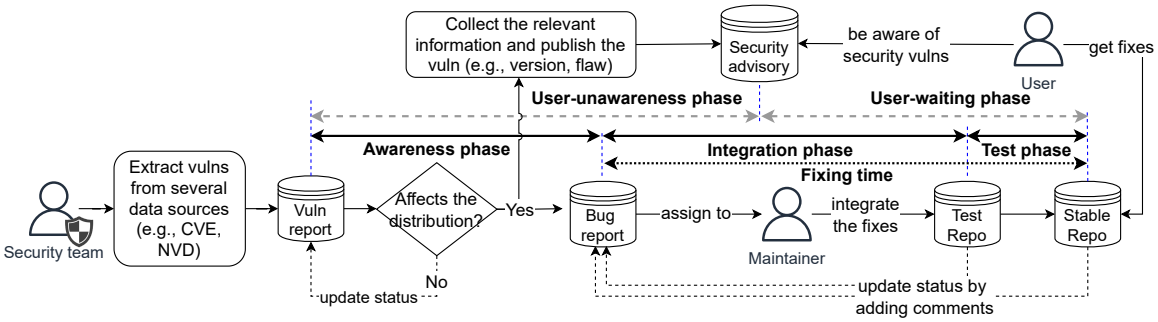


Figure 4.2: Overview of vulnerability management in a given distribution. The black solid and grey dashed arrow lines represent the periods of the five phases of tracking a vulnerability from the maintainer and user perspectives, respectively. To fix a vulnerability in the distribution, the required time (*i.e.*, the fixing time) is the sum of the integration and test periods (the black dotted arrow line). Note that the disclosure time of security advisories for end users (*i.e.*, when the user unawareness phase ends) could happen at any time during the maintainer awareness/integration/test periods.

who use them on servers. In a Linux distribution, security vulnerabilities are addressed by the collaboration between the security team, maintainers who integrate and maintain packages from third-party providers, *i.e.*, upstream projects, and upstream developers. The role of the security team is to track vulnerabilities that affect the packages in their distribution.

Figure 4.2 represents the typical process of vulnerability tracing in the studied Linux distributions. We derive the process from the official documentation in Debian[4] and Fedora[5] by an open-coding approach. The 1st author derived the initial version of the process, discussed this version with the 2nd and 3rd authors, then iteratively resolved any disagreements. We identified the process from two perspectives, *i.e.*, **maintainers** and **users**, and separated the process into three and two phases, respectively. Distribution maintainers are developers who integrate upstream projects into the distribution

[4] https://security-team.debian.org/security_tracker.html

[5] https://fedoraproject.org/wiki/SecurityBasics#Subscribing_to_Security_Announcement_Services

following the distribution's policy, and maintain the integrated upstream projects (*e.g.*, synchronize fixes from upstream, forward bugs to upstream) for end users. Users are stakeholders interested in the management of security vulnerabilities by a distribution. They might be technical experts or system administrators in companies, security specialists, developers in peer distributions, end users of the distributions, etc.

From the maintainer perspective, the three phases consist of awareness of vulnerabilities, integration of vulnerability fixes and testing of vulnerability fixes. The awareness phase starts when a distribution is made aware of a vulnerability through a report in the vulnerability repository, after which the security team creates corresponding bug reports for the vulnerability, one per affected package. Note that a vulnerability either can be found by the security team itself or the security team gets a notification of the vulnerability from somewhere else. Since the awareness phase represents the time taken by the security team to evaluate the extent to which a vulnerability affects the team's distribution, a smaller awareness period implies that the information regarding the vulnerability is sufficient for the security team to more quickly investigate the packages affected by the vulnerability, or the security team gets access more quickly to an existing investigation report, streamlining the start of vulnerability analysis.

During the awareness phase, when a vulnerability is discovered without a fix, the security team can choose to embargo the vulnerability to reduce its impact on the distribution. Embargo of a vulnerability means that the vulnerability will not be disclosed for a period of time to allow the affected software providers to develop a fix. When a vulnerability is under embargo, a small group of people (*e.g.*, the reporter, the developers of the affected software) coordinate to develop and test a fix together. In our studied distributions, Debian avoids embargoes "*since coordination in private tends to*

cause a lot of friction and makes it difficult to involve the right subject matter experts," (Debian.org, 2021e) while Fedora applies the embargo policy of Red Hat that keeps vulnerabilities private for a certain period (Fedora.org, 2021b).

The integration phase is calculated from the creation time of a bug report for the vulnerability to when a vulnerability fix has been integrated and pushed into the test repository. Since the vulnerability may affect several distributions or other software systems, the vulnerability fix can be developed by a wide range of involved people, *e.g.*, upstream developers, and maintainers, security experts. The integration period also allows maintainers to integrate a vulnerability fix that is compatible with the other packages in their distribution (Adams et al., 2016; Foundjem and Adams, 2021).

Lastly, the vulnerability fix needs to be tested before it is released into the stable repository, which we refer to as the vulnerability test phase.

The two phases from the user perspective are unawareness of vulnerabilities and waiting for vulnerability fixes. The security team in a distribution publishes a security advisory that describes the related information (*e.g.*, the names of the affected packages, how to fix them) for a particular vulnerability to users. In our studied distributions, security advisories are usually posted in a particular mailing list where users get proactive notifications by subscription.[6] We define the period before a security advisory is published as the user-unawareness period and the period after that as the user-waiting period. The user-waiting period represents the time for users to get a vulnerability fix, once they know that the vulnerability exists.

[6] https://www.debian.org/security,https://fedoraproject.org/wiki/SecurityBasics#Subscribing_to_Security_Announcement_Services

4.3 Related Work

We discuss prior empirical studies on (1) vulnerability life cycles and (2) vulnerability fix development.

4.3.1 Vulnerability Lifecycles

The vulnerability lifecycle describes the events that characterize the period of a vulnerability from its discovery until all victim systems have adopted a fix patch (Joh and Malaiya, 2011). Most of the existing research has focused on one or several particular events of the vulnerability lifecycles. For example, *zero-day* attacks, the damage caused by exploiting unpatched vulnerabilities, are believed to represent a large amount of targeted attacks (Anderson, 2002; Bilge and Dumitraş, 2012; Wang et al., 2019). Algarni and Malaiya (2014) identified several vulnerability markets where discoverers offer the undisclosed vulnerabilities to buyers for a monetary reward. Joh and Malaiya (2011) formalize the risk measurement in each stage of a vulnerability lifecycle by a stochastic model based on CVSS scores to help project managers make decisions about whether to apply the patches. Different from prior studies that focus on particular events in the vulnerability lifecycle of individual projects, we focus our study on the whole fixing process from the user and maintainer perspectives in the Linux ecosystem.

Prior studies also investigated the evolution of the events (*e.g.*, how long after its discovery is a vulnerability disclosed) in a vulnerability lifecycle. Frei et al. (2006) aggregated data from several public data sources (*e.g.*, the National Vulnerability Database (NVD)[7]) to analyze the distributions of the discovery, exploit, and patch-availability date with respect to the disclosure date of a vulnerability. For example, they found

[7]https://nvd.nist.gov/

that the trend of zero-day attacks has increased. Shahzad et al. (2012) extended (Frei et al., 2006)'s work and investigated the evolution of events (*e.g.*, disclosure trend, exploitation behaviors) in the vulnerability life cycle from three public data sources (*i.e.*, NVD, open-source vulnerability database (OSVDB) and the data source from Frei et al. (2006)).

However, the public data sources have been reported to have an undetectable bias due to the various disclosure methods from different software providers (Schryen, 2009; Christey and Martin, 2013). For example, the two studied distributions disclose vulnerabilities by publishing security advisories (Debian.org, 2021e; Fedora.org, 2021b). To avoid this problem, our work extracts multiple types of data (*i.e.*, vulnerability reports, bug reports and security advisories) from a particular software ecosystem and aggregates these data by CVE identifiers. In addition, we derive the lifecycle of a vulnerability in a particular software ecosystem instead of individual software projects.

4.3.2 Vulnerability Fix Development

Adopting security patches is a common practice to prevent the present vulnerabilities from being exploited. This practice involves a number of processes, such as impact analysis of the patch, integration of the patch, and resolving any potential side-effects caused by the patch. Some researchers analyzed the timelines of security patches by software projects (Frei et al., 2006; Shahzad et al., 2012). For example, Shahzad et al. (2012) observed that the trend of patch availability before or on the disclosure of a vulnerability had decreased before 2005 but improved after 2008 (until 2011). The authors also showed that closed-source providers release at least 70% of vulnerabilities on or

before disclosure dates.

Another major line of research has focused on the dynamics of patch development (Zaman et al., 2011; Huang et al., 2016; Ozment and Schechter, 2006). Ozment and Schechter (2006) observed that the density of vulnerabilities is in the range of 0 to 0.033 vulnerabilities per thousand lines of code in the OpenBSD system. Huang et al. (2016) proposed a model to generate security workarounds rapidly for vulnerabilities in five projects. The model generated workarounds for remediating 75% of vulnerabilities but possibly at the expense of loss of functionality. Different from the aforementioned studies that only focus on the stage of patch development in a few projects, our study investigates the entire fixing process of a vulnerability in software ecosystems (*i.e.*, Linux distributions). For example, our findings suggest that Debian and Fedora integrate vulnerability fixes of the majority of CSVs that are from vulnerable upstream projects.

Li and Paxson (2017) conducted a large-scale study of security patches on 4,080 commits across 682 open-source projects for 3,094 vulnerabilities. They identified that the changes for security patches in source code are more likely to be made by the project's own contributors, compared to ordinary (non-security) bug fixes made by external contributors. They also showed that vulnerabilities reside in source code for years prior to remediation. Li and Paxson (2017) studied vulnerability fixes in a large number of OSS projects that may/may not have relations with each other, while in our work we study OSS projects that form a self-sustainable software ecosystem. In addition, since collaboration is the nature of OSS development for successful software (Kakimoto et al., 2006; Duc et al., 2011), our study investigates the collaboration of patch development across maintainers in Linux distributions.

4.4 Study Design

This section presents the approach for our empirical study to understand how security vulnerabilities are managed in Linux distributions. RQ1 studies the duration of the five phases for fixing vulnerabilities according to the two perspectives and compares them. RQ2 investigates the characteristics of vulnerabilities that affect the two studied distributions. Finally, RQ3 studies the strategies that maintainers use to fix vulnerabilities that are shared by both analyzed distributions.

4.4.1 Subject System Selection

We select Debian and Fedora for several reasons. First, Linux distributions form families based on their package file formats. Deb-based and Rpm-based families are two notable Linux distribution families, with Debian and Fedora each belonging to a different family.[8] Second, within a family, ancestor relations exist based on the provenance of a distribution's packages. For example, Debian is the root of the Deb-based family of distributions, with 44 "child" distributions extending and customizing Debian's packages, each of which can have its own child distributions. Fedora is the root distribution of the Rpm-based distribution family, with 17 children. Despite being root, both Debian and Fedora are popular Linux distributions by themselves, according to the well-known `distrowatch.com` portal and considered amongst the most popular end-user Linux distributions,[9] and best multi-purpose Linux distributions.[10] Prior work leveraged Debian's and Fedora's data on studying security vulnerabilities, such as

[8] https://en.wikipedia.org/wiki/List_of_Linux_distributions
[9] https://www.zdnet.com/article/the-five-most-popular-end-user-linux-distributions/
[10] https://itsfoss.com/best-linux-distributions/

training vulnerability detection models (Russell et al., 2018; Harer et al., 2018), building vulnerability discovery tools (Alhazmi and Malaiya, 2008), security vulnerability assessment (Ristov et al., 2013) and studying the impact of vulnerabilities (Yilek et al., 2009).

Furthermore, Fedora is a not-for-profit version of Linux that is primarily sponsored by Red Hat.[11] For example, Fedora follows the "upstream first" policy[12] from Red Hat to do software development (*e.g.*, forwarding bugs to upstream, fixing bugs) side-by-side with upstream developers in the upstream projects. On the other hand, Debian, while still trying to get bug fixes merged upstream, is a purely open-design Linux distribution and has its own community of contributors (Debian.org, 2021c). As such, there are differences between the strategies of upstream package management in Debian and Fedora in terms of where bug fixes are developed. For these reasons, we are interested in comparing the strategies of vulnerability management between the two distributions.

4.4.2 Data Collection

Figure 4.3 presents an overview of our study approach. To extract the phases stated in Section 4.2.2, we leverage vulnerability reports, bug reports and security advisories to measure the duration of each lifecycle phase for each vulnerability. We archived our scripts and uploaded them to GitHub.[13] For a given vulnerability, we use its unique

[11] https://getfedora.org/sponsors/
[12] https://www.redhat.com/en/blog/what-open-source-upstream
[13] https://github.com/SAILResearch/suppmaterial-22-justina-vulnerability_management_in_linux_distributions

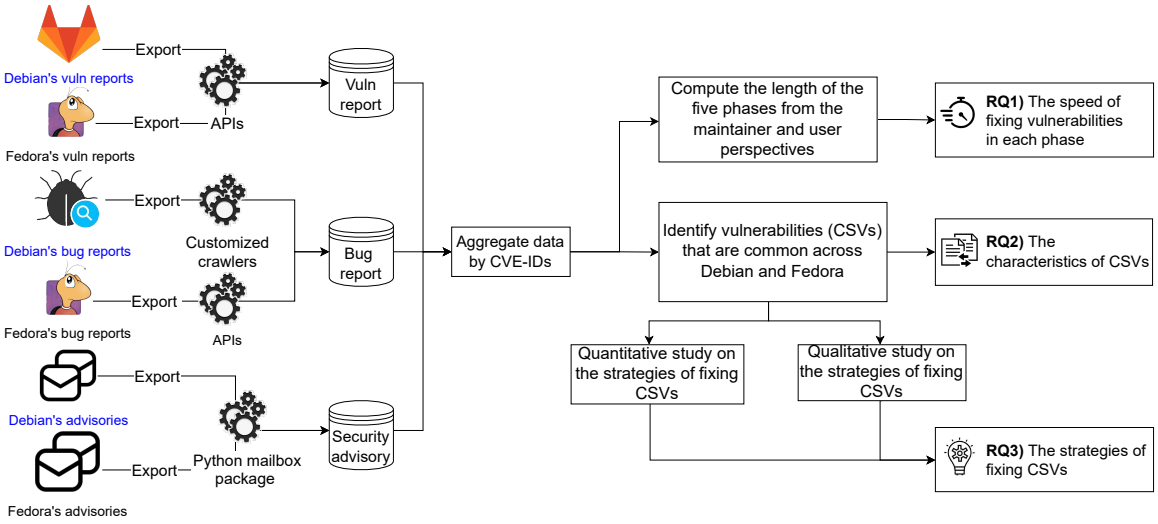


Figure 4.3: Overview of our study approach.

CVE-ID to identify its corresponding reports and advisories. A vulnerability report contains the CVE-ID, the creation date, description, affected packages and package versions, links to its bug reports, and references to other information (*e.g.*, the fix patch), for example, the vulnerability report of CVE-2018-7225.[14] A security advisory is issued for one or several CVE-IDs and consists of the affected packages, remediation plans (*e.g.*, security updates), and reference reports (*e.g.*, National Vulnerabilities Database (NVD) reports).

Table 4.1 presents the software systems and mailing lists for which we obtain the three types of data sources in each studied distribution. We extract data from Bugzilla (Fedora) and GitLab (Debian) using their relevant APIs, access security advisories by the mailbox package[15] in Python, and customize crawlers for Debian's Debbugs repository. We only consider security vulnerabilities with a valid CVE-ID according to the

[14] https://security-tracker.debian.org/tracker/CVE-2018-7225
[15] https://docs.python.org/3/library/mailbox.html

Table 4.1: The software systems and mailing lists where we extract the three kinds of repositories in the studied distributions.

Distribution	Vulnerability reports	Bug reports	Security advisories
Debian	Gitlab[1]	Debbugs[3]	Mailing list[4]
Fedora	Bugzilla[2]	Bugzilla[2]	Mailing list[5]

[1] https://salsa.debian.org/security-tracker-team/security-tracker
[2] https://bugzilla.redhat.com/
[3] https://www.debian.org/Bugs/
[4] debian-security-announce@lists.debian.org
[5] package-announce@fedoraproject.org

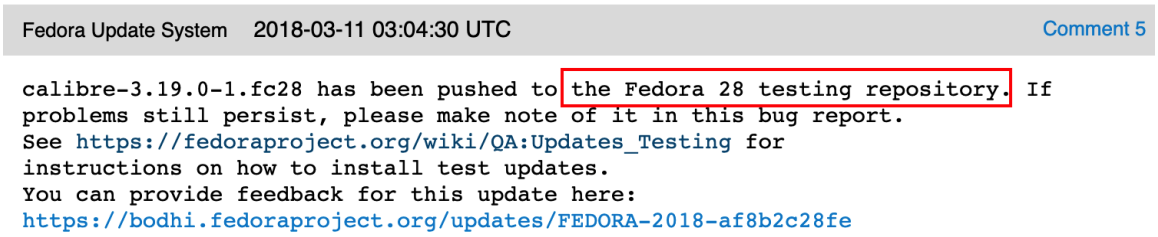


Figure 4.4: An example of a comment in a Fedora bug report that mentions that a vulnerability fix has been pushed into the test repository.

CVE system, as discussed in Section 4.2.1. Therefore, we filter out 5,138 security vulnerabilities with an invalid CVE-ID. For example, some CVE-IDs should not have been assigned (*e.g.*, not a real vulnerability or should remain private), some CVE-IDs refer to the same vulnerability, or one CVE-ID actually refers to two different vulnerabilities, etc.[16] This leaves us with a total of 21,752 and 17,434 security vulnerabilities, *i.e.*, unique CVE-IDs, in Debian and Fedora, respectively, whose CVE-ID was issued between Jan. 2007 and Dec. 2019.

From the maintainer perspective, the duration of the awareness phase represents the time difference between the creation time of a vulnerability report and the creation time of its associated bug report. The duration of the integration phase represents the

[16] https://cve.mitre.org/cve/list_rules_and_guidance/correcting_counting_issues.html

time difference between the creation time of the bug report and the time when the vulnerability fix is ready for testing. To identify when a vulnerability fix is ready for testing, *i.e.*, in the test repository, or available to users, *i.e.*, in the stable repository, we extract the corresponding automated messages in the comments of a bug report. For example, Figure 4.4 indicates that the vulnerability fix for CVE-2018-7889 has been pushed into the test repository on March 11th, 2018.[17] To validate the existence of the automated messages, we randomly select a set of 50 fixed vulnerabilities in each of the studied distributions and study their respective bug reports, *i.e.*, a total of 100 bug reports. We observe that all these bug reports contain such automated messages. The duration of the test phase represents the time difference between when the vulnerability fix is ready for testing and when it is available to users.

From the user perspective, the duration of the user-unawareness phase reflects the time difference between the creation time of a vulnerability report and when its security advisory is published. The duration of the user-waiting phase reflects the time difference between when the security advisory is published and when the vulnerability fix is available to users.

4.4.3 Methodology

In this section, we present why and how we leverage the vulnerability data for our RQs and how we perform the manual study (RQ3).

[17] https://bugzilla.redhat.com/show_bug.cgi?id=1553919

RQ1: How fast are security vulnerabilities fixed within the studied distributions?

Motivation: The goal of this research question is to measure an upper bound of how much time maintainers spend in each phase while fixing security vulnerabilities in a given distribution (Figure 4.2). Such an investigation will give insights into the maximum cost these phases require, both from the maintainer and user perspectives. Comparing the time taken in each phase can provide useful indications of the practices that maintainers use to fix vulnerabilities.

Approach: We study the duration of the five phases from the **maintainer** and **user** perspectives, as discussed in Section 4.2.2. From the maintainer perspective, we compute the number of days in each phase for each vulnerability and classify vulnerabilities into 6 classes based on the duration (*i.e.*, <0, 0-1, 1-7, 7-30, 30-90 and 90+ days). These 6 classes correspond to instantaneous, same-day, same-week, same-month, same-quarter, and more-than-one-quarter, respectively. To compare the speed of fixing vulnerabilities in the studied distributions, we analyze a cumulative density plot to show the proportion of vulnerabilities over time. For example, we study how fast maintainers integrate vulnerability fixes in the integration period. Since Debian is a purely community-driven distribution and Fedora is a not-for-profit distribution backed by Red Hat, we compare the speed in each period to understand whether they follow different strategies for vulnerability management.

From the user perspective, similar to the maintainer perspective, we compute the number of days for the two phases, *i.e.*, the user-unawareness and user-waiting phases, in the studied distributions. We classify the phases into the same 6 classes to understand how long users are unaware of a vulnerability and how much time it takes to get a fix. Since vulnerability exploits usually have a large impact (*e.g.*, data leakage, service

temporarily unavailable) on victim software systems (Klinke and Renn, 2002), we compare the duration of the two phases between the studied distributions to understand whether they have different practices of vulnerability management.

RQ2: What are the characteristics of common security vulnerabilities (CSVs) across the studied distributions?

Motivation: Developers may take more than two years to discover and fix vulnerabilities in software projects (Ozment and Schechter, 2006). During this period, the vulnerabilities might propagate from one software ecosystem that has packaged the affected software project to another (Li and Paxson, 2017). In this case, several software ecosystems (*i.e.*, Linux distributions) are affected by a given set of vulnerabilities. As such, given the reliance of any open (and even closed) source project on a supply chain of software dependencies, such vulnerabilities risk impacting not only direct end users of a distribution package, but large cross-sections of the software world.

On the plus side, such spreading of vulnerabilities also increases the chance of vulnerability detection and opportunities for collaboration on fixing vulnerabilities (which is studied in RQ3), *cf.* Linux's Law ("*given enough eyeballs, all bugs are shallow*") (Raymond, 1999). Given that such collaboration transcends project, ecosystem or even company borders, it is important to understand the dynamics of such collaboration, as well as when such collaboration happens (e.g., prioritization of vulnerabilities to fix collaboratively). For these reasons, we are interested in the characteristics of the security vulnerabilities that affect several distributions, and we investigate whether maintainers follow different strategies to fix these security vulnerabilities.

Approach: We analyze the characteristics of common security vulnerabilities (CSVs).

To study CSVs, we first identify security vulnerabilities whose CVE-ID exists across the two studied distributions. For example, CVE-2014-0160 is an example CSV that has affected both Debian[18] and Fedora[19] via the *OpenSSL* package.

We study the prevalence of CSVs and their threat level across the studied distributions. To obtain the prevalence, we calculate the number of CSVs that have been reported and fixed, and the number of their corresponding security advisories. To evaluate the impact, we compare the threat level of CSVs based on their CVSS scores, as discussed in Section 4.2.1. Similar to RQ1, we investigate the vulnerability fixing process of CSVs w.r.t. the five phases from the two perspectives in Figure 4.2. We calculate the number of days in each phase and compare these numbers of CSVs between the studied distributions.

RQ3: How do developers fix common security vulnerabilities (CSVs) across the studied distributions?

Motivation: In a software project, developers often collaborate for security vulnerabilities instead of relying on hero-centric contribution, which is different from ordinary bugs (Wang and Nagappan, 2019). Given the large number (i.e., 16,070) of CSVs across distributions and the observations in RQ2 that maintainers take a longer time to test vulnerability fixes of CSVs than to integrate fixes, we are interested in how maintainers across distributions collaborate when integrating and testing vulnerability fixes in their distributions. Therefore, understanding the collaboration patterns across distributions for fixing CSVs can provide insights for improving current practices and sharing with other ecosystems.

[18] https://bugs.debian.org/cgi-bin/bugreport.cgi?bug=743883
[19] https://bugzilla.redhat.com/show_bug.cgi?id=CVE-2014-0160

Approach: To understand how maintainers collaborate on fixing security vulnerabilities, we focus on CSVs that are marked as fixed by both Debian and Fedora. We select a set of 3,336 **fixed** CSVs in both studied distributions before we conduct a manual study on a representative random sample (confidence level = 95%, confidence interval = 5%) of 345 **fixed** CSVs across the studied distributions. The goal of this qualitative analysis is to identify the flow of strategies, *i.e.*, the actions for fixing vulnerabilities, as well as the prevalence of each strategy. In particular, we leverage the comments in the vulnerability reports, their corresponding bug reports in the distribution, the related bug reports in upstream projects where vulnerabilities could be discussed and fixed, and links to reference web pages in these reports. Based on these comments and links, we compile a catalog of strategies for fixing vulnerabilities, which we have grouped into three types according to their purpose (*i.e.*, awareness, embargo, vulnerability fix integration).

For a given CSV, we first study the strategies of fixing vulnerabilities in the distribution. The first author sorts the post time of each comment in the vulnerability and bug reports to investigate the timeline of strategies through developer discussions, then labels strategies (*e.g.*, report upstream) that are used for fixing the vulnerability. After that, the first author checks the links in the comments to reference web pages and the corresponding upstream reports. These reference links indicate the information relevant to the development of a vulnerability fix, such as a patch, a notification email for the disclosure of the vulnerability, and the reference reports of the vulnerability. Similarly, the first author labels strategies (*e.g.*, propose a patch to upstream) along with (if this data is available) when they happened relative to each other, before structuring them into a timeline. Finally, the first author discusses the strategies with the 2nd

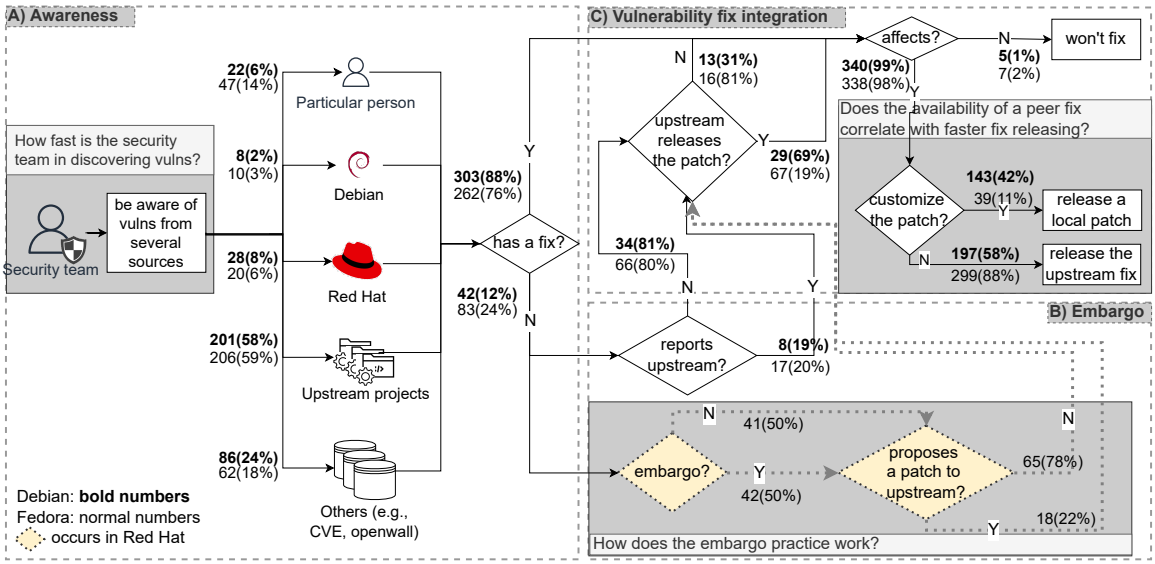


Figure 4.5: The flow chart of the potential strategies that are used in Debian and Fedora for dealing with vulnerabilities. The bold numbers represent the number and proportion of CSVs in each strategy in Debian and the normal numbers are for Fedora. The two grey dotted diamonds and dotted lines represent the strategies related to Red Hat's embargo policy. The three grey dashed boxes group the strategies based on their purpose. The three boxes with grey background represent the quantitative analyses to measure the degree of potential collaboration in the three groups.

and 3rd authors and addresses disagreement. We performed this process iteratively for each CSV and arrived at the strategies summarized in Figure 4.5, along with the numbers (proportions) of the analyzed CSVs using a particular strategy.

We also perform several quantitative analyses to evaluate the three types of strategies of Figure 4.5 (the boxes with grey background) to gain comprehensive insights on potential collaboration across distributions to fix CSVs. We describe the details of approaches for each type below:

(A) Awareness. Since the security team is responsible for discovering vulnerabilities for their distribution, we analyze how fast the security team becomes aware of a vulnerability. We compare the creation time of vulnerability reports for fixed CSVs between

Debian and Fedora to measure the speed of awareness. We leverage the number of bug reports related to vulnerabilities that were created by the security team (*i.e.*, internal developers) to measure the extent of responsibility for the security team.

(B) Embargo. When a vulnerability is first reported to a distribution, the security team could forward the vulnerability to upstream after it assesses the impact of the vulnerability. In such a case, the security team could embargo the vulnerability (*i.e.*, only certain people with permission can read the vulnerability report) to reduce its impact according to the distribution's policy. For severe vulnerabilities, vulnerability fixes are usually discussed privately under embargoes to prevent early disclosure (Ramsauer et al., 2020). To understand how embargoes work in practice, we investigate the embargo policy for CSVs in Fedora (Red Hat), since Debian encourages public disclosure of vulnerabilities even before a fix has been developed.[20]

From the history of a Fedora vulnerability report, we observe that the Fedora security team members add the *"EMBARGOED"* keyword as a prefix in the title of a vulnerability report to indicate that the vulnerability is under embargo. The keyword is removed once the embargo is lifted and this change is recorded in the history of the vulnerability report. For example, CVE-2015-8382 was embargoed for 5 days in Red Hat.[21] Therefore, we study the embargo policy by studying the characteristics of embargoed CSVs and the time taken by the five phases of the maintainer and user perspectives (Figure 4.2).

(C) Vulnerability fix integration. As discussed in Section 4.2.2, vulnerability fixes can be developed by anyone before being integrated into the distributions that contain the affected packages. We investigate the extent to which maintainers have access to

[20] https://www.debian.org/security/disclosure-policy
[21] https://bugzilla.redhat.com/show_activity.cgi?id=1187225

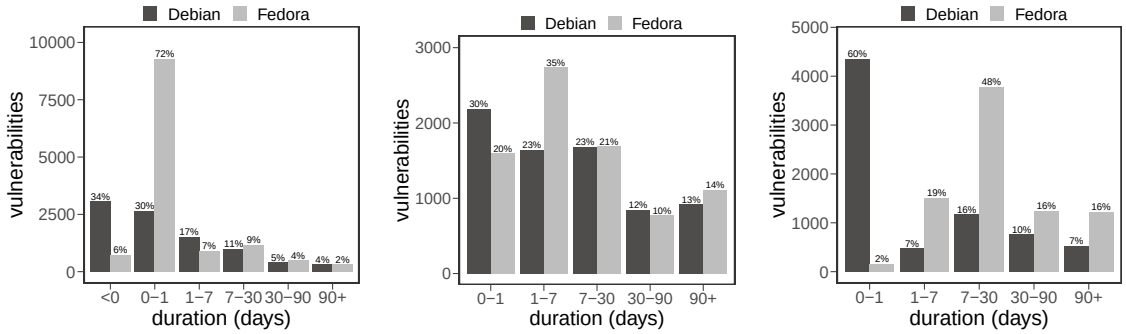
a vulnerability fix generated by the peer distribution (the other studied distribution), which we refer to as *"peer (vulnerability) fixes"*. For a CSV, we identify when such a peer fix is available, *i.e.*, before or during integration or not at all, by comparing the creation and fixing time of a bug report to the release time of the peer fix. To measure whether maintainers across distributions collaborate to integrate vulnerability fixes into their own packages, we hypothesize that distribution maintainers become aware of a peer fix once it is available. For example, Debian released a fix for CVE-2014-8145 on January 3rd, 2015,[22] while the bug for CVE-2014-8145 was reported in Fedora on January 20th, 2015.[23] We conclude that Fedora had access to a peer fix (in Debian) for CVE-2014-8145 before integration.

4.5 Research Questions

RQ1: How fast are security vulnerabilities fixed within the studied distributions?

Results: From the maintainer perspective, Debian and Fedora take a median of 20 and 27 days, respectively, from when the security team creates a vulnerability report to when the fix is released. Figure 4.6 indicates the duration of each phase. In particular, the awareness period accounts for a median of 2 and 0.003 days (4.3 minutes), which indicates a significant difference (Wilcoxon test: p-value $< 2.2e^{-16}$) in Debian and Fedora, respectively, although with negligible effect size (0.14). We will discuss the awareness period in the next finding.

[22] https://bugs.debian.org/cgi-bin/bugreport.cgi?bug=773720
[23] https://bugzilla.redhat.com/show_bug.cgi?id=1184079



(a) Awareness (b) Integration (c) Testing

Figure 4.6: The time taken in the three phases from the maintainers' perspective.

One possible reason for the significant difference between Debian and Fedora is that Fedora is a not-for-profit Linux distribution that is primarily sponsored by Red Hat (as discussed in Section 4.4.1). Furthermore, Red Hat also has a dedicated security team working on vulnerabilities in Fedora. The integration period accounts for a median of 6.1 and 5.3 days (Wilcoxon test: p-value = 0.0001) and the test period accounts for a median of 0 and 15 days (Wilcoxon test: p-value $< 2.2e^{-16}$) in Debian and Fedora, respectively. The differences between Debian and Fedora in both the integration and test periods are significant. The effect size of the integration period is negligible (0.0006), and the effect size of the test period is small (0.24).

The security team in Fedora notifies its maintainers of at least 78% of vulnerabilities within one day after the vulnerability reports were created (i.e., a median of 4.3 minutes), while the security team in Debian takes one week for 81% of vulnerabilities (i.e., a median of 2 days), as shown in Figure 4.6a. The awareness of a vulnerability is the first step of tracking a vulnerability that the security team of a distribution faces (cf. Figure 4.2). The security team can discover a new vulnerability by itself or via other channels, *e.g.*, open-source security communities. Once the security team is aware of the new vulnerability, the team evaluates, assesses and validates whether the packages

in the distribution are affected before assigning the vulnerability to maintainers.

Meanwhile, users report 34% and 6% of vulnerabilities as ordinary bugs to Debian and Fedora, respectively, before their security teams become aware of these vulnerabilities, as shown in Figure 4.6a. For example, a Debian bug, CVE-2017-12424,[24] was reported 3 years before the CVE-ID was assigned, *i.e.*, when the security team became aware of the vulnerability.[25] In a given distribution, a vulnerability report can be created earlier or later than its related bug report, suggesting two fixing processes of vulnerabilities. If the vulnerability report is created earlier, the respective bug report is created for tracking the progress of fixing the vulnerability. By contrast, when a bug report is created earlier, maintainers and the community collaborate and discuss how to fix the bug before the bug is identified as a vulnerability, suggesting the vulnerability is publicly disclosed without a fix. Researchers should take into account how a vulnerability was discovered to study the impact of such vulnerabilities.

Both Debian and Fedora maintainers integrate the vulnerability fixes for the majority (50%) of security vulnerabilities within 1 week of the creation of their corresponding bug reports, and integrate 75% of security vulnerabilities within 1 month. Figure 4.7 shows that Debian maintainers integrate more vulnerability fixes within 3 days compared to Fedora maintainers. After that, Fedora catches up and both Debian and Fedora maintainers integrate vulnerability fixes at the same speed. Since both studied distributions have a large number of packages with complex relations among them, this result indicates that the speed of integrating vulnerability fixes into a distribution is fast, compared to the time of integrating patches into one project, *i.e.*, the Linux Kernel project, which had an increasing trend in the integration time of patches

[24] https://bugs.debian.org/cgi-bin/bugreport.cgi?bug=756630

[25] https://salsa.debian.org/security-tracker-team/security-tracker/-/commit/1d9590647d

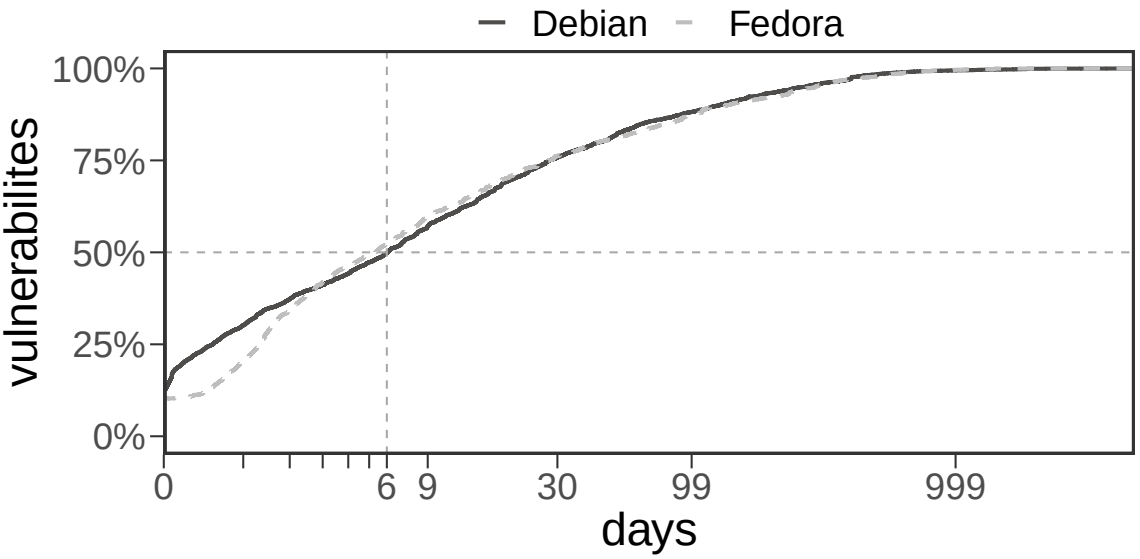


Figure 4.7: Cumulative density plot of the duration of the integration phase of a security vulnerability. Debian and Fedora maintainers take a median of 6.1 and 5.3 days, respectively, to integrate fixes for security vulnerabilities after their bug reports were created.

from 2006 to 2012, from instantly to a month as reported by (Jiang et al., 2013).

Vulnerability fixes for 75% of vulnerabilities in Debian pass testing in 17 days, compared to 44 days in Fedora. Figure 4.8 shows the cumulative distribution of the test period. Vulnerability fixes pass testing in a median of 10 days across Debian and Fedora. The delay in the number of days for a fixed vulnerability to become available for users is shorter than that of an ordinary fixed bug (da Costa et al., 2014). For example, da Costa et al. (2014) observed that the median delay time for fixed bugs was 42 days in the Firefox project (the minimum median delay time among three projects that they studied).

Figure 4.8 and Figure 4.6c indicate that the test period for 60% of vulnerabilities is 0 days in Debian. One possible reason is that Debian does not integrate the results of its continuous integration (CI) system into bug reports. The continuous integration

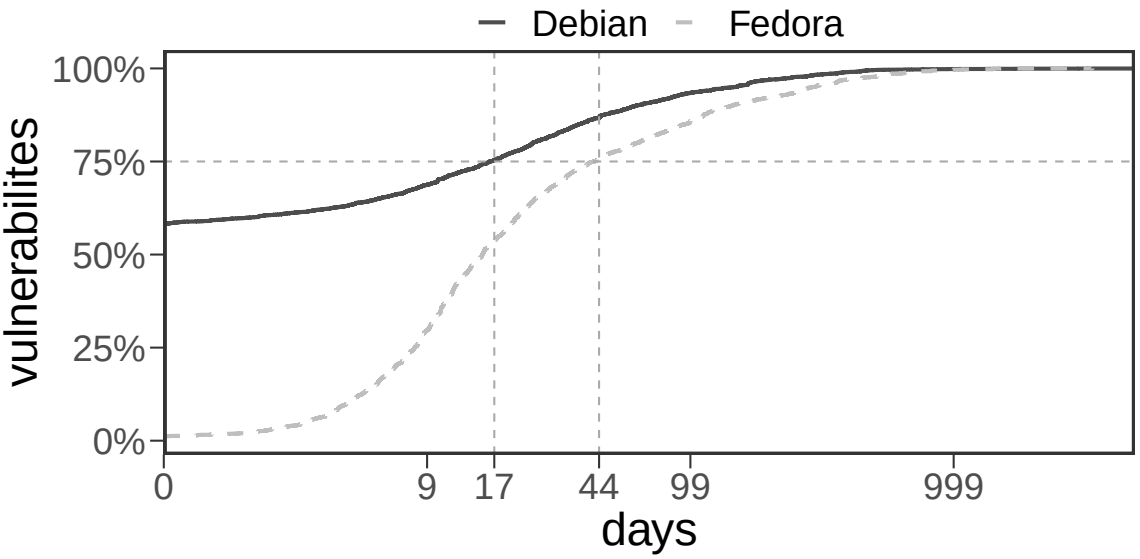


Figure 4.8: Cumulative density plot for the testing phase of a security vulnerability. Debian maintainers test the fixes of 75% of security vulnerabilities in 17 days, while Fedora maintainers take 44 days for testing. 60% of vulnerabilities in Debian have a 0-day test phase.

system in Debian automatically coordinates the automated test against packages in the Debian archive when the packages and/or their dependent packages have been updated.[26] However, since the test logs in the CI system do not include whether the test was triggered by which bug, we are not able to link the testing logs to the corresponding bug. In addition, 53% of these vulnerabilities only affect the development (*i.e.*, unstable) version of Debian, which are less likely to impact users. For example, a denial-of-service flaw in the *Quassel* IRC, *i.e.*, CVE-2015-2779, only affected the development version.[27] Furthermore, 11% out of the 53% vulnerabilities with 0 days of testing only affect the long-term support (LTS) releases that are maintained by the Debian

[26] https://ci.debian.net/doc/
[27] https://security-tracker.debian.org/tracker/CVE-2015-2779

LTS team,[28] which has a different development policy for security vulnerabilities.[29]

The integration phase accounts for the largest proportion of time (*i.e.*, median of 6.1 days, 40%) for tracking a security vulnerability in Debian, while Fedora spends the largest proportion of time (*i.e.*, median of 15.7 days, 58%) on vulnerability fix testing. The number of days in the integration phase in Debian (median of 6.1 days) is similar to the number in Fedora (median of 5.8 days). Figure 4.6b indicates that Debian integrates more vulnerability fixes within one day than Fedora.

Concerning the testing phase, Fedora spends a higher proportion of time (median of 15.7 days) for testing vulnerability fixes, compared to Debian. Figure 4.6c indicates that Fedora takes 7 and up to 30 days to test 48% of vulnerabilities (the largest proportion among the 6 classes). One possible explanation is that the test logs in Debian's continuous integration (CI) system do not link back to the respective bug reports, making it impossible for us to trace CI test results back to possible vulnerability fixes being tested. The CI system executes automated test suites against packages when they and/or their dependent packages have been updated (Debian.org, 2021a). The test logs only include packages that have been tested and their testing status (*e.g.*, pass or fail), but identifiers of bug reports are not included, since a test can be triggered for a variety of reasons (*e.g.*, a feature change, bug fixes in the dependent packages). As a rough estimation, Debian states that it usually takes 10 days to test a security update of a package[30], though it might take longer due to the dependent packages needing testing, or shorter due to the emergency level of releasing a new version of the package, suggesting that Debian's test phase might be longer than the integration phase. In

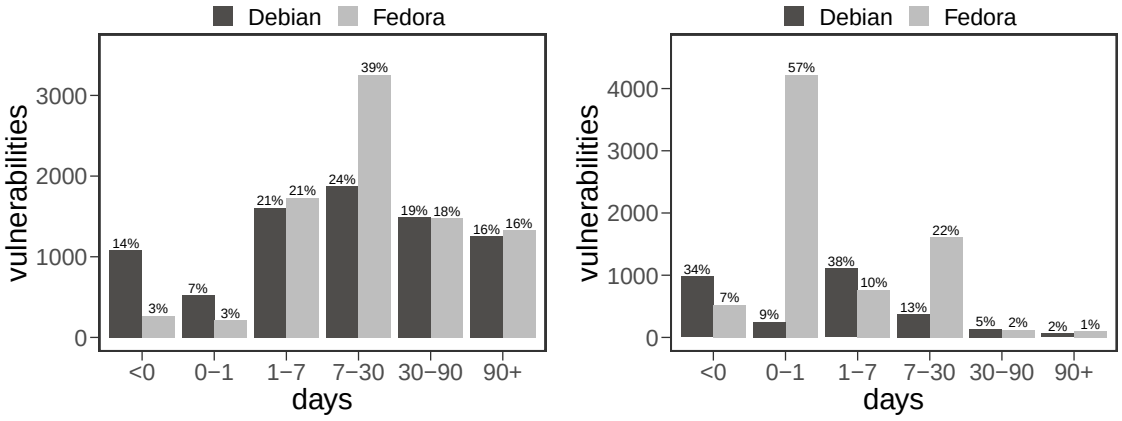
[28] https://wiki.debian.org/LTS
[29] https://wiki.debian.org/LTS/Development#Prepare_security_updates_for_LTS
[30] https://www.debian.org/doc/manuals/securing-debian-manual/ch10.en.html#security-support-testing

comparison with Debian, Fedora integrates its CI and issue report system, and thereby the testing results are recorded back into the bug reports.[31]

From the user perspective, in general, Debian and Fedora publish security advisories, *i.e.*, end the user-unawareness period, in a median of 17 days. Figure 4.9a indicates that Debian's users are aware of the majority (>50%) of vulnerabilities 30 days after the creation of vulnerability reports. Fedora's users are aware of the largest proportion (39%) of vulnerabilities within 7 to 30 days after the creation of vulnerability reports. In order to assess how fast Debian and Fedora publish security advisories, we compare them with GitHub security advisories, since GitHub is one of the most popular OSS code repositories. The speed of publishing security advisories in Debian and Fedora is faster than the speed of publishing security advisories in open-source projects on GitHub, based on the 2020 GitHub security report (Github.com, 2020). That report stated that users usually receive the alert of a security advisory more than 10 weeks after a vulnerability becomes known across GitHub projects. In particular, the speed of publishing security advisories provided by GitHub directly is 1 week, while the speed of publishing security advisories that are imported from other sources into GitHub projects is 20 weeks. A potential explanation could be that the two Linux distributions are self-sustainable with organized efforts, while GitHub projects are loosely coupled in nature, unless they are part of other ecosystems (*e.g.*, npm). As such, our results could be explained by a community-driven effort to manage vulnerabilities in a Linux distribution ecosystem.

Users wait for a median of 0.2 days to get vulnerability fixes across Debian and Fedora, which is 85 times shorter than the user-unawareness period. Figure 4.9b shows that users wait for less than 7 days to get fixes for 81% and 74% of vulnerabilities

[31] https://docs.fedoraproject.org/en-US/fesco/Updates_Policy/



(a) User unawareness (b) User waiting

Figure 4.9: The number of vulnerabilities during the two phases from the user perspective.

in Debian and Fedora, respectively. Fedora releases the largest proportion (57%) of vulnerability fixes within 1 day after the end of the user-unawareness period, dropping to 10% within 7 days and increasing to 22% within 30 days. The priority of fixing vulnerabilities is high due to their potential impact (*e.g.*, market value loss (Telang and Wattal, 2005)) on a large number of users. Figure 4.9b indicates that the fixes for 34% (*i.e.*, 980) of vulnerabilities in Debian were available before Debian published their security advisories, suggesting insufficient awareness about those security vulnerabilities by the Debian's security team. For example, for a Debian bug representing a vulnerability that was reported for the *vlc* package (*i.e.*, CVE-2008-2147),[32] its fix was available for users on May 17, 2008, while its security advisory was published on June 18, 2009.[33]

One possible explanation is that vulnerabilities might be assumed to be ordinary bugs when they were reported. As a consequence, their fixes might be ignored until

[32] https://bugs.debian.org/cgi-bin/bugreport.cgi?bug=480724
[33] https://www.debian.org/security/2009/dsa-1819

they are identified as a vulnerability due to various reasons. For example, bug fixes often are developed relative to a new version of a package that the package maintainers cannot or do not plan to upgrade to. Kula et al. (2018a) identified that 81.5% of the analyzed software systems stick to outdated dependencies. Second, a bug fix could introduce a new bug impacting the software system (Yin et al., 2011), which is challenging for the maintainers to assess before the adoption of bug fixes. Even if the maintainers would like to integrate the bug fix, they usually update the software system periodically for a batch of bug fixes. This is because the integration could require changes to their package's code, and because they need to organize (manual) testing activities. Depending on the workload of the maintainers and the testers, the integration of a batch of bug fixes might be skipped until more fixes are available, at which time all test activities are performed once.

Another explanation is Debian's policy regarding publishing security advisories based on the impact of vulnerabilities to Debian.[34] To avoid cognitive overload, Debian typically includes the security advisory for low-impact vulnerabilities of a particular package into the security advisory of a high-impact vulnerability in that package. As such, the advisory of low-impact vulnerabilities might be delayed until the next high-impact advisory is issued. Moreover, out of the 34% vulnerabilities whose fixes were available before their security advisory in Debian, 55% did not affect any packages in Fedora, since Fedora does not contain those packages for a variety of reasons (*e.g.*, license, patents).[35] Only 1% already had a fix prior to the disclosure of their security advisories in Fedora.

[34] https://security-team.debian.org/security_tracker.html
[35] https://docs.fedoraproject.org/en-US/quick-docs/packagekit-not-found/

In terms of phases from the maintainers' perspective, both Debian and Fedora maintainers integrate vulnerability fixes at a fast pace, *i.e.*, within a median of 1 week, after they are assigned the task. Fedora maintainers spend the longest time to test vulnerability fixes in the vulnerability fixing process. In terms of phases from the user perspective, the user-unawareness period accounts for a median of 17 days and the user-waiting period accounts for a median of 0.2 days.

RQ2: What are the characteristics of common security vulnerabilities (CSVs) across the studied distributions?

Table 4.2: The descriptive statistics of common security vulnerabilities (CSVs) along with their related information. Note that a vulnerability report may map to several bug reports, and vice versa.

Feature	Debian	Fedora
# reported vulns	16,070	16,070
# vulns with bug reports	7,265	12,034
# vulns with advisories	6,760	8,710
# vulns with fixes	5,951	7,310

Results: There are 16,070 CSVs across Debian (74%) and Fedora (92%). Table 4.2 shows the statistics of the number of reported CSVs along with their related information for the two studied distributions. One possible reason for the higher proportion (92%) of CSVs in Fedora is that Debian includes over 89,000 packages[36], while Fedora only includes over 33,000 packages[37]. Another possible reason is that Debian offers its users 3 years of support for each release and 2 years of extra long-term support (LTS),[38]

[36] https://packages.debian.org/stable/
[37] https://src.fedoraproject.org/?page=1&sorting=None
[38] https://wiki.debian.org/DebianReleases

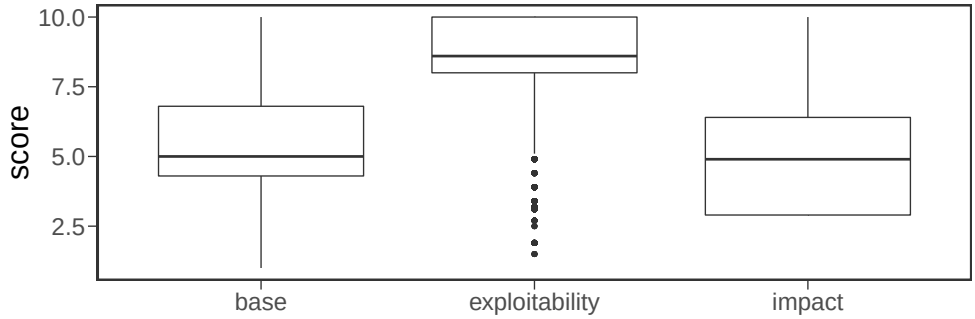
while Fedora supports its releases for approximately 13 months.[39] The longer support period increases the possibility of identifying and having to fix security vulnerabilities specific to Debian in the packages of old releases.

Based on CVSS 2.0, the CSVs can be exploited easily (*i.e.*, median exploitability score of 8.6), though their severity level and impact is medium (*i.e.*, median of both base and impact score of 5.0), as shown in Figure 4.10a. A high exploitability score indicates that attackers can easily exploit these CSVs to infiltrate a victim system, potentially on a large scale. Figure 4.10b indicates that the complexity to exploit 51% of CSVs is low and that of 46% of CSVs is medium. Furthermore, attackers can exploit the majority of CSVs using networks and without any authentication. In terms of the impact of CSVs, Figure 4.10c indicates that the majority of CSVs have partial or no impact on victim systems based on the three metrics, *i.e.*, availability, confidentiality and integrity.

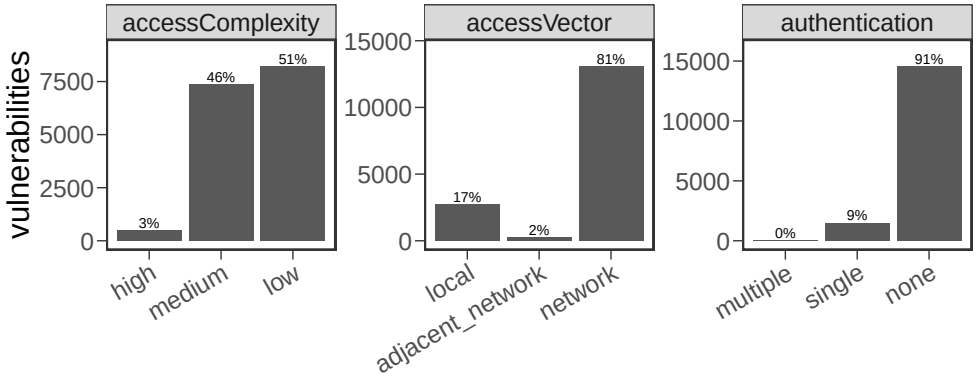
From the maintainer perspective, Debian and Fedora take a median of 20 and 27.1 days, respectively, to fix CSVs, from when the security team creates a vulnerability report to when the fix is released. In particular, the security teams in Debian and Fedora notify their maintainers in a median of 3.6 hours and 2.9 minutes, respectively, after the creation of a vulnerability report. Maintainers in Debian and Fedora take a median of 6.1 and 5.5 days to integrate fixes, respectively, and a median of 0 and 15 days to test fixes, respectively. These results are in line with the result in RQ1 (Figure 4.6).

Figure 4.11 indicates the distribution of CSVs in each of the three phases. The security teams in Debian and Fedora notify their maintainers of 81% and 84% of CSVs,

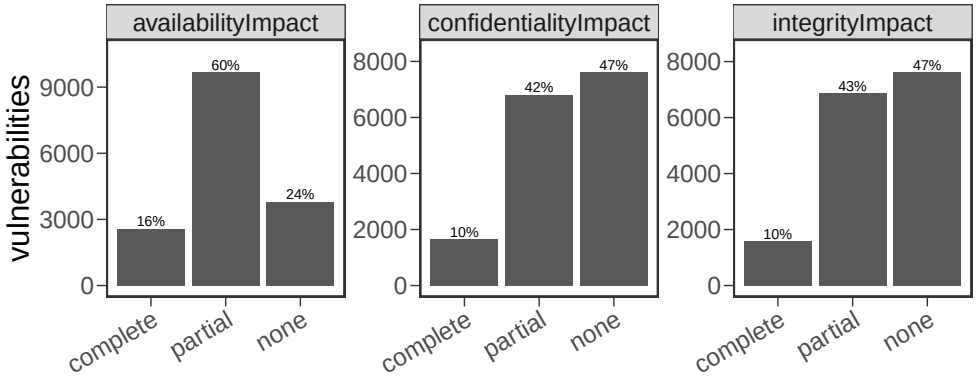
[39] https://fedoraproject.org/wiki/Fedora_Release_Life_Cycle



(a) The majority of CSVs are medium-severity vulnerabilities (median base score of 5), but they are easy to be exploited (median exploitability score of 8.6).

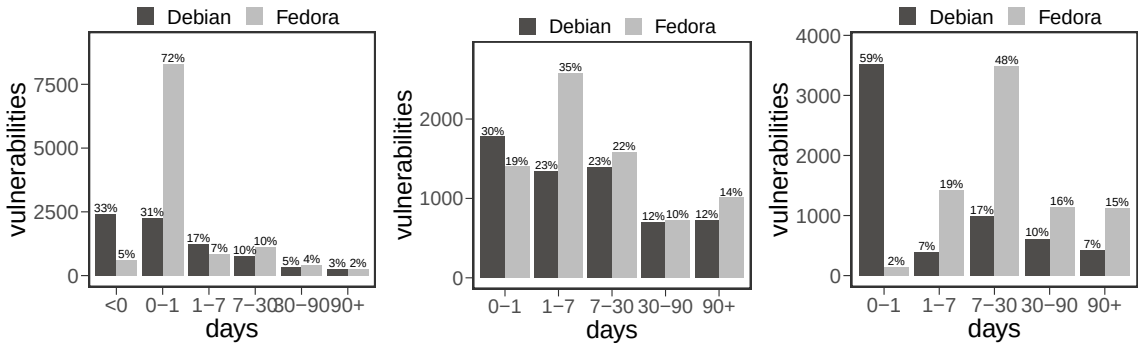


(b) The majority of CSVs are easy to be exploited, due to low complexity, remote exploits and no authentication.



(c) The majority of CSVs that have been exploited have partial or no impact.

Figure 4.10: The (a) high-level CVSS 2.0 scores for CSVs, and a breakdown for (b) exploitability and (c) impact scores.



(a) Awareness (b) Integration (c) Test

Figure 4.11: The duration of the three phases for CSVs from the maintainers' perspective.

respectively, within 7 days after the creation of a vulnerability report. Maintainers integrate fixes for the majority of CSVs within one week. For testing fixes, Fedora takes up to 30 days for 59% of CSVs, while Debian takes one day for the same proportion. As discussed in RQ1, Debian performs part of tests in its CI system but there is no link between bug reports and the test logs in the CI system.

Testing fixes for CSVs requires a median of 4.3 days longer than integrating fixes for CSVs across Debian and Fedora, as shown in Figure 4.12. The difference between the integration and test phases is significantly different, according to the Wilcoxon test, with a p-value of 0.001 ($\alpha = 0.01$) and a negligible effect size of 0.03.

From the user perspective, users wait for a median of 0.2 days to get fixes for CSVs while they take a median of 17 days to be aware of CSVs. The Wilcoxon test indicates that there is a significant difference between the user-unawareness and user-waiting period, with a p-value $< 2.2e^{-16}$ and a medium effect size of 0.47. These median values of the two phases for CSVs are the same as the median values of the two phases in RQ1 (Figure 4.9).

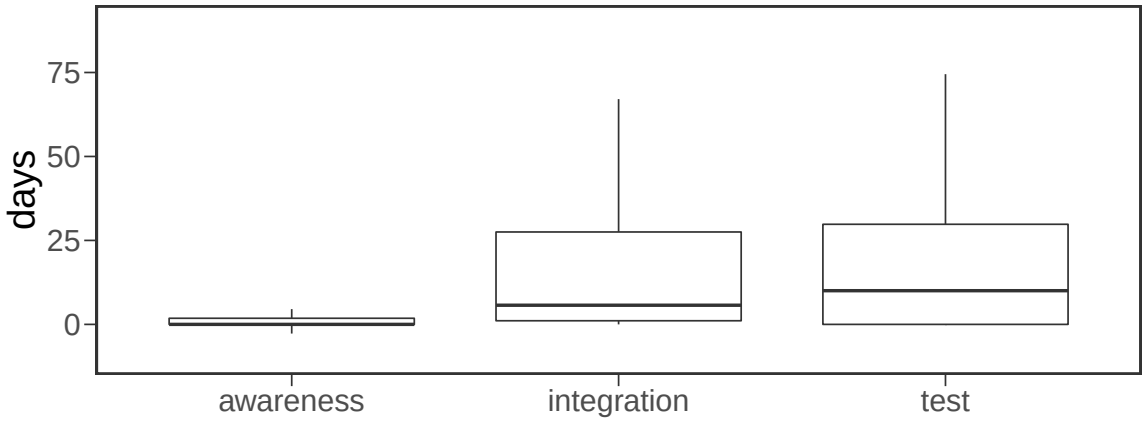
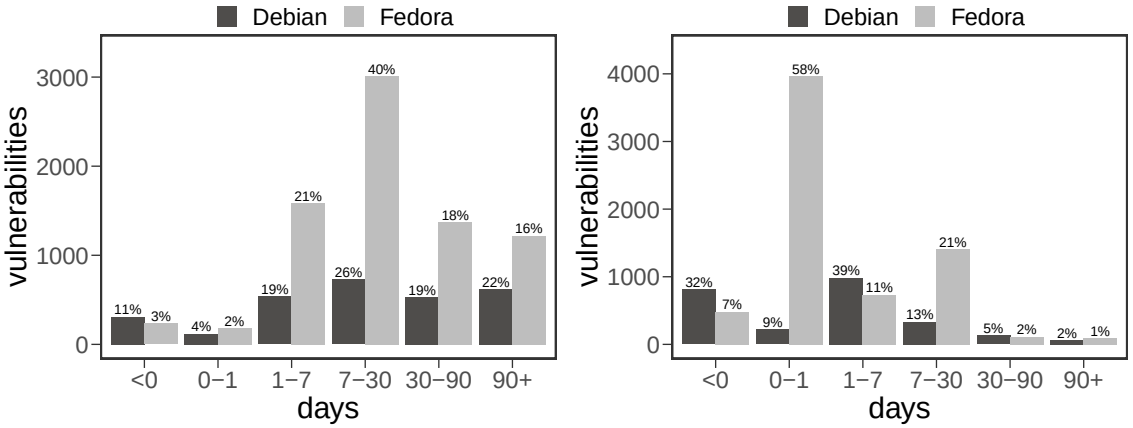


Figure 4.12: The comparison of the length of each phase from the maintainers' perspective for CSVs. The awareness period is shorter than the integration and test phases.



(a) User unawareness

(b) User waiting

Figure 4.13: The duration of the two phases from the user perspective for CSVs.

In addition, similar to the results from the maintainer perspective, the user-unawareness and user-waiting periods have different distributions in Debian and Fedora. Figures 4.13a and 4.13b show the distribution of CSVs in the two user perspective phases in Debian and Fedora. These differences might be due to the different strategies of vulnerability management in Debian and Fedora. Future work could replicate our work on other distributions to generalize our findings.

The majority of reported vulnerabilities in Debian (74%) and Fedora (92%) are security vulnerabilities common to both distributions (CSVs). The CSVs are easy to exploit, though with a medium impact on victim systems. Maintainers take less time to integrate vulnerability fixes for CSVs than testing fixes. Fedora tests the fixes of 59% of CSVs in 30 days, compared to one day in Debian. Furthermore, Fedora users get fixes for 65% of CSVs within one day, compared to within one week in Debian.

RQ3: How do developers fix common security vulnerabilities (CSVs) across the studied distributions?

Results (awareness):

For the majority of fixed CSVs, distributions are notified about vulnerabilities by the upstream projects. Figure 4.5 presents the 5 identified sources through which the security team becomes aware of CSVs. The security teams in Debian and Fedora are aware of 58% (201 out of 345) and 59% (206 out of 345) of the fixed CSVs from the affected upstream projects directly. Our results show that the awareness of distributions

about vulnerabilities comes mainly from upstream projects, along with a fix. This indicates that the collaboration across distributions and upstream projects on vulnerability fix propagation is in line with the guidelines for multi-party coordination from first.org (as mentioned in Section 4.1).

For the majority of vulnerabilities, the security teams in the studied distributions are also made aware of an available fix. Figure 4.5 shows that the security team provides the information (*e.g.*, links) to a fix patch for the majority of the fixed CSVs in Debian (303 out of 345, 88%) and Fedora (262 out of 345, 76%). Maintainers in Debian and Fedora need to fix vulnerabilities for their users even if they did not introduce the vulnerabilities. However, due to their lack of detailed knowledge about the upstream projects' internals (Ma et al., 2017), adopting a provided fix for a vulnerability is the common way for distribution maintainers to remediate the vulnerability and quickly release a new version to users, compared to investigating and addressing the root causes by themselves.

To this end, our results indicate that the security team usually offers maintainers sufficient information (*e.g.*, patch, upstream reports that they require) to achieve the goal of fast remediation for vulnerabilities. For example, for the RTP resource exhaustion (CVE-2016-7551) in the *asterisk* package, the security team posted the upstream bug report along with the reference link to a fix patch when the security teams created the corresponding bug reports in both Debian[40] and Fedora.[41]

For Fedora, the security team of Red Hat is almost exclusively responsible for discovering the CSVs, while, in Debian, 30% of fixed CSVs are reported by users. More specifically, the security team of Red Hat creates bug reports for 99% of fixed CSVs,

[40] https://bugs.debian.org/cgi-bin/bugreport.cgi?bug=838832
[41] https://bugzilla.redhat.com/show_bug.cgi?id=CVE-2016-7551

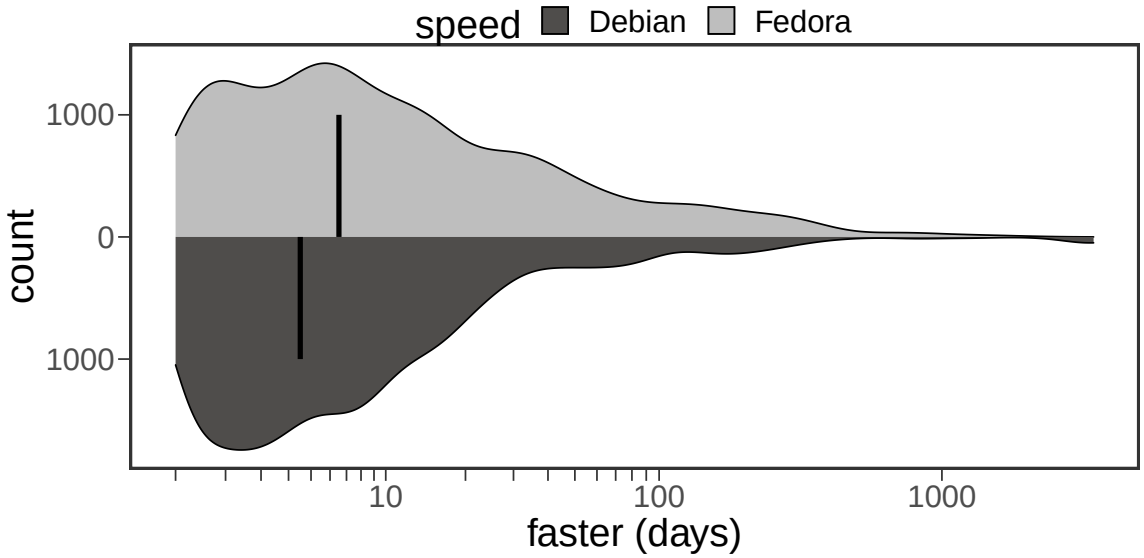


Figure 4.14: The distributions of the difference (in log-scale days) between the creation times of fixed CSV reports with the same CVE-ID in Debian and Fedora. The black lines indicate the median values of the two distributions.

while the security team in Debian "only" reports 70% of fixed CSVs, although with an increasing trend. As mentioned in Section 4.4.1, since Fedora is primarily sponsored by Red Hat,[42] the security team of Red Hat handles vulnerabilities in Fedora.[43] This result is similar to the result we found in RQ1 that users report 34% and 6% of vulnerabilities to Debian and Fedora, respectively.

The security teams in Debian and Fedora are aware of fixed CSVs at a similar speed. The security team in Debian is aware of 41% of fixed CSVs faster than Fedora and vice versa, while the security teams are aware of 18% of fixed CSVs on the same day. Figure 4.14 shows that the awareness speed for the 41% of fixed CSVs in Debian is a median of 5.5 days faster than in Fedora. For the 41% of fixed CSVs that Fedora became aware of earlier, the awareness speed is a median of 7.5 days faster than Debian.

[42] https://getfedora.org/sponsors/
[43] https://fedoraproject.org/wiki/Security_Bugs

Results (embargo):

Red Hat becomes aware of 24% (*i.e.*, 83 out of 243) of CSVs without a fix and embargoes half (42 out of 83) of these CSVs to prevent early disclosure. Since Red Hat plays the role of a trusted partner for many upstream open-source projects to assist them with security issues,[44] upstream projects reach out to Red Hat for rapidly developing a fix when they became aware of an embargoed vulnerability. In addition, the Red Hat security team contributes patches for 13 (31%) out of the 42 embargoed CSVs, suggesting a close collaboration between Red Hat and upstream projects on fixing vulnerabilities.

On the other hand, Debian does not embargo vulnerabilities (Debian.org, 2021e), which increases the chance of attackers exploiting vulnerabilities after their disclosure, if the fixing process takes longer than foreseen. For example, the bug report for a CSV (CVE-2013-6456) was publicly disclosed in Debian[45] when Debian maintainers were not yet able to fix it. One Red Hat developer was involved in fixing the vulnerability by evaluating a patch and providing suggestions. In the end, the upstream project accepted the patch that was submitted by the Red Hat developer.[46]

Although Debian does not embargo any vulnerabilities, Debian is potentially involved in vulnerability fix development during the embargo period. Out of the 42 embargoed CSVs, Debian referred to Red Hat's vulnerability reports and/or security advisories for 12 (29%) embargoed CSVs and attached the same references that were attached to Red Hat's vulnerability reports for 6 (14%) embargoed CSVs. For example, CVE-2017-1000409[47] was a buffer overflow in the *glibc* 2.5 package and can be triggered by a specific environment variable. The fix was developed by the members of the

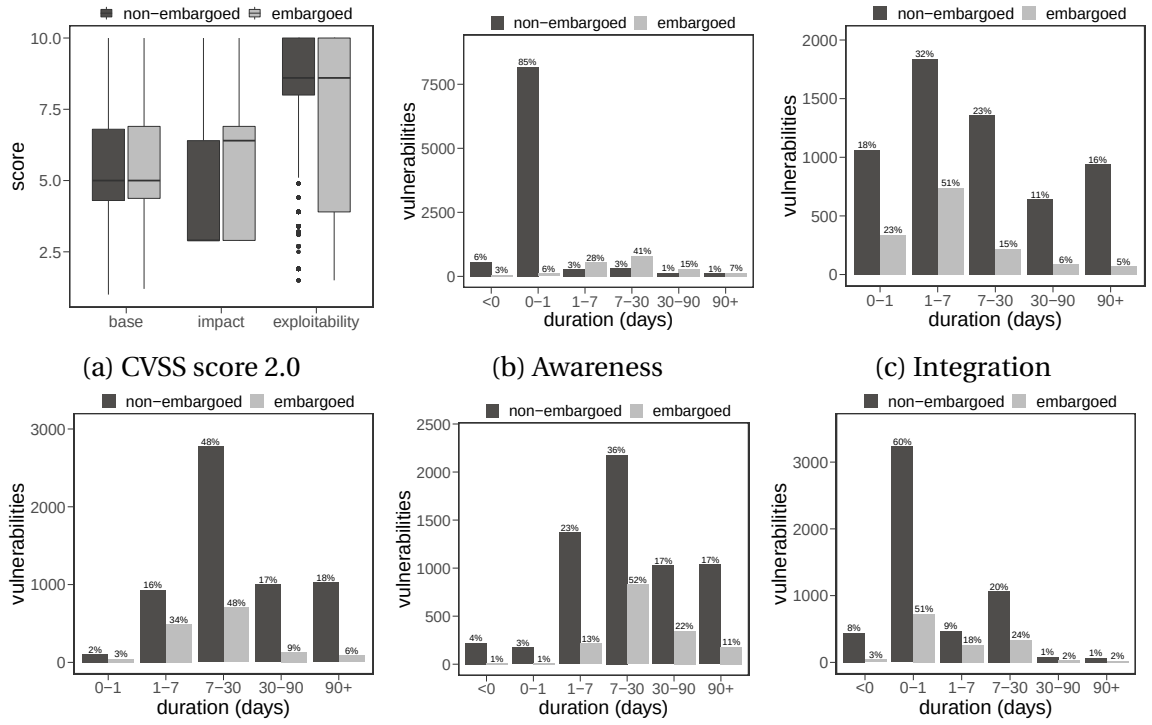
[44] https://www.redhat.com/en/resources/managing-vulnerabilities-FAQ
[45] https://bugs.debian.org/cgi-bin/bugreport.cgi?bug=732394
[46] https://libvirt.org/git/?p=libvirt.git;a=commit;h=5fc590ad9f4
[47] https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-1000409

open-source security group[48] of which both Debian and Fedora are a member. Another embargoed CSV, CVE-2012-2737,[49] was found and embargoed by Red Hat[50] and also affected the *accountsservice* package in Debian. The respective bug report in Debian was created within one hour after the embargo was lifted and one Debian maintainer posted the link to the Red Hat's report,[51] with the report hinting at the corresponding maintainer being aware of the embargoed vulnerability during the embargo period.

Embargoed CSVs are more risky than non-embargoed CSVs since they have a higher impact score, based on CVSS score 2.0. Figure 4.15 shows the comparison between embargoed and non-embargoed CSVs. Embargoes can reduce the risk of a vulnerability being exploited by attackers before having a mitigation (Redhat.com, 2021). However, embargoing a vulnerability is challenging in an open source development setting, since the code repositories (*e.g.*, GitHub) of software projects are publicly available. Hence, the embargoed vulnerability can be found by anyone during the embargo period, such as the *Heartbleed* vulnerability (CVE-2014-0160), which was initially found by Codenomicon, before being independently found by the Google security team.[52] The history of the vulnerability report of *Heartbleed* indicates that it was embargoed for a short period (*i.e.*, about 12 hours) after the security team created the vulnerability report.[53]

Embargoed CSVs have a slower speed of awareness for maintainers but a faster speed of integrating and testing vulnerability fixes, compared to non-embargoed CSVs. Figures 4.15b - 4.15f show the five phases for the embargoed and non-embargoed CSVs

[48] https://seclists.org/oss-sec/2017/q4/385
[49] https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2012-2737
[50] https://bugzilla.redhat.com/show_bug.cgi?id=CVE-2012-2737
[51] https://bugs.debian.org/cgi-bin/bugreport.cgi?bug=679429
[52] https://www.zdnet.com/article/heartbleed-serious-openssl-zero-day-vulnerability-reveale
[53] https://bugzilla.redhat.com/show_activity.cgi?id=1084875



(a) CVSS score 2.0 (b) Awareness (c) Integration

(d) Test (e) User unawareness (f) User waiting

Figure 4.15: The comparison between the embargoed and non-embargoed CSVs in Fedora. (a) shows the distribution of the base, impact and exploitability scores. (b), (c), and (d) reflect the three phases from the maintainer perspective. (e) and (f) reflect the two phases from the user perspective.

from the maintainer and user perspectives, respectively. For the maintainer perspective, the awareness period of embargoed CSVs accounts for a median of 43% of the total time for tracking a CSV in Fedora, while that of non-embargoed CSVs accounts for a median of 0%, though those non-embargoed CSVs might have been embargoed somewhere else (without Fedora being aware of this). However, the integration and test phases for embargoed CSVs are shorter than non-embargoed CSVs (Wilcoxon test: p-value $< 2.2e^{-16}$, $\alpha = 0.01$). Both the integration and test phases have a small effect size of 0.29 and 0.3, respectively. By contrast, from the user perspective, both the user-unawareness and user-waiting phases for the embargoed CSVs are longer than

non-embargoed bugs, by the Wilcoxon test (user-unawareness: p-value $= 5e^{-10}$, user-waiting: p-value $< 2.2e^{-16}$, $\alpha = 0.01$). Both the user-unawareness and user-waiting phases have a negligible effect size of 0.13 and 0.002, respectively.

Results (vulnerability fix integration):

Debian maintainers fix 42% (143 out of 345) of the CSVs by applying customized patches, compared to 11% (39 out of 345) of the CSVs in Fedora. Once the upstream project has a patch to fix a vulnerability, distribution maintainers either leverage the fix patch and integrate it into their package manually or wait for the upstream release that incorporates the patch. The former case is the fastest way for distribution maintainers to release remediation of vulnerabilities to their users. However, distribution maintainers need to maintain these patches until the upstream has a release (Ma et al., 2017). By contrast, adopting the upstream release represents the simplest way for distribution maintainers to fix vulnerabilities though users might need to wait a longer time for the vulnerability fix. For example, Debian maintainers fixed the CVE-2017-16808 vulnerability[54] by producing a patch, 54 days faster than Fedora maintainers who applied the upstream release.[55]

In addition, distribution maintainers might not be available to rapidly produce a new version for the affected packages or there might not even be a maintainer available for it. Therefore, the security team would take care of these affected packages and release new versions for users. For example, CVE-2010-3695 in Debian[56] was fixed by

[54] https://bugs.debian.org/cgi-bin/bugreport.cgi?bug=881862
[55] https://bugzilla.redhat.com/show_bug.cgi?id=1516995
[56] https://bugs.debian.org/cgi-bin/bugreport.cgi?bug=698916

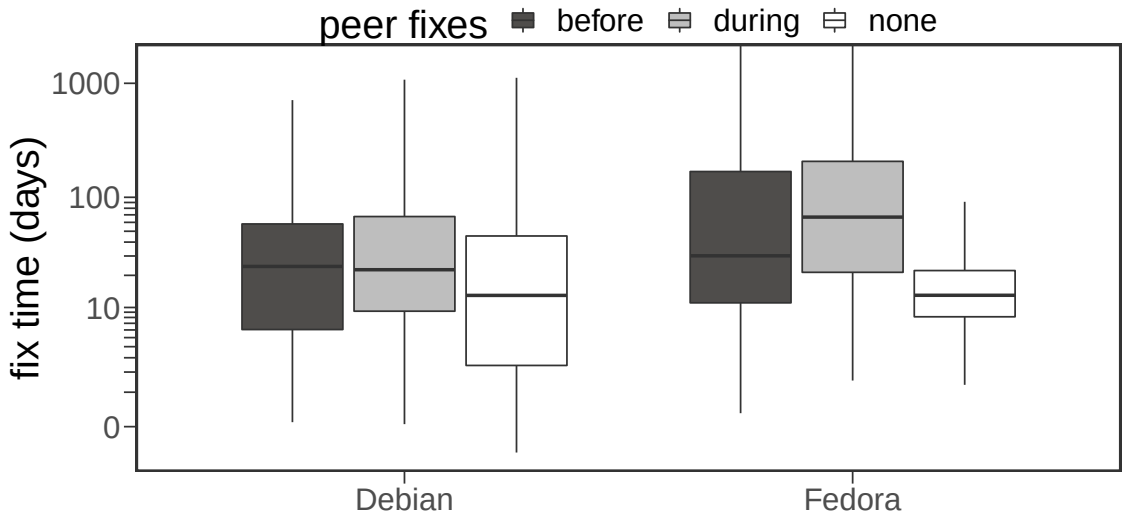


Figure 4.16: Debian and Fedora maintainers do not speed up the integration of vulnerability fixes (in log-scale days) when peer fixes are available before the integration.

the security team, as stated in the release message *"Non-maintainer upload by the security team"*. In fact, the security team in Debian has fixed 1,707 (8%) vulnerabilities for the affected packages that did not have maintainers when the vulnerabilities were reported.

In a distribution, the availability of a peer fix for a CSV before maintainers start to integrate the CSV fix does not correlate with faster integration and testing of the vulnerability fix. Figure 4.16 shows boxplots of the fixing time, *i.e.*, the sum of the duration of the integration and test phases from the maintainer perspective, for CSVs based on when a peer fix is available in the two studied distributions. In particular, Fedora maintainers fix CSVs without a peer fix (the white box) faster than those that have a peer fix before integration (the dark grey box) (Wilcoxon test: p-value $= 2.2e^{-16}$, $\alpha = 0.01$), with a medium effect size of 0.71. In Debian, we observe a similar phenomenon where maintainers take more time to fix CSVs when a peer fix is available before or during development, compared to those without a peer fix (Wilcoxon test: p-value $= 2.6e^{-6}$,

$\alpha = 0.01$). The effect size is small (0.15). Our results suggest that collaboration across distributions is less likely to happen after maintainers start to integrate a vulnerability fix, since maintainers integrate packages into their distributions following their own packaging policy.

Our qualitative analysis showed five sources used by the security teams in Debian and Fedora to become aware of CSVs and seven strategies used by distribution maintainers in the fixing process of vulnerabilities. In the awareness phase, the security team becomes aware of the majority of CSVs from upstream along with the respective fixes. When a CSV fix has not yet been developed, Red Hat (Fedora) embargoes half of the CSVs to prevent early disclosure due to their threat level. In the phase of vulnerability fix integration, having a peer fix available does not reduce the needed time for maintainers to integrate vulnerability fixes.

4.6 Discussion

In this section, we discuss the findings presented in Section 4.5.

Distribution maintainers collaborate closely across distribution boundaries before vulnerability fixes are available, but less afterwards, *i.e.*, in the integration and test phases. Even though the majority of CSVs in both Debian (51%) and Fedora (65%) have a peer fix before the integration of CSV fixes, we observe in RQ3 that maintainers do not release CSV fixes faster in such cases. In addition, Debian (58%) and Fedora (59%) are notified about the majority of fixed CSVs by upstream projects, even though

they share a mailing list with other Linux distributions (OSS-security.org, 2021), indicating that Debian and Fedora are collaborating on developing fixes rather than on the discovery of a vulnerability.

Hence, since the two studied distributions are major software ecosystems in the larger Linux ecosystem, there is still room for more collaboration on fixing vulnerabilities at the scale of the whole Linux ecosystem. This should lead to more benefits in terms of reduced fixing time, especially since the majority of vulnerabilities of both Debian and Fedora are CSVs (RQ2).

In addition, there is **a need for better tracking of the collaborative fixing process of vulnerabilities** to enable the ability of evaluating and continuously improving the collaboration model. For example, in the studied distributions, some of the collaborative work occurs via email (both public and private mailing lists) or other channels, including reporting a vulnerability to upstream, discussions before a fix is developed, and notifications of a vulnerability (with its fix). Some parts of the process happen under embargo, with some of the work being opened up afterwards. Similarly, some vulnerability fixes are applied locally in the distribution's code base, while others come in through new version updates of packages.

Hence, there is a lack of traceability from a particular vulnerability fix to all the phases of work involved. Since vulnerabilities have a large impact on software systems, having effective collaboration is necessary to develop vulnerability fixes quickly for vulnerable software systems and to minimize the losses of vulnerability exploits. Adding labels and corresponding times for the collaborative behaviors (*e.g.*, receiving the notification of the discovery of a vulnerability) in issue tracking systems might be

a first step towards improved traceability. Researchers could leverage this data to improve the collaboration model across distributions in the Linux ecosystem.

Embargoed high-severity vulnerabilities have quicker fixes for users than non-embargoed ones. Embargoes can reduce the risk of a vulnerability being exploited by attackers before the fix has been developed. Red Hat states the expected benefit of embargoes as *"we'll fix these issues in private and release the update in a coordinated fashion in order to minimize the time an attacker knows about the issue and an update is not available"* (Redhat.com, 2021). We found in Figure 4.15a that the embargoed vulnerabilities have a higher impact on victim software systems. Figure 4.15c indicates that 74% of embargoed CSVs have been integrated into the distribution within 7 days after maintainers have been assigned the task, compared to 50% for non-embargoed CSVs. For the testing phase, Figure 4.15d indicates that 37% of embargoed CSVs passed testing within 7 days, compared to 18% of non-embargoed CSVs.

However, this success comes at a cost, since fixing an embargoed vulnerability in secret requires additional costs (Redhat.com, 2021). For this reason, Red Hat has chosen to only embargo vulnerabilities having a high risk with respect to its products (Redhat.com, 2021).

4.7 Threats to Validity

4.7.1 Internal Validity

We study vulnerability management based on vulnerability reports, bug reports and security advisories in a given distribution. These three sources might not capture all activities related to fixing vulnerabilities in a distribution. For example, developers

have other channels (*e.g.*, emails) where they discuss vulnerabilities. Our dataset does not include vulnerabilities in the Long-Term-Support (LTS) versions of Debian releases since those are not recorded in Debian's issue tracker. In addition, the LTS versions are maintained by a group of volunteers and companies, instead of by Debian itself[57]. Nevertheless, to the best of our knowledge, no prior work leverages the three data sources in a given distribution to draw insights into how its security team, maintainers and upstream projects collaborate to fix vulnerabilities. We encourage future work to add additional data sources to study developer communication on vulnerability management.

Our study focuses on vulnerabilities with a valid CVE-ID and ruled out those with an invalid CVE-ID.[58] These invalid CVE-IDs may still correspond to vulnerabilities to a certain extent but they were invalid due to a variety of reasons. For example, the CVE website stated that a CVE-ID could be rejected when further research determines the issue not to be a vulnerability.

In RQ2, we only study CSVs across the studied distributions. We did not consider vulnerabilities that are found only in one of the studied distributions, since our focus on two distributions only is not sufficient to conclude that such vulnerabilities would only impact one distribution. On the other hand, the studied distributions reflect the root distribution of two notable Linux distribution families (*i.e.*, RPM-based and Deb-based) and both of them have existed over a decade. Many prior works leveraged the data in these studied distributions to analyze security vulnerabilities (Russell et al., 2018; Harer et al., 2018). Future studies could consider adding other distributions to

[57] https://wiki.debian.org/LTS

[58] https://cve.mitre.org/cve/list_rules_and_guidance/correcting_counting_issues.html

identify vulnerabilities that are specific to one distribution (as opposed to being common to other distributions) and why they are specific.

Our study involved a qualitative study performed by humans based on their experience and knowledge as well as the available data sources, (*e.g.*, vulnerability reports, bug reports). To reduce the bias of our analysis, each vulnerability is labeled by two of the authors individually and discrepancies are discussed until a consensus is reached. The data sources might not capture all activities related to fixing vulnerabilities in the distribution. For example, Debian has its continuous integration (CI) system[59] perform automated tests against each package when it and/or its dependencies have been updated. However, we cannot link the logs in the CI system to vulnerability fixes because of the lack of explicit information on how/why the test was triggered. For example, a package can be tested because of a bug fix, a new feature, or its dependent packages that are updated.

4.7.2 External Validity

A threat to the external validity of our study is that we only studied vulnerability fixes and propagation in Debian and Fedora. However, Debian and Fedora are two popular Linux distributions that have existed for more than 10 years, as discussed in Section 4.4. In addition, there are Linux distributions (*e.g.*, Arch Linux) using another type of release model, *i.e.*, rolling release. Such distributions release a new version once their code has been updated, and might follow different practices to manage vulnerabilities. Thus, we cannot claim that our results generalize to other Linux distributions that are not studied in this work. Hence, future work should replicate this work to reach more

[59] https://ci.debian.net/doc/

general conclusions.

4.8 Chapter Summary

Our study focuses on 21,752 and 17,434 vulnerabilities in Debian and Fedora, respectively, to study the practices of vulnerability management within a distribution and across the distributions. Our analysis shows that Debian and Fedora maintainers integrate vulnerability fixes for the majority (50%) of vulnerabilities within 1 week. Fedora maintainers take a longer time to test vulnerability fixes in the vulnerability fixing process. From the user perspective, the time during which users are not aware of vulnerabilities is 85 times longer than the time during which users wait for vulnerability fixes after becoming aware, across Debian and Fedora. Users wait for less than 2 weeks for the fixes of 75% vulnerabilities after becoming aware.

Common security vulnerabilities (CSVs) are dominant across Debian (74%) and Fedora (94%). These CSVs are easy to exploit, *i.e.*, they have a median exploitability score of 8.6, though with a medium impact (median of 5), according to CVSS score 2.0. Maintainers spend more time testing vulnerability fixes for CSVs than integrating vulnerability fixes by a median of 4.3 days across Debian and Fedora. Users wait for a median of 0.2 days to get fixes of CSVs after the releases of their respective security advisories.

Finally, we conduct a qualitative study that investigates the sources through which maintainers become aware of CSVs and the practices with which maintainers manage CSVs. In general, upstream projects provide notifications and fixes for the majority of fixed CSVs to Debian (58%) and Fedora (59%). For the CSVs without an available fix, Fedora embargoes half of them to prevent early disclosure since embargoed CSVs have a higher impact score, according to CVSS score 2.0. Although the embargo policy makes

maintainers become aware of vulnerabilities later than non-embargoed vulnerabilities, maintainers integrate and test fixes of the embargoed vulnerabilities faster than non-embargoed vulnerabilities. Additionally, Debian maintainers are more likely to integrate their own patches for CSVs, while Fedora maintainers tend to integrate new upstream releases.

Our study is a first step to aggregate and empirically study several data sources in Linux distributions to understand vulnerability management in practice. Our results suggest that distribution maintainers collaborate to develop vulnerabilities together before public disclosure, while they integrate fixes into their distributions in parallel. Future work could investigate additional practices by exploring other datasets (*e.g.*, emails, developer communication channels, vulnerability fix releases) and interviewing maintainers. Future work could also investigate the practices of vulnerability management in upstream projects to have comprehensive insights of the collaboration model in the Linux ecosystem.

CHAPTER 5

Multi-party coordination for vulnerability fixing and disclosure

The Common Vulnerabilities and Exposures (CVE) program is dedicated to analyzing vulnerabilities, then to assigning a unique ID to them and disclosing the vulnerabilities to affected software vendors. A CVE Numbering Authority (CNA) is a key partner in the CVE program responsible for assigning an official ID to a CVE and registering a description of the vulnerability in order to communicate it to the other CNAs and the affected software vendors. To avoid the disclosure of vulnerabilities before the development of a fix, the CNAs and the affected vendors need to coordinate a proper schedule for the disclosure of vulnerabilities and the release of their fixes through multi-party coordination. This work analyzes the practices used by CNAs to coordinate on vulnerability fix releases and disclosure by empirically studying the 13 CNAs that assigned the most CVEs from 2010 to 2020 and are also software vendors. Our results show that the studied CNAs discover and assign CVE IDs for the majority of vulnerabilities that affect their own products, which we refer to as self-assigned vulnerabilities. While the vulnerabilities that are assigned for other CNAs' products, which we refer to as delegated vulnerabilities, tend to be more severe than the self-assigned vulnerabilities, (median Common Vulnerability Scoring System score of 7.5), we observe that their fixes are released at a slower pace. Moreover, when such a delegated vulnerability affects several CNAs' products, the fixes are released a median of 4 days after the disclosure date, with a median delay between the first and last patch releases of those products of 35 days up to more than one year, which corresponds to a large window of exploitation.

An earlier version of this chapter is suggested major revision by the Empirical Software Engineering Journal (EMSE).

5.1 Introduction

The surge of vulnerabilities in the past few years indicates the high risk of keeping software systems operational, since software vendors cannot guarantee a patch for all vulnerabilities before those are exploited by attackers. In fact, the number of exploits on zero-day software vulnerabilities has doubled from 2020 to 2021.[1] Releasing a timely fix becomes vital to reduce exploits of such vulnerabilities and save the losses of vulnerable software. In other words, a delay of fix releases until after disclosure implies higher risks to users. For example, millions of users waited for at least 6 months to get fixes of a particular vulnerability in Microsoft Word.[2] As a result, this delay led to a large number of attacks on victim systems and a substantial amount of losses.

To deal with vulnerabilities, organizations and companies often rely on vulnerability databases to become aware of vulnerabilities and get their fixes. The CVE program (Mitre.org, 2021) is designed to link such vulnerability databases and other tools together by assigning a unique identifier (CVE ID) to a given vulnerability, which software vendors often include in the relevant mitigation plans (e.g., security advisories, fixes). Users leverage the CVE ID to find all the information (e.g., fixes) related to that vulnerability.

CVE IDs are assigned by a diverse group of organizations, companies and researchers, acting as CVE Numbering Authorities (CNAs) (cna, 2022). There are over two hundred CNAs, each of which are security experts within their own specific scopes of coverage. They evaluate a vulnerability before assigning an identifier, then control the disclosure process. That is, the CNAs, the discoverer, and the affected software

[1] https://www.technologyreview.com/2021/09/23/1036140/2021-record-zero-day-hacks-reasons/
[2] https://www.businessinsider.com/hackers-microsoft-word-flaw-reuters-2017-4

vendors coordinate to develop a vulnerability fix and schedule a proper disclosure time, i.e., multi-party coordination.

There are many challenges for parties involved in multi-party vulnerability coordination and disclosure (First.org, 2022). For example, the number of affected vendors is typically too large for the discoverer to deal with, thus involving a coordinator (CNA) to cope with this. However, the higher the number of involved parties in fixing a vulnerability, the higher the complexity of coordination between the involved parties. In such a case, the affected vendors are unlikely to release their fix in a coordinated fashion. In addition, when an affected vendor does not consider a report to be a vulnerability, the CNA is forced to disclose the vulnerability to nudge the vendor to release a fix, implying that the vulnerable products become the target of attackers. Furthermore, cloud computing services often use customized versions of open-source projects like distributions or middleware. As such, the fix for a particular vulnerability in, say, a customized Debian version on OpenStack is likely to be released later than the fix for Debian itself.

It is important to note that, while CNAs correspond to security experts generating new CVE IDs, they also can play other roles in parallel, such as that of the discoverer, coordinator, and/or affected vendor. First, a CNA could be the discoverer of a vulnerability in its own products, and be responsible for assigning a CVE ID for the vulnerability, before, subsequently, having to develop a vulnerability fix in a timely manner. Second, the CNA could be responsible for both assigning a CVE ID and controlling the disclosure process, notifying the public of the vulnerability to remediate the vulnerable products. In this case, the CNA coordinates the schedule of releasing the fixes with the discoverer and the affected vendors, including the (upstream) vendor that developed

the vulnerable product and those (downstream) that leveraged the vulnerable product as part of their products. As typically multiple downstream vendors rely on the fixes released by the upstream vendor, this increases the difficulty of coordination. Finally, the CNA could be one of the affected vendors receiving the notification of its vulnerable product, then has to release the fix according to the coordinated schedule.

The large threat of vulnerabilities led researchers to systematically analyze vulnerabilities from a wide range of aspects. Some prior studies analyzed vulnerability lifecycles to shed light on the vulnerability disclosure process (Frei et al., 2006; Shahzad et al., 2012). Other studies explored the disclosure process from the economy of vulnerabilities, such as vulnerability trading markets (BHARGAVA and CHANDINI, 2018; Sood et al., 2012), and evaluated the correlation with exploits (Allodi, 2017). Some studies worked on understanding the patch development process (Huang et al., 2016; Ozment and Schechter, 2006; Li and Paxson, 2017). While providing insights of vulnerabilities for a certain aspect, these investigations have been limited to the software vendors' perspective. However, to the best of our knowledge, no study has focused on the special role of CNAs.

In this work, we conduct a large-scale empirical study of 39,409 vulnerabilities that affect products of the top 13 CNAs with the largest number of assigned CVE IDs, based on a dataset that merges vulnerability entries from the National Vulnerability Database (NVD) (nvd, 2022) and their associated security fixes. Due to the different responsibility of being a CNA and being a software vendor, we study the CNAs who have played both roles to gain an understanding of the efficiency of releasing vulnerability fixes w.r.t the disclosure date. We classify vulnerabilities into two groups: vulnerabilities whose CVE-IDs are assigned by the affected CNAs, i.e., *self-assigned vulnerabilities*, and those

whose CVE IDs are not assigned by the affected CNAs, i.e., *delegated vulnerabilities*. We characterize in detail the delay of releasing vulnerability fixes in comparison to different affected product types and the number of vendors. We also explore the delay in each of the studied CNAs and summarize common practices. We structure our study by answering the following research questions, yielding the following key findings:

RQ1: What are the characteristics of self-assigned vs. delegated vulnerabilities?)

The first RQ analyzes the prevalence of self-assigned vs. delegated vulnerabilities to measure the degree to which the CNAs focus on vulnerability discovery, and characterizes these two types of vulnerabilities. The majority of vulnerabilities affecting CNAs are self-assigned, though 3 out of the 13 CNAs have a similar percentage (≈50%) of self-assigned and delegated vulnerabilities. Application-type products have the largest proportion (47%) of self-assigned vulnerabilities across the CNAs, compared to the operating-system (35%) and hardware (18%) types of products. Furthermore, delegated vulnerabilities are more severe than self-assigned vulnerabilities, with a median Common Vulnerability Scoring System (CVSS) score of 7.5.

RQ2: How fast do the CNAs develop fixes for self-assigned vs. delegated vulnerabilities?)

Due to the high severity level of delegated vulnerabilities, RQ2 measures how fast the studied CNAs release fixes for these vulnerabilities per product type, and compare the results with self-assigned vulnerabilities. In general, the studied CNAs release fixes for delegated vulnerabilities slower than self-assigned ones. For delegated vulnerabilities affecting the application type of product, while the

studied CNAs release fixes in a median of -1 day (before the disclosure date), 2 of the CNAs (i.e., Apple and Oracle) release the majority of fixes at least 90 days after the disclosure date. For delegated vulnerabilities affecting the operating-system type of product, the delay of patch releases is a median of 3 days after the disclosure date. 3 out of the studied CNAs (i.e., Apple, Oracle, and Red Hat) release fixes for the majority of such vulnerabilities at least 90 days after the disclosure date.

RQ3: How effectively do the CNAs coordinate to release fixes for shared vulnerabilities?)

Since the delay of patch releases for delegated vulnerabilities is longer than self-assigned ones, this RQ studies potential collaboration between the studied CNAs on releasing fixes. We measure the delay of vulnerabilities that affect multiple CNAs' products, i.e., multi-affected vulnerabilities, vs. those that affect one CNA's products, i.e., single-affected vulnerabilities. 3 out of the studied CNAs have a majority of their vulnerabilities as multi-affected. The CVSS score of multi-affected vulnerabilities that affected both application and operating-system types of products reaches almost the full score of 10 (extremely severe), while the CVSS score of those that affected one product type is a median of 5.1 (medium-severity). In addition, the delay of multi-affected vulnerabilities that are delegated is a median of 4 days after the disclosure date. Moreover, 4 out of the studied CNAs (Apple, IBM, Oracle, and Red Hat) release the fixes of such vulnerabilities slower than the other studied CNAs, with the majority of fixes being released at least 90 days after the disclosure date.

The studied CNAs release prompt fixes for self-assigned vulnerabilities, while they

are relatively slower for delegated vulnerabilities, which are more severe. These results suggest that some CNAs coordinate on releasing fixes while some do not, ending up with a large window (of at least 90 days) for potential vulnerability exploits related to delegated vulnerabilities in the operating-system type of product or those that affect multiple CNAs' products. Researchers should focus on better understanding and analyzing the collaboration processes for such kinds of vulnerabilities to reduce their impact.

The chapter is organized as follows. Section 5.2 provides the background of common security specifications and multi-party coordination on fixing vulnerabilities. Section 5.3 discusses prior related work to our study. Section 5.4 describes how we design our study and obtain our studied dataset. Section 5.5 answers our research questions. Section 5.6 discusses the implications of our study. Section 5.7 discusses the threats to the validity of our study. Finally, Section 5.8 concludes the work.

5.2 Background

In this section, we briefly introduce the Common Vulnerabilities and Exposures (CVE) program and National Vulnerability Database (NVD), the Common Vulnerability Scoring System (CVSS), the Common platform enumeration (CPE) specification, and the multi-party coordination model on fixing vulnerabilities.

5.2.1 Common Vulnerabilities and Exposures (CVE) and National Vulnerability Database (NVD)

The CVE platform (Mitre.org, 2021) is one of the most popular platforms storing lists of publicly disclosed computer security vulnerabilities. Since there are many databases storing information regarding security vulnerabilities, the CVE platform assigns a CVE ID to each vulnerability to give users a way to recognize unique vulnerabilities across various databases, and coordinates the development of fixes across software vendors. Such CVE IDs are assigned by CVE Numbering Authorities (CNAs), each of which represents a large software company, security company, or research organization. The diversity of CNAs provides varied yet specific areas of expertise for different types of vulnerabilities. After a CVE ID is assigned, the CNA records concise information (e.g., ID, description) about the vulnerability and includes references, while the full details (e.g., affected products) appear in other databases, such as the NVD. Then, the new CVE record will be posted on the CVE website.

The NVD (nvd, 2022) is tasked with analyzing CVE records after they have been synchronized from the CVE website to the NVD, typically within an hour. After a CVE record is in the NVD, NVD analysts analyze the vulnerability based on the relevant information provided with the CVE record, such as references. The analysts also manually search any publicly available information on the internet at the time of the analysis. The analysis process ensures that any publicly available information is used to document affected vendors and products for the CVE. This information also includes CVSS scores (Section 5.2.2) and a specification of the affected products using the CPE standard (Section 5.2.3).

5.2.2 Common Vulnerability Scoring System (CVSS)

The CVSS (NVD, 2021) is an open framework for assigning scores to a vulnerability to indicate its threat level. CVSS scores are widely used by many prior studies (Joh and Malaiya, 2011; Shahzad et al., 2012; Zhang et al., 2021) and large organizations and companies (e.g., Computer Emergency Response Team (CERT)) to communicate the severity of the vulnerabilities found in their products. A CVSS score is composed of three sets of metrics: base, temporal and environmental, each of which has its own underlying scoring component.

The base metrics represent the innate characteristics of a vulnerability to indicate its severity via two subscore groups: exploitability, i.e., how the vulnerability is exploited, and impact, i.e., the extent of victim system losses if the vulnerability is exploited. Computing all the subscores results into a threat score of a vulnerability. For example, a base score in the range of 4.0 to 6.9 indicates medium severity.

The temporal metrics represent metrics that can change over time. These metrics measure the current exploitability of the vulnerability, as well as the availability of mitigation plans, such as a patch.

Finally, the environmental metrics reflect additional attributes of a vulnerability in a specific environment, in terms of potential losses of a brand or a product, the proportion of vulnerable systems and the extent of impact. Note that we use CVSS 2.0 as CVSS 3.0 (released in 2015) and 3.1 (released in 2019) are inadequate for the majority of our data as we study vulnerabilities whose CVE ID was issued between 2010 and 2019.

5.2.3 Common platform enumeration (CPE) specification

The CPE specification[3] defines standardized methods for assigning names to software vendors and their products. For example,

"*cpe:2.3:a:microsoft:internet_explorer:8.0.6001:beta:*:*:*:*:*:*:*"*

represents a CPE string following the CPE 2.3 specification. In this case, the vendor and its product are Microsoft Internet Explorer 8.0.6001, Beta version. The CPE specification classifies products into three types: applications (a), operating systems (o), and hardware devices (h). The "a" before "microsoft" in the CPE string indicates that Internet Explorer is an application. The string after "Beta" is a formatted string to reflect additional attributes (e.g., target hardware version is x86 or x64) of the product.

5.2.4 Multi-party coordination on fixing vulnerabilities

Figure 5.1 presents the multi-party coordination involved with fixing vulnerabilities. After a vulnerability in a product is found, the discoverer reports it to the affected software vendor(s) and coordinates with the software vendor(s) for mitigation plans. Meanwhile, the discoverer requests a CVE ID from a CNA that controls the vulnerability disclosure process. Before the disclosure, the CNA assigns and validates the vulnerability and coordinates a proper disclosure time with the discoverer and affected software vendor(s). Users typically become aware of vulnerabilities from the CVE record and get the fixes from the affected software vendors.

A CNA can also be the discoverer of a vulnerability, or could be the software vendor of a vulnerable product. In all these cases, the CNA could be the one issuing its own

[3] https://cpe.mitre.org/specification/

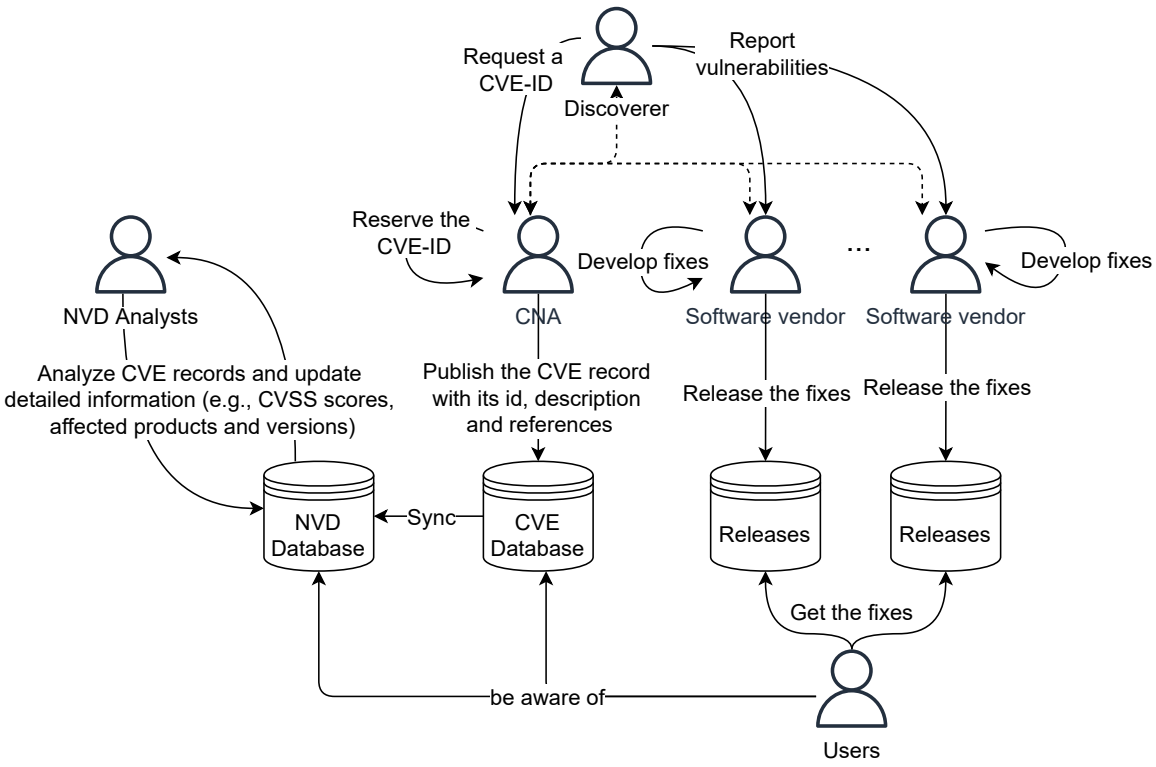


Figure 5.1: Overview of multi-party coordination on fixing vulnerabilities. The dashed lines indicate possible coordination between the parties involved in fixing a vulnerability. The coordination activities include collecting information of a vulnerability, investigating root causes, discussing and developing a fix, and testing the fix. Note that software vendors can be a CNA and/or the discoverer and vice versa.

CVE IDs, or the CVE ID could be assigned by another CNA. Since being a CNA indicates that the CNA has the expertise of vulnerabilities in a specific area, self-assigned vulnerabilities by a CNA are not unheard of. On the one hand, self-assignment reflects an earlier awareness of a vulnerability than attackers, while on the other hand, self-assignment could be subjective and underestimate the impact of the vulnerability. By contrast, delegated vulnerabilities indicate that there is a potential delay of becoming aware of a vulnerability from the discoverer.

There is no uniform policy amongst organizations on how to disclose a vulnerability in multi-party coordination. Several corporations, organizations, and governments have been working on providing guidelines and practices for multi-party vulnerability coordination. For example, MITRE has published the practices for a Coordinated Vulnerability Disclosure (CVD) process.[4] The US' Cybersecurity and Infrastructure Security Agency (CISA) proposed important steps and factors for CVD to ensure that users and administrators receive clear and actionable information in a timely manner.[5] The European Cyber Resilience Act was published in September 2022.[6] However, only CISA suggested more concrete guidelines, such as an explicit number of days for disclosure, as stated "*CISA may disclose vulnerabilities as early as 45 days after the initial attempt to contact the vendor is made regardless of the availability of a patch or update.*" Therefore, our study aims to figure out the current practices from the 13 leading companies (CNAs) by measuring the time taken between the disclosure of vulnerabilities and the release of their fixes in the context of multi-party coordination.

[4] https://cve.mitre.org/cve/researcher_reservation_guidelines
[5] https://www.cisa.gov/coordinated-vulnerability-disclosure-process
[6] https://digital-strategy.ec.europa.eu/en/library/cyber-resilience-act

5.3 Related Work

We discuss prior empirical studies of disclosure and patching (Section 5.3.1) and large-scale vulnerability analysis (Section 5.3.2) and highlight the difference between this study and prior studies (Section 5.3.3).

5.3.1 Disclosure and Patching

Vulnerability disclosure reflects exposing sufficient information of vulnerabilities to the public, where stakeholders can pick up the information and perform mitigation plans. Zhao et al. (2015) observed from two bounty platforms, i.e., Wooyun and HackerOne, that researchers have been making significant contributions, in terms of vulnerability discovery, to secure tens of thousands of organizations on the Internet. Arora et al. (2004) conducted a study to estimate the impact of vulnerability disclosure and availability of patches from both the attackers' and software vendors' perspectives. The study suggested that although software vendors quickly reacted to disclosed vulnerabilities, the frequency of attacks also increased. The study further revealed that open source vendors patched faster than closed source vendors.

Patching applications and operating systems is a common strategy to avoid vulnerability exploits. Some prior works focused on patch development. Nappa et al. (2015) evaluated the impact of code reuse on patching vulnerabilities on 1,593 vulnerabilities from 10 popular client applications. The authors observed that the delay of patch releases between a reused code snippet in two applications is a median of 11 days up to 118 days. Li and Paxson (2017) investigated the patch release process and characterized security patches from a diverse set of 682 open-source software projects, including more than 4k bug fixes for over 3k vulnerabilities. The study revealed that the fixes

for 78.8% of all CVEs were committed on the disclosure date, while 50% of them were patched more than a week before the disclosure date. In our work, we study the delay of patch releases w.r.t the disclosure date, which is the most risky time in a vulnerability lifecycle, and characterize the influential factors related to the delay.

Another line of research is related to patch identification and propagation in various sets of open-source projects, i.e., program languages (e.g., Python (Ruohonen, 2018), JavaScript (Chinthanet et al., 2021)), devices (e.g., IoT (Nakajima et al., 2019), Android (Wu et al., 2019; Farhang et al., 2019)), or diverse sets of projects (Wang et al., 2019; Machiry et al., 2020; Chinthanet et al., 2021). For example, Wang et al. (2019) developed a machine learning model to identify hidden security patches in open-source projects, i.e., patches without explicit description in change logs or release notes. These patches could be adopted by similar types of open-source projects that have the same vulnerability and are unpatched. Our study focuses on vulnerabilities in products of top companies (CNAs) in security and investigates their practices.

5.3.2 Large-scale Vulnerability Analysis

Due to the increase in availability of big data technology in recent years, many prior works have conducted analyses on a large-scale set of vulnerabilities. Liu et al. (2020) studied the characteristics (e.g., developer, blamed code) of vulnerabilities from the patches of 3,806 vulnerabilities in 5 popular open source projects, written in C/C++. For example, vulnerable code are usually introduced by less than 10% of developers in a project. Shahzad et al. (2012) characterized vulnerabilities from a wide range of aspects, including when they were introduced, how they evolved over time and which types of vulnerabilities lead to a higher number of exploits. Feutrill et al. (2020) showed

that the time series of public vulnerability disclosures have a long range dependence, with burstiness, high variation, and slow convergence. Many prior work relied on their own set of vulnerabilities to develop relevant tools, e.g., detection (Grieco et al., 2016; Dolan-Gavitt et al., 2016; Li et al., 2021) prediction (Chen et al., 2019; Shin et al., 2010), notification (Stock et al., 2016), etc.

5.3.3 Multi-party coordination of CNAs for vulnerability fix releases and disclosure

Differentiating from the aforementioned prior work, we study the efficiency of patch releases from top companies that control the vulnerability disclosure process, as a CNA. As discussed in Section 5.2.4, these CNAs have expertise in security, compared to ordinary software projects/vendors, and are involved in multi-party coordination on fixing vulnerabilities.

Although the number of vulnerability exploits surges quickly after disclosure and drops quickly after patching (Arora et al., 2004), in such a complex coordination model, the involved parties face many challenges. First of all, each party has its own strategy for dealing with vulnerabilities, development policy and release cycles. Second, these policies and release cycles prevent simultaneous rollout of fixes from all affected vendors at once. The more time passes since the release of the first fixed product, the greater the pressure for the affected vendors that have not yet developed or integrated a fix into their product. In this work, we call the time between the rollout of the first and the last product impacted by a vulnerability, the vulnerability's "*patch coherence*". Moreover, when an affected vendor is not able to fix a vulnerability, the vendor seeks

help from another vendor, leading to an increase in involved parties and further complicating the fixing process. Furthermore, a vulnerable project might have its own derived projects that are blocked until the vulnerable project releases the fix, again increasing the difficulty of finding a coordinated schedule and delaying the release of fixes.

Given the special role of CNAs in multi-party coordination on fixing vulnerabilities, this work focuses on empirically understanding the coordination practices employed by the top CNAs.

5.4 Study Design

This section presents the approach for our empirical study to understand the coordination efficiency of CNAs as software vendors. RQ1 studies the amount of self-assigned and delegated vulnerabilities and their threat levels. RQ2 measures whether the delay of patch releases for self-assigned and delegated vulnerabilities is influenced by the type of the affected product. Finally, in order to understand the efficiency of collaboration across CNAs, RQ3 studies the delay of patch releases between CNAs that are affected by the shared vulnerabilities.

5.4.1 Subject CNA Selection

As discussed in Section 5.2.4, CNAs are responsible for assigning, validating and disclosing vulnerabilities, while software vendors are responsible for developing and releasing vulnerability fixes. To gain insights from current practices, we selected 13 CNAs who are also software vendors and amongst the top 20 CNAs in terms of the number

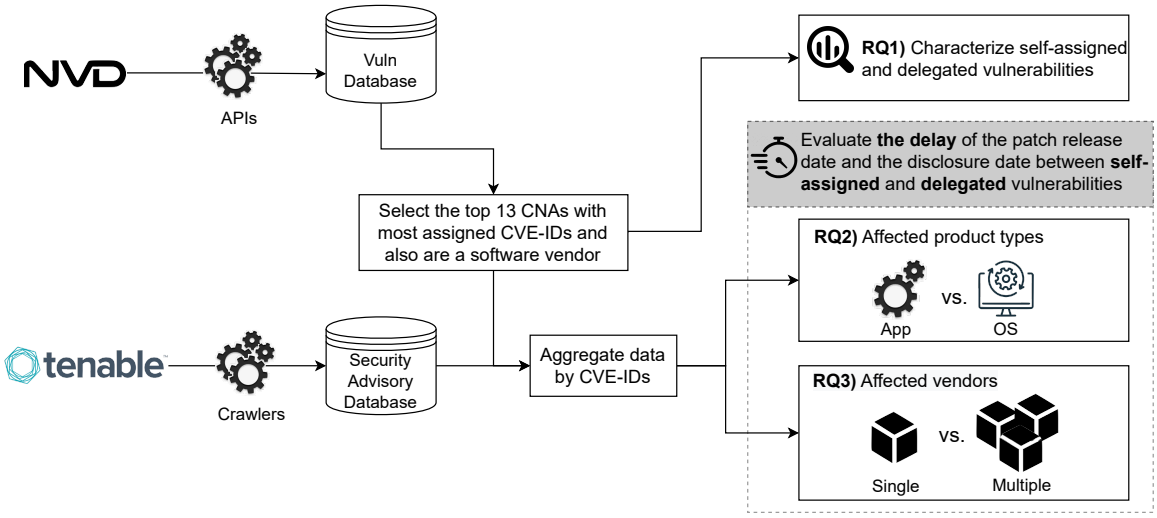


Figure 5.2: Overview of our study approach.

of CVE IDs they assigned. Table 5.1 presents the top 20 CNAs along with the number of assigned CVEs, the affected product types, the month of their first assigned CVE ID, and their organization type. Table 5.1 shows that 14 out of the top 20 CNAs are software vendors. 4 out of 20 are organizations regarding security (i.e., MITRE and CERT), 1 is a bug bounty platform and 1 represents a group of corporations related to the Android platform. We exclude the #10 ranked CNA related to the Android platform since it involves several corporations. We unify vulnerabilities assigned by Talos (#16) with those assigned by Cisco (#6) since Talos is an intelligence group of Cisco. Finally, we obtain a total of 13 CNAs that are also software vendors for our study.

5.4.2 Data Collection

Figure 5.2 presents an overview of our study approach. Our dataset consists of vulnerability reports from the NVD and their security advisories from `tenable.com`.[7] A

[7] https://www.tenable.com/

Table 5.1: The top 20 CNAs with the most valid CVE IDs assigned. Note that the three RQs study the bold CNAs, which are both CNAs and software vendors. Note that a vulnerability may affect several CNAs' products, i.e., both self-assigned and delegated vulnerabilities, so that the sum of the 4th and 5th columns is greater than the unique number of the studied vulnerabilities that affected the 13 CNAs, i.e., 39,409.

	CNAs	**# of CVEs**	**# of delegated CVEs**[9]	**# of self-assigned CVEs**[8]	**Affected product types**[6]	**Type**[7]
1	MITRE	44,383	3,939	0	-	NP
2	**Red Hat**	7,331	333	1,466	A.O	C
3	**Oracle**	6,076	218	5,529	A.H.O	C
4	**Microsoft**	5,786	13	5,438	A.H.O	C
5	**IBM**	4,241	209	4,113	A.H.O	C
6	**Cisco systems**	4,098	32	4,069	A.H.O	C
7	**Apple**	3,906	481	3,827	A.H.O	C
8	**Adobe**	3,583	2,716	3,224	A	C
9	CERT/CC[1]	2,635	105	0	-	N
10	Android[2]	2,607	2,331	0	-	C
11	**Qualcomm**	1,722	597	1,124	A.H.O	C
12	**Google**	1,719	576	1,361	A.H.O	C
13	ICS-CERT[3]	1,714	10	0	-	N
14	JPCERT/CC[4]	1,655	93	0	-	N
15	**Mozilla**	1,278	458	1,201	A.H.O	C
16	Talos[5]	962	0	49	A.H.O	C
17	HackerOne	920	30	0	-	B
18	**Dell**	848	31	263	A.H.O	C
19	**Intel**	770	31	669	A.H.O	C
20	**Huawei**	758	1	732	A.H.O	C

[1] A computer emergency response team (CERT) is an expert group that handles computer security incidents. CERT/CC represents the CERT Coordination Center.
[2] Android indicates a group of people associated with Google Inc. or the Open Handset Alliance.
[3] ICS-CERT represents the CERT for industrial control systems (ICS).
[4] JPCERT/CC represents the CERT in Japan.
[5] Talos is an intelligence group in Cisco.
[6] 'A' represents application product type, 'H' represents hardware and 'O' represents operating system.
[7] Organizational type: 'C' represents commercial, 'N' represents not-profit, 'B' represents bug bounty platform and 'NP' represents not-for-profit.
[8] The CNAs that are not a software vendor do not have self-assigned CVEs.
[9] Note that the delegated CVEs only affected the products of the other studied CNAs.

vulnerability report includes the information (e.g., CVE ID, description) related to a vulnerability (cf. Section 5.2), while a security advisory indicates how to fix the vulnerable product of a particular affected vendor. We only consider vulnerability reports with a valid CVE ID.[8] First, we extract 106,828 vulnerabilities, i.e., unique CVE IDs, that were assigned by the top 20 CNAs and whose CVE ID was issued between Jan. 2010 and Dec. 2020, before we remove 9,836 invalid vulnerabilities that were rejected, disputed or duplicated, i.e., 96,992. The numbers in the 3rd column of Table 5.1 present the numbers of valid CVE IDs assigned by the top 20 CNAs. As discovering vulnerabilities takes time and developers usually take more than two years to fix them after the discovery (Ozment and Schechter, 2006), we decided to keep a safety margin of 1 year for developing vulnerability fixes between the end of our data set (Dec. 2020) and the time at which we started this work (Jan. 2022). To better understand the efficiency of vulnerability fix releases for both self-assigned and delegated vulnerabilities, we then identify delegated vulnerabilities in each CNA amongst the vulnerabilities assigned by all the 20 CNAs, while discarding the vulnerabilities that did not affect any product of the 13 studied CNAs. Finally, our dataset contains a total of 39,409 vulnerabilities.

For each of these 39,409 vulnerabilities, we extract the corresponding vulnerability report from the NVD official data feeds.[9] Such a vulnerability report includes its CVE ID, assignor (CNA), CVSS scores, published date (i.e., disclose to the public), and affected software configurations according to the CPE specification (discussed in Section 5.2.3).

Furthermore, we also extract patch dates from security advisories in `tenable.`

[8] https://cve.mitre.org/cve/list_rules_and_guidance/correcting_counting_issues.html

[9] https://nvd.nist.gov/vuln/data-feeds

com.[10] Tenable.com is a vulnerability exposure company that offers tools for over 30k companies around the world to protect their systems from security risks. Tenable.com organizes security advisories that are made by a software vendor who released a fix (e.g. security advisory, patch) for a given vulnerability. A security advisory in Tenable.com includes its CVE ID(s), affected software configurations (CPEs), and its patch publication date. Note that a security advisory might contain several vulnerabilities when a software vendor releases the fixes of these vulnerabilities in one security update.

We aggregate the vulnerability reports and security advisories by CVE IDs and affected software configurations (CPEs).

5.4.3 Efficiency measurement

To study the efficiency of CNAs in multi-party coordination on vulnerability fix releases and disclosure, we measure the efficiency of a CNA (as a software vendor) from three aspects: 1) the awareness of vulnerabilities of its own products in RQ1, 2) the delay of patch releases w.r.t. the disclosure date in RQ2 and RQ3, i.e., Diff_{pd}, and 3) the delay between the first and last patch releases across the CNAs when they are affected by a shared vulnerability in RQ3, i.e., its patch coherence Diff_{pp} (see Section 5.3.3).

First, as discussed in Section 5.2.4, since a CNA represents an expert in vulnerabilities, identifying self-assigned vulnerabilities reflects the extent of efficiency in finding

[10]Note that we use tenable.com, since the open-source vulnerability database (OSVDB), which was widely used to study the patch available date for security vulnerabilities (Frei et al., 2006; Shahzad et al., 2012), was shutdown permanently in 2016, https://www.securityweek.com/osvdb-shut-down-permanently.

vulnerabilities in its own products. In addition, self-assigned vs. delegated vulnerabilities are treated in terms of the capability of a given CNA to validate whether a vulnerability affects its own product vs. other CNAs' products.

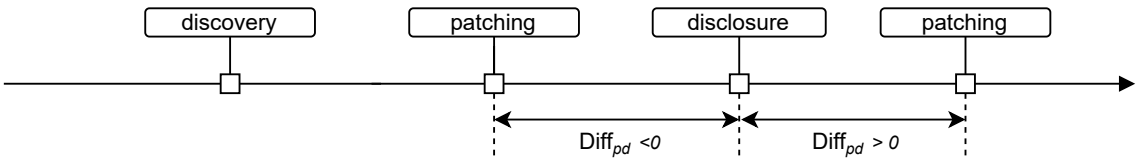


Figure 5.3: Illustration of the measurement of Diff$_{pd}$. The Diff$_{pd}$ of a vulnerability is calculated by subtracting the disclosure date from the patch (available) date. A negative value (Diff$_{pd}$ <0) indicates the patch is available before the disclosure date.

Second, for a given CNA, the delay (Diff$_{pd}$) of patch releases of a vulnerability represents the time between the disclosure date of a vulnerability and the availability of a patch for it, as shown in Figure 5.3. The most risky vulnerabilities are those that are disclosed to the public without an available fix (Shahzad et al., 2012), in which case the Diff$_{pd}$ is positive. In fact, in many cases, an exploitation method is provided when a vulnerability is disclosed (Sabottke et al., 2015), enabling anyone to exploit the vulnerability. In other words, the delay Diff$_{pd}$ indicates the size of the window during which attackers could actively exploit the vulnerability at a large scale. A negative value of Diff$_{pd}$ indicates that a fix (patch) was released before the disclosure date of the vulnerability, which is the optimal case for users to avoid vulnerability exploits. The larger the value of Diff$_{pd}$, the more risky a victim system is.

We compare the delay (Diff$_{pd}$) from several aspects within one CNA and across the studied CNAs: self-assigned vs. delegated, affected product types, and the number of affected vendors (CNAs). We detail our approaches in the section of each RQ.

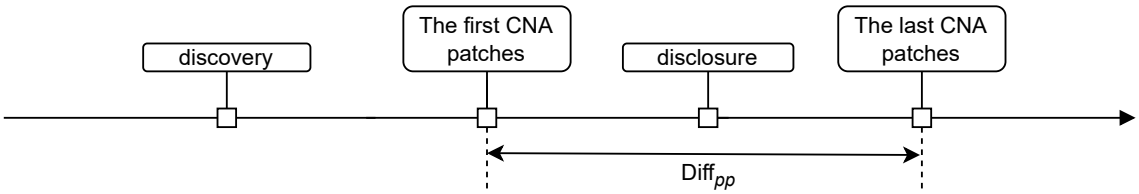


Figure 5.4: Illustration of the measurement of Diff$_{pp}$ (patch coherence).

Lastly, as a vulnerability could affect several CNAs' products, these CNAs in principle should coordinate vulnerability fix releases and disclosure. We measure such coordination across the affected CNAs by the metric of "*patch coherence*", which is the delay (Diff$_{pp}$) between the first and last patch releases across the CNAs, as shown in Figure 5.4. A small value of patch coherence indicates a more consistent speed of patch releases from the affected vendors, suggesting that users of all affected products become protected in the same period. We compare the delay (Diff$_{pp}$) from several aspects: self-assigned vs. delegated, vulnerability weaknesses, and pairs of affected CNAs. We detail our approaches in RQ3.

5.5 Results

RQ1: What are the characteristics of self-assigned vs. delegated vulnerabilities?

Motivation: Given the rise of security issues within software supply chains, fixing vulnerabilities requires multi-party coordination (as discussed in Section 5.2.4). Delegated vulnerabilities indicate not only a potential delay of becoming aware of the vulnerabilities, but also indicate that users might wait for a longer time to get the fixes.

This increases the chance of vulnerability exploits and results in disruption in software supply chains and promoting conflicts between their users. Understanding the characteristics of self-assigned and delegated vulnerabilities gives insights to software vendors when developing fixes and to users on securing their systems.

Approach: We measure the prevalence of self-assigned and delegated vulnerabilities by calculating the number and percentage of vulnerabilities in each group for each of the 13 CNAs. To analyze the characteristics of self-assigned and delegated vulnerabilities, we investigate the affected product types (i.e., application, hardware and operating system) from the CPE configurations, the severity level using the CVSS 2.0, and the assignor-assignee pairs of the CNAs.

Results: 12 out of the 13 studied CNAs (i.e., all except for Google) have 3 up to 49 times as many self-assigned vulnerabilities as delegated ones. Google instead has a majority (77%) of delegated vulnerabilities. In general, the CNAs seem to dedicate substantial effort on their own products. More specifically, Figure 5 indicates that Apple, Microsoft and Red Hat are the CNAs with a similar percentage of self-assigned and delegated vulnerabilities, i.e., Apple: 53% vs. 47%, Microsoft: 53% vs. 47%, Red Hat: 50% vs. 50%.

Mitre (not one of the 13 CNAs) has assigned vulnerabilities to all the 13 CNAs. Figure 5.6 shows the heat map of the assignor and assignee pairs of the 13 CNAs (and Mitre) for delegated vulnerabilities. In terms of being an assignor, Mitre is the #1 CNA

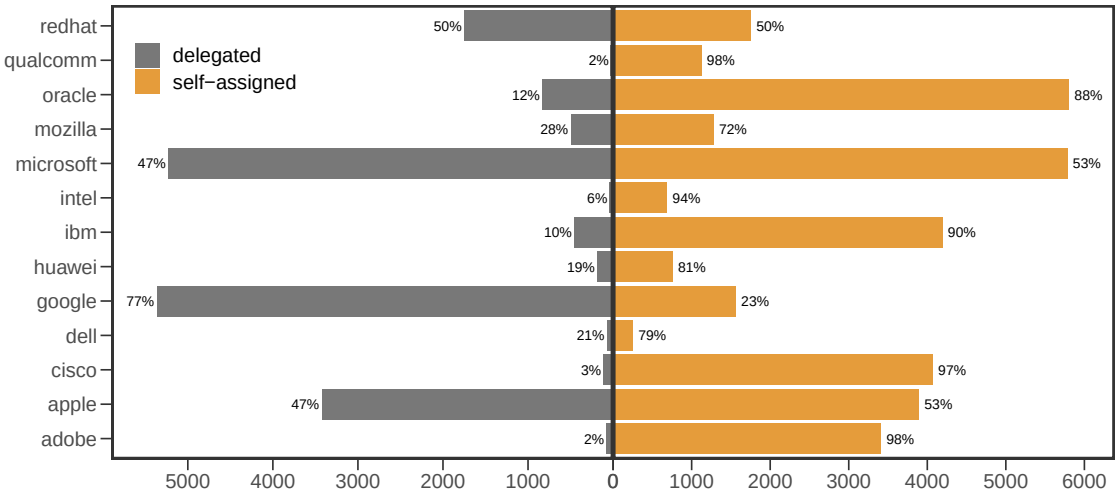


Figure 5.5: The number and percentage of self-assigned vs. delegated vulnerabilities in the 13 CNAs.

followed by Red Hat and Dell, who assigned vulnerabilities to 11 and 8 CNAs, respectively. Mitre has focused on software security for more than two decades and won several awards and notable achievements.[11] Red Hat[12] and Dell[13] have a sterling reputation in software security. Adobe assigned vulnerabilities to 6 CNAs and has assigned the largest number of vulnerabilities to Microsoft (2,700) and Apple (2,516) among all the assignors.

On the other hand, Red Hat, Google and Apple are the assignee of 12 CNAs, followed by Microsoft who is the assignee of 11 CNAs. Among these four CNAs, Red Hat has the smallest number of delegated vulnerabilities (1,555), compared to the other three CNAs (Google: 2,481, Apple: 3,296, Microsoft: 4,418). Red Hat and Google have the largest number of vulnerabilities delegated from Mitre, while Apple and Microsoft get them from Adobe. In addition, Qualcomm only has one assignor, i.e., Mitre.

[11] https://www.mitre.org/about/awards-and-recognition
[12] https://www.redhat.com/ja/about/press-releases/press-bestlinuxsolutions
[13] https://www.dell.com/en-us/blog/dell-software-security-solutions-of-awards-and-honors/

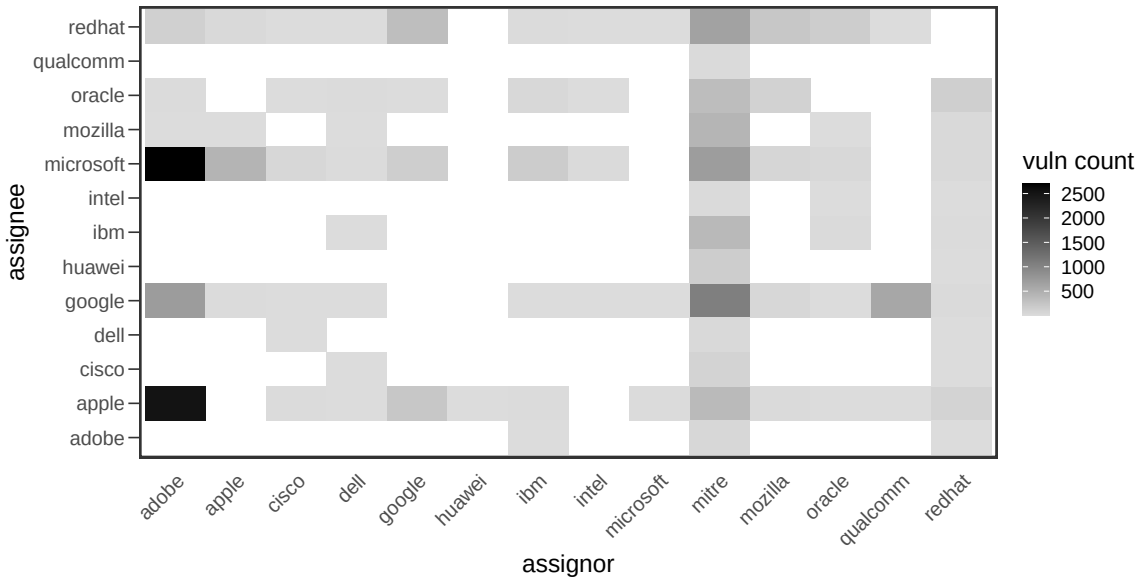


Figure 5.6: Heat map of the assignor-assignee pairs based on the number of delegated vulnerabilities. Note that Mitre is the #1 assignor but is not included in the studied CNAs.

For the application type of product, the 12 CNAs have 2 up to 49 times as many self-assigned vulnerabilities as delegated vulnerabilities. For the operating-system type of product, Google, Microsoft, Red Hat, and Oracle have a majority of delegated vulnerabilities (98%, 58%, 67%, and 55%, respectively), while Apple has a similar percentage of delegated (49%) and self-assigned (51%) vulnerabilities. In general, the application type of product is responsible for the largest percentage (47%) of vulnerabilities across the CNAs, while the hardware type of product accounts for the smallest percentage (18%) of vulnerabilities. Both application and hardware types of products have a median of 3 vulnerabilities across the CNAs. The unique number of affected hardware products (i.e., 9,364) of the studied affected vendors is approximately three times as high as that of affected application products (i.e., 3,346). Figure 5.7 shows the percentages of self-assigned and delegated vulnerabilities in the three

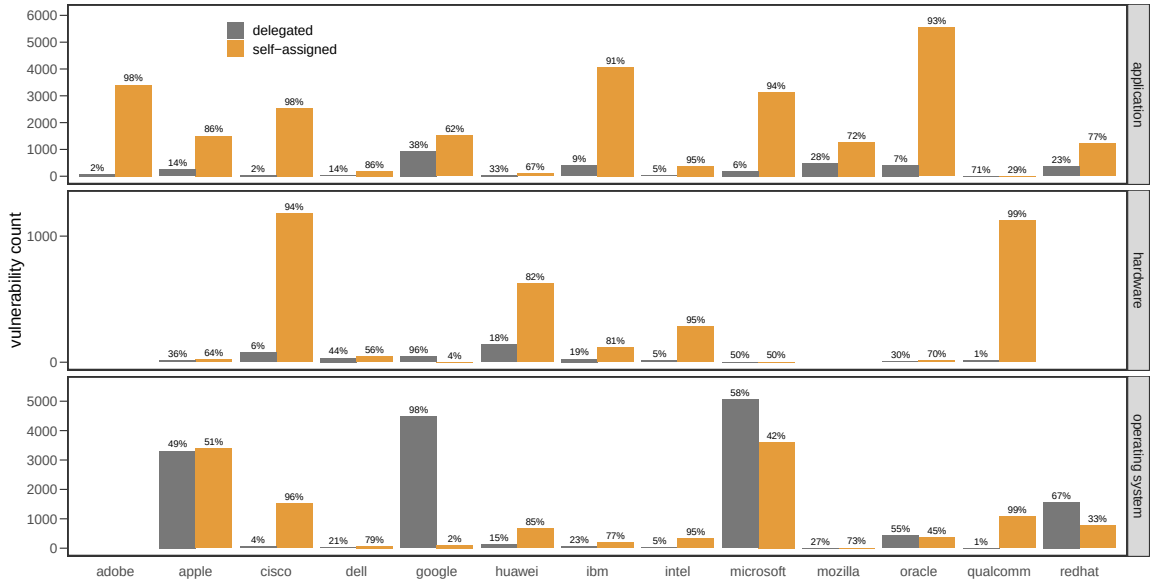


Figure 5.7: The number and percentage of self-assigned vs. delegated vulnerabilities in the 13 CNAs, based on the three product types, i.e., application, hardware and operating-system.

product types for each of the studied CNAs. In 12 out of the 13 CNAs, the majority (at least 62%) of vulnerabilities affecting the application type of product are self-assigned, compared to 8 out the 13 CNAs for the operating-system type of product.

Delegated vulnerabilities are more severe than self-assigned vulnerabilities. For the application type of product, delegated vulnerabilities have a median CVSS score of 7.5, i.e., high-severity, while self-assigned vulnerabilities have a median CVSS score of 6.8 (medium-severity) in Figure 5.8. The difference between delegated and self-assigned vulnerabilities is statistically significant by a Wilcoxon test, with a p-value $<$ 0.01 ($\alpha = 0.01$). The effect size is small (0.26). For the operating-system type of product, although delegated and self-assigned vulnerabilities both have a median CVSS score of 7.1, delegated vulnerabilities have higher first and third quartiles than self-assigned

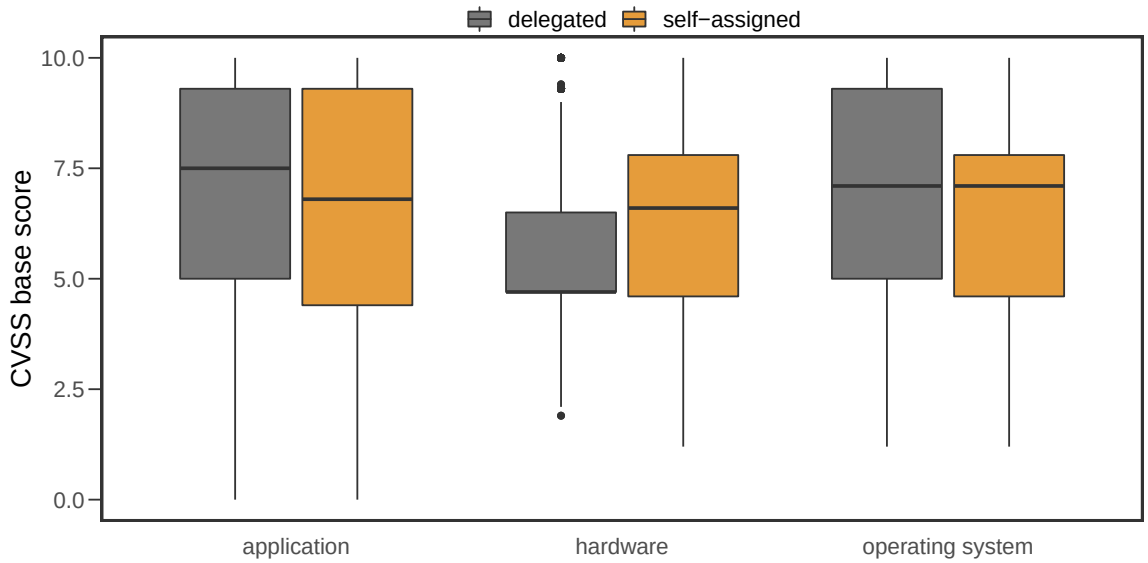


Figure 5.8: For the application type of products, delegated vulnerabilities have a higher CVSS 2.0 score than self-assigned vulnerabilities.

vulnerabilities, i.e., both the top and bottom half of the distribution of delegated vulnerabilities are more severe than that of self-assigned vulnerabilities.

12 out of the 13 studied CNAs have a higher percentage of self-assigned vulnerabilities than delegated vulnerabilities, though 3 of them have a similar percentage of both types of vulnerabilities. The studied CNAs have a majority of self-assigned vulnerabilities for the application type of product, compared to the hardware and operating-system types of products. However, delegated vulnerabilities are more severe than self-assigned vulnerabilities for the application and operating-system types of products across the CNAs.

RQ2: How fast do the CNAs develop fixes for self-assigned vs. delegated vulnerabilities?

Motivation: The results in RQ1 reveal that the studied CNAs have a majority of self-assigned vulnerabilities. However, delegated vulnerabilities have a higher threat level, suggesting a higher impact when such a vulnerability is disclosed without an available fix. We aim to understand the speed of releasing vulnerability fixes in these CNAs, the duration of the period users are at risk, and whether the threat level of a vulnerability influences the speed of patch releases.

Approach: As discussed in Section 5.4.3, we compute the delay (Diff_{pd}) of patch releases for each vulnerability. To measure the speed of patch releases, we classify the delay into 5 duration categories (w.r.t. the disclosure date): $<$ -30, -30∼0, 0∼30, 30∼90, and $\geqslant$ 90 days and calculate the number of self-assigned and delegated vulnerabilities in each category for the studied CNAs. These five categories represent earlier-than-one-month before the disclosure date, within-one-month before the disclosure date, within-one-month after the disclosure date, one-quarter after the disclosure date, and longer-than-one-quarter after the disclosure date.

We compare the Diff_{pd} delay between the **application** and **operating-system** types of products, since the hardware type of product only accounts for the minority (13%) of vulnerabilities and `tenable.com` does not have security advisories for vulnerabilities that affected this type of product. Note that Qualcomm has insufficient data in `tenable.com` because it has the majority of its vulnerabilities in the hardware type of product (as shown in Figure 5.7). Therefore, we only study 12 out of the 13 CNAs in this RQ. In addition, we compute the CVSS scores of the vulnerabilities for the two product types.

Results: For the *application* type of product, the CNAs release fixes for delegated vulnerabilities at a slower pace than self-assigned vulnerabilities by a median of 1 day, yet typically still before disclosure. In particular, self-assigned vulnerabilities are fixed in a median of 2 days before the disclosure date across the CNAs, while delegated vulnerabilities are fixed in a median of 1 day before the disclosure date. The difference of the delay is significant by a Wilcoxon test, with a p-value < 0.01 and a medium effect size of 0.53.

One possible reason why patches for delegated vulnerabilities still appear before the disclosure date is that, even though there is a potential delay of becoming aware of vulnerabilities, the vulnerability has a low severity and can be fixed easily. Another possible reason is that the CNA (as a coordinator) and the affected software vendors agreed on a schedule to only disclose the vulnerability once the fixes are released. It is also possible that a vulnerability prompts the affected vendor to re-design certain functionalities in the vulnerable product, then only to publicly disclose upon release of the first re-design product version (even if this causes substantial delays). For example, Adobe was aware of CVE-2016-4271 approximately six months before the disclosure date.[14] This CVE was a flaw in Adobe Flash allowing to bypass the local sandbox and exfiltrate important data on Windows (e.g., user credentials). Adobe redesigned Flash Player 23 to no longer use a sandbox.

For the application type of product, all the studied CNAs release the fixes for most of the **self-assigned** vulnerabilities within 30 days before the disclosure date, while 2 of the CNAs release fixes for most of the **delegated** vulnerabilities at least 90 days after the disclosure date ($\geqslant$ 90 days). Table 5.2 shows all details of the 5 duration categories

[14] https://blog.bjornweb.nl/2017/02/flash-bypassing-local-sandbox-data-exfiltration-credentials-leak/

Table 5.2: The number of self-assigned and delegated vulnerabilities that affect the **application** type of product for the 5 duration (days) categories of the Diff_{pd} delay for the studied CNAs. For each row, the bold number indicates the category with the largest number of respective vulnerabilities among the 5 duration categories.

CNA	**Assignor**	**< -30**	**-30∼0**	**0∼30**	**30∼90**	**⩾ 90**
Adobe	Self-assigned	447	**2,609**	366	97	12
	Delegated	5	**33**	8	3	3
Apple	Self-assigned	177	**686**	358	160	114
	Delegated	3	33	26	41	**88**
Cisco	Self-assigned	55	**380**	138	5	6
	Delegated	**1**	-	-	-	-
Dell	Self-assigned	1	**3**	-	-	-
	Delegated	**2**	1	1	-	-
Google	Self-assigned	586	**856**	136	3	-
	Delegated	48	**576**	67	2	-
Huawei	Self-assigned	-	-	-	-	-
	Delegated	-	-	1	1	1
IBM	Self-assigned	141	**451**	101	84	101
	Delegated	42	**66**	22	22	25
Intel	Self-assigned	-	-	-	-	-
	Delegated	-	**2**	-	-	-
Microsoft	Self-assigned	16	**1,104**	401	11	7
	Delegated	1	**10**	7	8	5
Mozilla	Self-assigned	584	**731**	97	25	1
	Delegated	85	**343**	47	6	5
Oracle	Self-assigned	226	**2,775**	730	94	208
	Delegated	31	26	44	66	**175**
Red Hat	Self-assigned	144	**234**	94	60	99
	Delegated	10	34	46	**79**	67

of the Diff_{pd} delay for self-assigned and delegated vulnerabilities. In general, the CNAs are anticipated to deal better with self-assigned vulnerabilities than delegated vulnerabilities in terms of the delay of patch releases.

Although there is a potential delay of becoming aware of delegated vulnerabilities,

6 out of the 12 CNAs release the fixes for most of their delegated vulnerabilities within 30 days before disclosure. On the other hand, 1 CNA (i.e., Red Hat) requires 30∼90 days after disclosure (median of 42 days), and 2 CNAs (i.e., Apple and Oracle) even more than 90 days after disclosure (median of 117 and 120 days, respectively). Due to insufficient vulnerabilities for Dell, Intel and Huawei, we encourage future studies to collect more data to analyze the generalizability of our results.

For the *operating-system* type of product, the CNAs again release fixes for delegated vulnerabilities at a slower pace than self-assigned vulnerabilities, by a median of 5 days. More specifically, the fixes for delegated vulnerabilities are released in a median of 3 days after the disclosure date, while the fixes for self-assigned vulnerabilities are released in a median of -2 days before the disclosure date. Table 5.3 indicates that 8 out of the 12 CNAs release fixes for most of the **self-assigned** vulnerabilities within 30 days before the disclosure date (-30∼0 days), 1 CNA (Mozilla) more than 30 days before the disclosure date (< -30 days), and 1 CNA (Red Hat) more than 90 days after the disclosure date (⩾ 90 days). In contrast, for **delegated** vulnerabilities, 3 out of the 12 CNAs release fixes for the largest number of vulnerabilities within 30 days before the disclosure date (-30∼0 days), 3 CNAs (i.e., Apple, Oracle, and Red Hat) are more than 90 days after the disclosure date (⩾ 90 days), and 4 CNAs are more than 30 days before the disclosure date (< -30 days), though the numbers for these 4 CNAs are relatively small.

The delay ($\mathbf{Diff}_{pd}$) of patch releases for delegated vulnerabilities that affect the *operating-system* type of product is larger than for those that affect the *application* type of product, by a median of 4 days. In particular, only for high-severity **delegated** vulnerabilities, the delay (Diff_{pd}) of patch releases for the operating-system type of

Table 5.3: The number of self-assigned and delegated vulnerabilities that affect the **operating-system** type of product for the 5 duration (days) categories of the Diff_{pd} delay for the studied CNAs. For each row, the bold number indicates the category with the largest number of respective vulnerabilities among the 5 duration categories.

CNA	**Assignor**	**< -30**	**-30∼0**	**0∼30**	**30∼90**	**⩾ 90**
Adobe	Self-assigned	-	-	-	-	-
	Delegated	-	-	-	-	-
Apple	Self-assigned	1,086	**1,861**	223	231	70
	Delegated	32	33	27	94	**112**
Cisco	Self-assigned	27	**338**	281	4	39
	Delegated	**2**	1	-	-	-
Dell	Self-assigned	-	-	-	-	-
	Delegated	**2**	-	-	-	-
Google	Self-assigned	3	**10**	-	-	-
	Delegated	11	**23**	-	-	-
Huawei	Self-assigned	4	**6**	-	-	-
	Delegated	**2**	-	-	-	-
IBM	Self-assigned	11	**22**	2	-	1
	Delegated	3	**6**	2	-	1
Intel	Self-assigned	-	**1**	-	-	-
	Delegated	-	-	-	-	-
Microsoft	Self-assigned	7	**2,031**	616	25	5
	Delegated	5	**206**	89	13	3
Mozilla	Self-assigned	**3**	1	-	-	-
	Delegated	**3**	-	-	-	-
Oracle	Self-assigned	1	**90**	56	13	15
	Delegated	1	4	8	18	**30**
Red Hat	Self-assigned	4	6	6	10	**42**
	Delegated	-	5	25	31	**89**

product is similar to the application type of product (Wilcoxon test: p-value = 0.23) in Figure 5.9. For medium-severity **delegated** vulnerabilities, the delay (Diff_{pd}) of patch releases for the operating-system type of product is longer than the application type

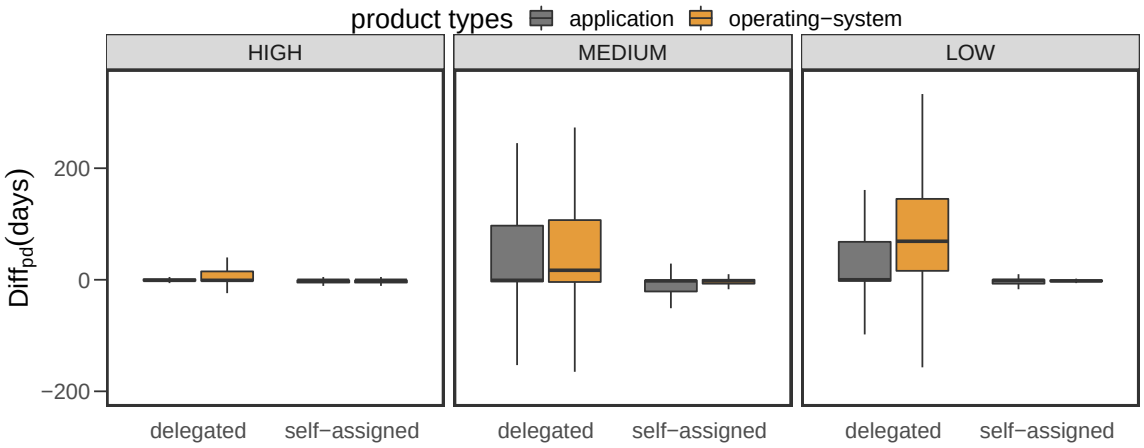


Figure 5.9: The comparison of the delay (Diff$_{pd}$) of releasing patches between the severity levels (i.e., high, medium and low), self-assigned vs delegated, and the product types across the CNAs.

of product by a median of 18 days (Wilcoxon test: p-value < 0.01), with a negligible effect size of 0.04. For low-severity **delegated** vulnerabilities, the gap of the delay (Diff$_{pd}$) between the two product types is a median of 69 days (Wilcoxon test: p-value < 0.01), with a small effect size of 0.3. Hence, the Diff$_{pd}$ delay of delegated vulnerabilities in operating-systems-related products interact with the severity level of the vulnerabilities. The higher the severity of a delegated vulnerability, the smaller the Diff$_{pd}$ delay, i.e., a quicker fix.

On the other hand, for **self-assigned** vulnerabilities, the delay of vulnerabilities that affect the application type of product is similar to those that affect the operating-system type of product across all severity levels (i.e., high, medium, and low). According to a Chi-square test, the severity levels and assignor (i.e., self-assigned or delegated) of a vulnerability are two independent variables (p-value < 0.05). When a vulnerability in a product is found by the assignor (i.e., the product owner), the Diff$_{pd}$ delay is similar across all severity levels for both product types, typically close to the disclosure date.

Across the two product types, the studied CNAs release fixes for self-assigned vulnerabilities faster than for delegated vulnerabilities. The delay (Diff_{pd}) of patch releases for self-assigned vulnerabilities does not correlate with the vulnerabilities' severity level, while for delegated vulnerabilities, the lower the severity level, the slower the patch is released. Furthermore, the gap of the delay between the severity levels in vulnerable operating-systems-type products is larger than in application-type products.

RQ3: How effectively do the CNAs coordinate to release fixes for shared vulnerabilities?

Motivation: The results in RQ1 and RQ2 indicate that delegated vulnerabilities are more severe than self-assigned vulnerabilities and the delay of patch releases for delegated vulnerabilities is longer than self-assigned vulnerabilities. Furthermore, delegated vulnerabilities indicate that there exists a potential delay before the affected CNAs are aware of these vulnerabilities, unless CNA coordination is able to mitigate it. In order to measure the coordination efficiency on fixing vulnerabilities, we focus on vulnerabilities that affect more than one CNA's products, which we refer to as *multi-affected vulnerabilities*.

Approach: To know the extent of efficiency of coordination across the CNAs, we first compare the number and percentage of vulnerabilities that affected more than one CNA's products to those that affect only one CNA (i.e., *single-affected vulnerabilities*) for the 5 duration categories of RQ2. We compare the three CVSS score metrics (i.e.,

base, impact, and exploitability) between multi-affected and single-affected vulnerabilities. Similar to RQ2, we compute the Diff_{pd} delay and compare the delay between multi-affected and single-affected vulnerabilities from two aspects: (1) self-assigned vs. delegated and (2) the affected product types (i.e., application vs. operating system vs. both). Vulnerabilities affecting products of both types indicate more complex issues for which the affected vendors need a longer time to (jointly) develop a fix, potentially having a higher impact on victim systems. Similar to RQ2, due to insufficient data for hardware-type products in `tenable.com`, we only study 12 out of the 13 CNAs (exclude Qualcomm).

In addition, as discussed in Section 5.4.3, we evaluate coordination efficiency across the CNAs by the Diff_{pp} delay between patch releases for shared vulnerabilities, i.e., *patch coherence*. In particular, we compute the Diff_{pp} delay for each of the multi-affected vulnerabilities across the studied CNAs. A large Diff_{pp} indicates that the CNAs are less effective in coordinating vulnerability fixes across their products, since the fixes are spread out across time, with some affected products being prone to exploitation of a vulnerability for a much longer time.

We analyze the delay (Diff_{pp}) between the first and last patch dates from the following three aspects:

1. **The number of affected variants**: A variant represents a flavour of a product designed for particular conditions. For example, Cisco Catalyst's campus LAN switches have several variants, such as the Catalyst 6500 series and Catalyst 7600 series that support different network layers. A higher number of affected variants indicates that the affected vendors need more time to adopt the fixes to all the affected variants of a particular product.

2. **Vulnerability weaknesses**: Since every vulnerability corresponds to a variety of weaknesses on a victim system, we study whether the delay of a particular type of weakness is longer than the others. We identify the top 10 weaknesses in the multi-affected vulnerabilities (see Table 5.5), based on the CWE (Common Weakness Enumeration) framework (cwe, 2022) that has been widely used in prior work (Blocken, 2014; Zhang et al., 2021), and compare the delay among the top 10 CWEs. The CWE framework lists common security weaknesses with their information, including description, code examples, detection methods, and potential mitigation. The CWE framework also classifies associated weaknesses in groups that developers and practitioners could then use as a common language for vulnerability discussions.

3. **Pairs of studied CNAs**: We are interested in whether certain pairs of CNAs have a closer coordination of vulnerabilities than others. As the results in RQ2 indicate that the Diff_{pd} delay varies across the CNAs (see Tables 5.2 and 5.3), we classify pairs of the studied CNAs that exhibit a similar pattern of releasing fixes. For example, a pair of CNAs that usually release fixes before the disclosure is more likely to coordinate on a fix, achieving better (lower) patch coherence. We consider any pairs from Adobe, Google, Microsoft and Mozilla as a group, which we refer to as "AGMM", since these CNAs release fixes of most of both types (i.e., self-assigned and delegated) of vulnerabilities across the two product types before disclosure (Tables 5.2 and Table 5.3). By contrast, Apple, Oracle and Red Hat have an inconsistent speed of releasing fixes, which we group together as "AOR". We assign the remaining pairs as "others". In addition, the AGMM and AOR groups represent two extreme groups of CNAs: one of them is quicker with releasing fixes, while

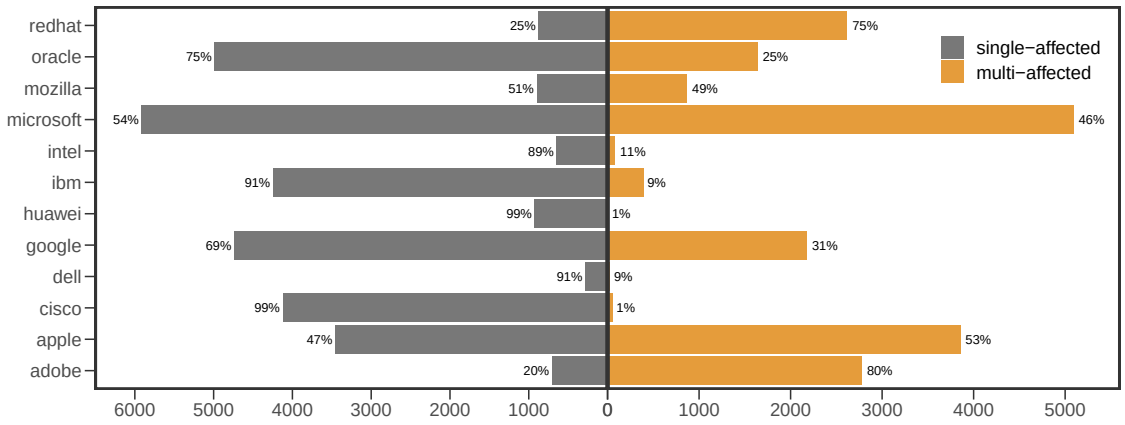


Figure 5.10: The number and percentage of single-affected and multi-affected vulnerabilities in each of the studied CNAs.

the other is slower, yielding a lower and upper bound of patch coherence across the studied CNAs.

Results: Adobe (80%), Apple (53%) and Red Hat (75%) are impacted by the majority of multi-affected vulnerabilities. Apple and two other CNAs (Microsoft and Mozilla) have a similar percentage (≈50%) of single-affected and multi-affected vulnerabilities, i.e., Apple: 53% vs. 47%, Microsoft: 46% vs. 54% and Mozilla: 49% vs. 51%. Figure 5.10 indicates that the other 9 out of 12 studied CNAs have a majority (at least 69%) of single-affected vulnerabilities.

The 5,339 multi-affected vulnerabilities that affect both product types are more severe than (1) those that affect one product type and (2) single-affected vulnerabilities (Kruskal-Wallis test: p-value < 0.05). More specifically, Figure 5.11 shows that multi-affected vulnerabilities that affect the two product types have a median base score of 9.3, a median exploitability score of 8.6 and a median impact score of 10, which suggests an extremely high-severity level. By contrast, multi-affected vulnerabilities that only affect one type of product (i.e., application or operating-system) are

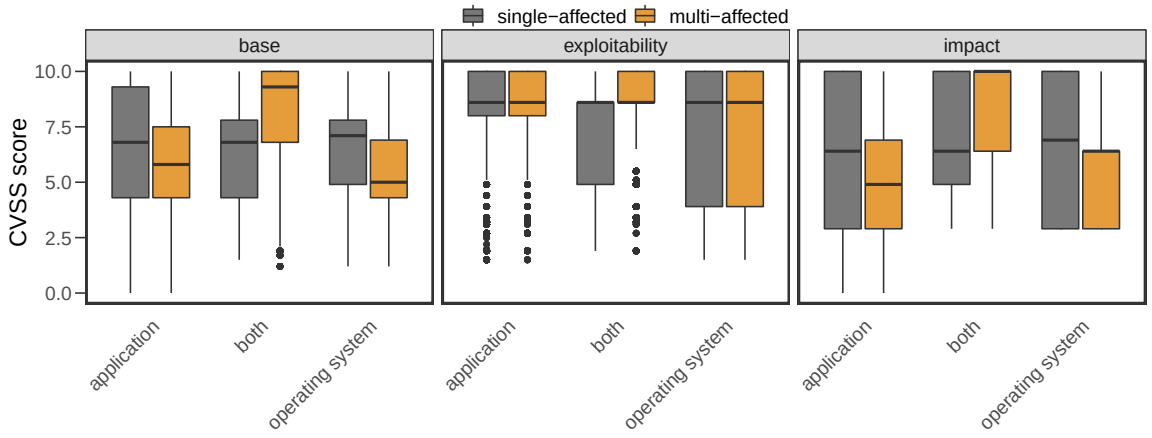


Figure 5.11: The three metrics of CVSS scores for single-affected and multi-affected vulnerabilities that affect one (i.e., either application or operating-system) or two types (i.e., both) of products.

less severe than single-affected vulnerabilities, with median base scores of 5.8 and 5, respectively. Such vulnerabilities have a similar median score across the three metrics of CVSS scores in Figure 5.11 (grey bars).

The delay (Diff$_{pd}$) of patch releases for multi-affected vulnerabilities is longer than for single-affected vulnerabilities, especially for delegated ones that are also multi-affected. For **delegated** vulnerabilities, the CNAs release fixes for **multi-affected** vulnerabilities in a median of 4 days (after the disclosure date), while they release fixes for **single-affected** vulnerabilities in a median of -2 days (before the disclosure date). For **self-assigned** vulnerabilities, the CNAs release fixes for both **single- and multi-affected** vulnerabilities in a median of -2 days (before the disclosure date). For an individual CNA, similar to the results for the application and operating-system types of products in RQ2 (Table 5.2 and 5.3), IBM, Apple, Oracle, and Red Hat release fixes for the largest number of **multi-affected and delegate** vulnerabilities as late as at least 90 days after the disclosure date ($\geqslant$ 90 days) in Table 5.4, which is slower than the other

Table 5.4: The comparison of the delay Diff_{pd} of patch releases between single-affected vs. multi-affected and delegated vs. self-assigned vulnerabilities for the studied CNAs. For each row, the bold number indicates the category with the largest number of vulnerabilities among the 5 duration categories.

CNA	Assignor	Type	< -30	-30∼0	0∼30	30∼90	⩾ 90
Adobe	Self-assigned	Single	26	**515**	35	3	5
		Multi	421	**2,094**	331	94	7
	Delegated	Single	3	**18**	1	2	2
		Multi	2	**15**	7	1	1
Apple	Self-assigned	Single	1,087	**2,150**	458	346	149
		Multi	176	**397**	123	45	35
	Delegated	Single	11	**36**	19	15	15
		Multi	24	30	34	120	**185**
Cisco	Self-assigned	Single	80	**723**	414	9	44
		Multi	4	**11**	10	-	1
	Delegated	Single	-	1	**4**	-	-
		Multi	1	1	-	-	-
Dell	Self-assigned	Single	-	1	**3**	-	-
		Multi	-	-	-	-	-
	Delegated	Single	4	1	-	4	-
		Multi	-	-	1	-	-
Google	Self-assigned	Single	260	**431**	83	1	-
		Multi	329	**435**	53	2	-
	Delegated	Single	27	**293**	42	1	-
		Multi	32	**306**	25	1	-
Huawei	Self-assigned	Single	4	**6**	-	-	-
		Multi	-	-	-	-	-
	Delegated	Single	**2**	-	-	-	-
		Multi	-	-	1	1	1
IBM	Self-assigned	Single	137	**422**	95	72	86
		Multi	15	**51**	8	12	16
	Delegated	Single	42	**65**	22	22	17
		Multi	3	7	2	-	**9**
Intel	Self-assigned	Single	-	**1**	-	-	-
		Multi	-	-	-	-	-
	Delegated	Single	-	**1**	-	-	-
		Multi	-	**1**	-	-	-
Microsoft	Self-assigned	Single	21	**3,126**	1,013	36	11
		Multi	2	**9**	4	-	1
	Delegated	Single	1	**17**	5	13	6
		Multi	5	**199**	91	8	2
Mozilla	Self-assigned	Single	231	**367**	40	21	-
		Multi	356	**365**	57	4	1
	Delegated	Single	69	**251**	15	3	2
		Multi	19	**92**	32	3	3
Oracle	Self-assigned	Single	132	**2,251**	436	59	118
		Multi	95	**614**	359	48	105
	Delegated	Single	6	**8**	2	3	6
		Multi	26	22	50	81	**199**
Red Hat	Self-assigned	Single	106	**174**	34	24	25
		Multi	42	66	66	46	**116**
	Delegated	Single	2	**4**	1	3	2
		Multi	8	35	70	107	**154**

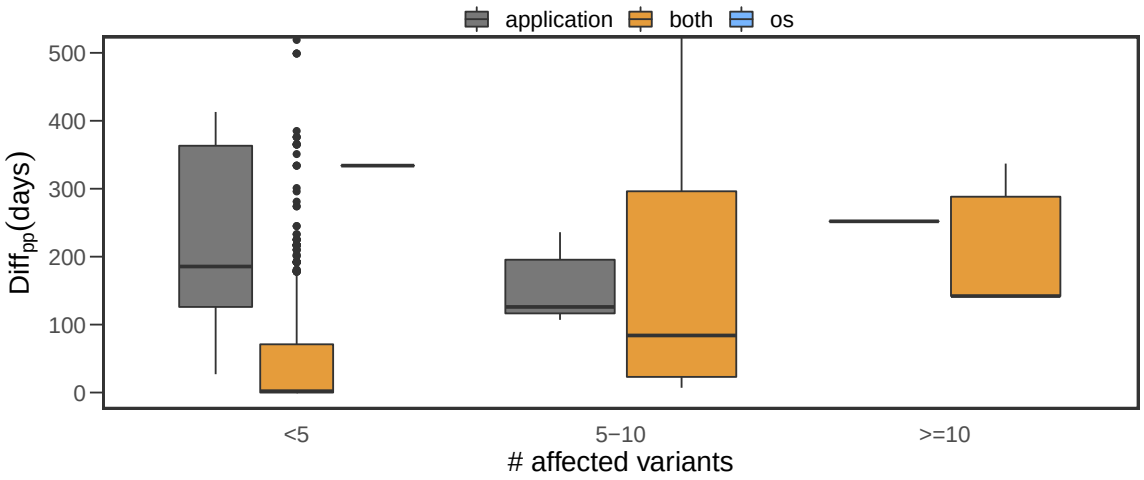


Figure 5.12: Comparison of the delay ($Diff_{pp}$) between when the first and last patches are available across the studied CNAs, based on affected product types and the number of affected variants. A variant represents a flavour of a product designed for particular conditions (e.g., x84 or x64).

studied CNAs.

The $Diff_{pp}$ delay (i.e., patch coherence) for multi-affected vulnerabilities that affected less than 5 variants of both product types is the shortest (median of 2 days), compared to vulnerabilities that affected one type of product (median of 186 days for the application type, and 334 for the operating system type), as shown in Figure 5.12. One possible reason for this is that multi-affected vulnerabilities that affected both types of products are more severe (see Figure 5.11). In general, the $Diff_{pp}$ delay of multi-affected vulnerabilities is a median of 35 days across the CNAs. Figure 5.12 also indicates that the more products affected by a multi-affected vulnerability, the longer the patch coherence.

The $Diff_{pp}$ delay (i.e., patch coherence) for vulnerabilities with weaknesses related to buffer overflow (CWE-119) and permissions, privileges, and access controls (CWE-264) is the shortest (median of 0 days), as shown in Figure 5.13. In general, the

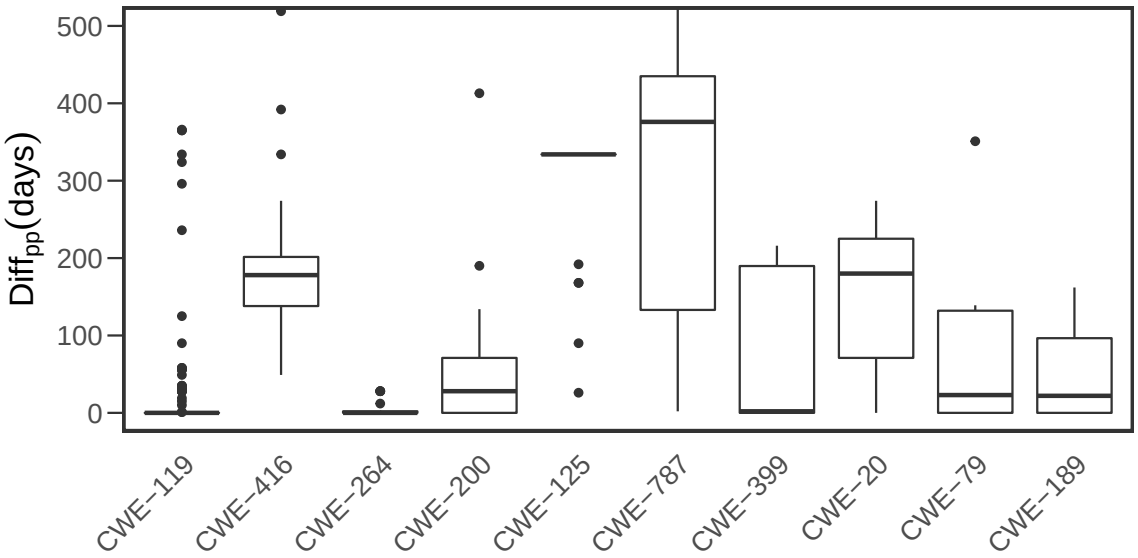


Figure 5.13: The comparison of the delay ($Diff_{pp}$) between when the first and last patch dates, base on the top 10 vulnerability weaknesses (CWEs) in the multi-affected vulnerabilities. Table 5.5 reflects the details of the top 10 CWEs.

Table 5.5: The top 10 common weaknesses (CWEs) in the multi-affected vulnerabilities.

#	CWE	Name	Count
1	CWE-119	Improper restriction of operations within the bounds of a memory buffer	142
2	CWE-416	Use after free	60
3	CWE-264	Permissions, privileges, and access controls	27
4	CWE-200	exposure of sensitive information to an unauthorized actor	25
5	CWE-125	Out-of-bounds read	25
6	CWE-787	Out-of-bounds write	21
7	CWE-399	Resource management errors	16
8	CWE-20	Improper input validation	14
9	CWE-79	Cross-site scripting	11
10	CWE-189	Numeric errors	11

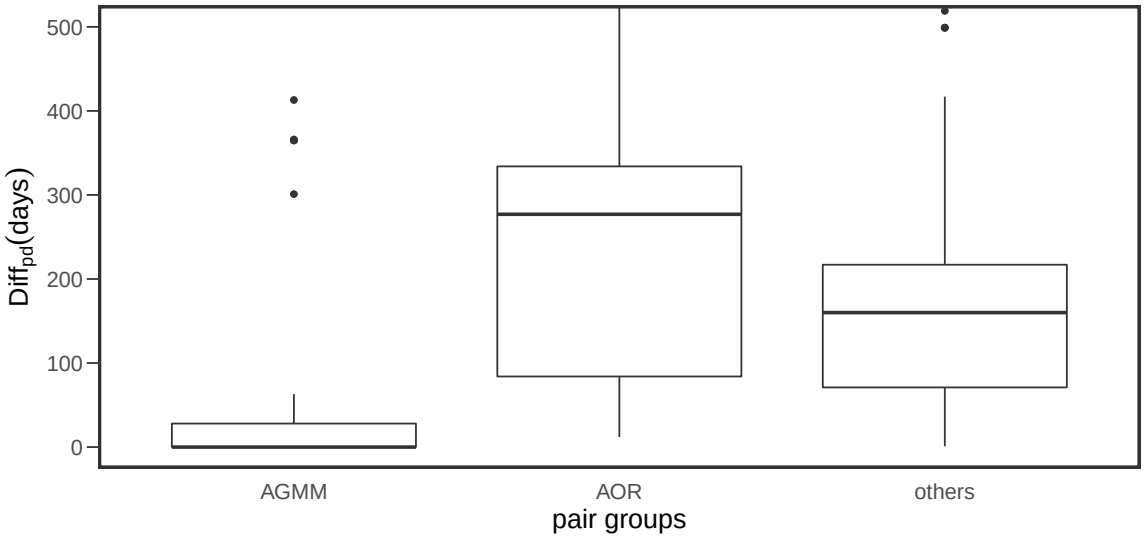


Figure 5.14: Comparison of the delay ($Diff_{pp}$) between the first and last patches. "AGMM" represents any pair of vendors from Adobe, Google, Microsoft, and Mozilla. "AOR" represents any pair of vendors from Apple, Oracle, and Red Hat.

$Diff_{pp}$ delay of multi-affected vulnerabilities is a median of 35 days across the CNAs. Table 5.5 shows the top 10 weaknesses in the multi-affected vulnerabilities, along with their name and count. Four (CWE-119, CWE-416, CWE-125, CWE-787) out of the top 10 CWEs are memory errors, four (CWE-264, CWE-200, CWE-20, CWE-189) are related to design flaws, two (CWE-399, CWE-79) are related to resource management errors.

Although some CWEs are related to a similar issue, the patch coherence of such related vulnerability types can still vary substantially. For example, vulnerabilities with CWE-119 (#1) "buffer overflow" and CWE-416 (#2) "use after free" are memory errors that both could be detected by static analysis tools and even be automatically fixed (Lee et al., 2018). However, the patch coherence for CWE-119 is a median of 0 days, while for CWE-416 is a median of 178 days (Wilcoxon test: p-value < 0.01, a large effect size of 1.32).

When any pair of CNAs from Adobe, Google, Microsoft, and Mozilla (i.e., the

AGMM group) are affected by a common vulnerability, the Diff$_{pp}$ delay (i.e., patch coherence) is a median of 0 days, while for any pairs of CNAs from Apple, Oracle, and Red Hat (i.e., the AOR group), the Diff$_{pp}$ delay is a median of 277 days. The difference between the two groups of CNAs is significant (Wilcoxon test: p-value $<$ 0.01) and the effect size is large (1.94). Figure 5.14 shows a large variance in the AOR group, compared to the variance in the AGMM group. This could be due to the security update policy of each CNA, which may vary by product. For example, in the AOR group, Oracle releases security updates quarterly,[15] while Apple[16] and Red Hat[17] have a more flexible schedule. By contrast, in the AGMM group, Google,[18] Microsoft[19] and Mozilla[20] release security updates either monthly or in (under) 30 days. As a result, the variance of the patch coherence in the AGMM group is relatively small. Another possible reason could be the development process and management in a CNA, e.g., low-priority for products in an old version.

[15] https://www.oracle.com/corporate/security-practices/assurance/vulnerability/
[16] https://support.apple.com/en-us/HT201220
[17] https://access.redhat.com/solutions/3711551
[18] https://chromium.googlesource.com/chromium/src/+/master/docs/security/severity-guidelines.md
[19] https://www.microsoft.com/en-us/msrc/faqs-security-update-guide
[20] https://support.mozilla.org/en-US/kb/managing-firefox-updates

Although the majority (i.e., 9) of the studied CNAs have a minority (<50%) of multi-affected vulnerabilities, these vulnerabilities are severe according to the CVSS scores, especially when they affect both types of products. The delay ($Diff_{pd}$) of patch releases for multi-affected vulnerabilities is longer than that of single-affected vulnerabilities. When the CNAs are affected by the common vulnerabilities, the patch coherence of vulnerabilities with weaknesses related to buffer overflow (CWE-119) and access controls (CWE-264) is a median of 0 days. Furthermore, the patch coherence in the AGMM group (i.e., Adobe, Google, Microsoft, and Mozilla) has a median of 0 days.

5.6 Discussion

In this section, we discuss the implications of our findings for practitioners (Section 5.6.1), researchers (Section 5.6.2) and CNAs (Section 5.6.3).

5.6.1 Practitioners

Practitioners should be aware of the potential issue of patch incoherence, i.e., a large value of $Diff_{pp}$, after the fix of a given vulnerability is available. Although the studied CNAs are leading security companies, the delay $Diff_{pp}$ of patch releases between them for a given vulnerability could vary from 0 days up to more than one year. The patch incoherence indicates the period that some of the vulnerable products affected by a vulnerability with an available fix remain unfixed. The delay $Diff_{pp}$ is impacted by product

types, vulnerability weaknesses, and the affected vendors. For example, CVE-2011-3071[21] allowed remote attackers to cause a denial of service in Chrome (Google) and Safari (Apple). Google made an announcement of security updates for CVE-2011-3071 along with 11 other vulnerabilities in a point release,[22] while Apple made an announcement for CVE-2011-3071 along with 120 other vulnerabilities in a major release.[23]

We compare patch coherence with two prior studies. Nappa et al. (2015) observed that the patch coherence of vulnerabilities in a shared code snippet between two applications is a median of 11 days from 10 popular client-side applications. The patch coherence in our work is longer (i.e., a median of 35 days) than the prior work since we study vulnerabilities that affected two CNAs' products with different code bases, which needs developers of each of the CNAs to develop their own fix. For dependent packages in the npm ecosystem, Decan et al. (2018b) showed that 50% of the dependent packages affected by vulnerable upstream projects took nearly 14 months to release fixes, while their upstream usually had a fix available within one month. The patch coherence of the dependent packages is longer than of the projects in our work. One possible reason of the poor patch coherence for dependent packages in npm could be due to the development process and management of these dependent projects, e.g., additional migration effort (e.g., code changes) to fix the vulnerable upstream projects (Kula et al., 2018a).

Practitioners should develop a better coordination model on vulnerability fix releases to reduce the size of the window of vulnerability exploits (i.e., obtain a negative value of Diff$_{pd}$) for delegated vulnerabilities. Each individual CNA releases fixes

[21] https://nvd.nist.gov/vuln/detail/CVE-2011-3071

[22] https://chromereleases.googleblog.com/2012/04/stable-and-beta-channel-updates.html

[23] https://support.apple.com/en-us/HT202561

for vulnerabilities with a delay Diff_{pd} w.r.t. the disclosure date (Tables 5.2 to 5.4). The Diff_{pd} delay for delegated vulnerabilities is longer than for self-assigned vulnerabilities, even though the risk level for delegated vulnerabilities is higher than for self-assigned ones. Since delegated vulnerabilities indicate that at least one CNA develops a fix and another CNA controls the disclosure process, i.e., at least two CNAs are involved, a larger value of Diff_{pd} (>0) could be an indicator of low-quality coordination between these involved CNAs. Since this of course depends on other factors, like the difficulty of fixing a given vulnerability, future work should study the quality of coordination in more detail.

In order to put our findings in perspective, we compare the Diff_{pd} delay with prior studies and summarize the result in Table 5.6. We observe that the trend of the Diff_{pd} for vulnerabilities in our work is similar to vulnerabilities from 1998 to 2011 studied by Shahzad et al. (2012). While several other works (Li and Paxson, 2017; Piantadosi et al., 2019; Decan et al., 2018b; Alfadel et al., 2021) studied vulnerabilities in open-source projects, our study focuses on corporations (CNAs). Yet, the gap of the Diff_{pd} delay between open-source projects and corporations is not significantly large, as these studies found that the value of the (Diff_{pd}) delay in open-source projects usually is negative (before disclosure), except in the Python ecosystem (Alfadel et al., 2021).

5.6.2 Researchers

Researchers could explore more dimensions (e.g., developer team size) on understanding the delay of patch releases w.r.t the disclosure date in order to reduce such delay, especially for delegated vulnerabilities. We investigate several dimensions (e.g., product types, assignor) and observe the similarity and difference across the CNAs. For

Table 5.6: Comparison of the Diff_{pd} delay with prior work.

	Data[1]	**Subjects**	**Types**	**Diff**[2]$_{pd}$
Our work	2010 - 2019	13 CNAs	companies	median of -2 day
Shahzad et al. (2012)	1998 - 2011	8 vendors	both[3]	< 0 days[4]
Li and Paxson (2017)	Till 2016	682 projects	open-source	< -7 days
Piantadosi et al. (2019)	337 vulns	Apache Http and Tomcat	open-source	median -12 and -54 days
Decan et al. (2018b)	2012 - 2017	npm	open-source	< 0 days[5]
Alfadel et al. (2021)	550 vulns	Python	open-source	> 0 days[6]

[1] Note that we present either the data period of studied vulnerabilities or the number of studied vulnerabilities, when applicable.
[2] Note that we present an approximate value with the percentage in the footnote when the prior work did not state a precise value.
[3] 2 open-source vendors and 6 companies.
[3] 73% before disclosure.
[4] 84% before disclosure.
[5] 51% after disclosure.

example, the CNAs often release fixes for self-assigned vulnerabilities within 30 days before the disclosure date, while they have an inconsistent pace for delegated vulnerabilities, from -30 days up to ⩾ 90 days (RQ2). Given the increasing number of vulnerabilities and software supply chain attacks, and the fact that a vulnerability disclosed without any available fix indicates the highest risk, a shorter delay of patch releases w.r.t. the disclosure date reduces the chance of vulnerability exploits at a large scale.

In addition, **more research is required for vulnerabilities in the operating-system and hardware types of product.** The delay Diff_{pd} of releasing fixes for delegated vulnerabilities in the operating-system type of product is longer than that of the application type of product, by a median of 4 days (RQ2). However, for the application type of product, the median is minus 1 day (fix available before disclosure), compared to plus 3 days (after disclosure) for the operating-system type of product. Furthermore,

the lower the severity level of a vulnerability in the operating-system type of product, the longer the delay ($Diff_{pd}$) of releasing fixes is. While a large number of prior studies has been conducted on application vulnerabilities (Jovanovic et al., 2006; Gupta and Gupta, 2017; Goyal et al., 2011; Rafique et al., 2015), vulnerabilities in operating systems (e.g., Windows, Debian) are explored less commonly, even though operating-system vulnerabilities impact an order of magnitude more users than application-type ones. Furthermore, software providers often do not provide updates for all affected operating system versions, due to end-of-life (EOL) (Farhang et al., 2019).

Finally, a large number (5,339) of multi-affected vulnerabilities affect both operating-system and application types of products. These vulnerabilities are extremely severe, with a median base score of 9.3, median exploitability score of 8.6 and a median impact score of 10. As such, the delay of releasing fixes in a median of 4 days after the disclosure date leads to a large impact on a victim system. More research should dive into detection tools for such a kind of vulnerability or techniques to secure operating systems.

5.6.3 CNAs

The existing CNAs should consider designing a different disclosure process for delegated vulnerabilities, compared to self-assigned vulnerabilities. First, affected software vendors have a potential delay of becoming aware of delegated vulnerabilities, compared to self-assigned vulnerabilities. Second, although delegated vulnerabilities account for a minority (<50%) of vulnerabilities in each of the studied CNAs except Google (Figure 5.5), they are more severe than self-assigned vulnerabilities. Delegated

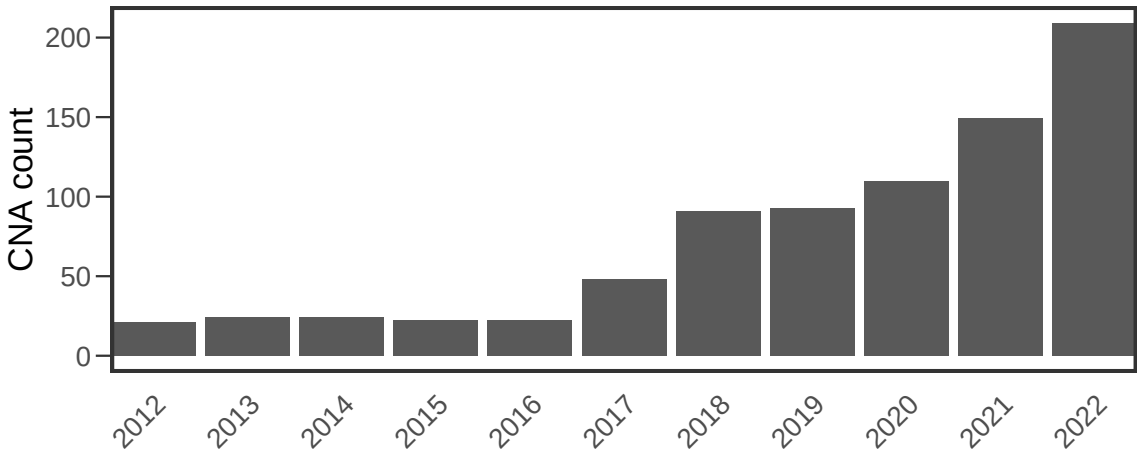


Figure 5.15: The number of CNAs in the CVE program.

vulnerabilities in the application and operating-system types of products have a median CVSS score of 7.5 and 7.1, respectively, i.e., both are high-severity. Moreover, the results in RQ2 indicate that the delay Diff_{pd} of releasing fixes for delegated vulnerabilities in the operating-system type of product is a median of 3 days up to more than 90 days after the disclosure date. This gives attackers a large window to exploit delegated vulnerabilities because the fixes are not yet available, even though the public is aware of these vulnerabilities. Furthermore, given the increasing number of software supply chains and the number of software vendors in a software supply chain, such a large window implies a potential large scale of losses.

Due to the increasing number of vulnerabilities in recent years and a large number (>15k) of vulnerabilities in recent years,[24] many new CNAs have joined the CVE program to share efforts of assigning and disclosing vulnerabilities. Figure 5.15 indicates that the number of CNAs doubled twice, from 2017 to 2018 and from 2020 to 2022. Since the CVE program does not specify general guidelines of vulnerability disclosure

[24] https://nvd.nist.gov/general/visualizations/vulnerability-visualizations/cvss-severity-distribution-over-time

for CNAs, each CNA discloses vulnerabilities based on its own background knowledge and policy. As such, given the high-risk of delegated vulnerabilities, a new CNA may disclose a vulnerability earlier/later than it should be. In addition, even the delay of patch releases for delegated vulnerabilities is a median of 1 day before disclosure, there exists a high chance of vulnerability exploitation due to the inconsistent pace of patch releases across the affected vendors, i.e., patch incoherence.

The concept of coordinated vulnerability disclosure (CVD) should be widely broadcast to ensure the security of affected products. The software security-related communities, including researchers, companies and governments, have been devoted to deriving best practices for multi-party coordination of fixing vulnerabilities, given today's era of widespread open-source reuse. However, it takes time to develop practices across software projects, companies, organizations and countries, and to start applying them. For example, the European Union Agency for Cybersecurity (ENISA) reported that the notion of CVD remains fragmented in the 2023 report[25], even though the European Cyber Resilience Act was published in September, 2022.[26] Our results identify traces of such fragmentation as there exist Diff_{pd} inconsistency and a large delay of patch releases across CNAs.

Our results could be taken into account when developing CVD best practices. For example, Table 5.4 indicates that most of the studied CNAs release the fixes for the largest (bold) number of multi-affected vulnerabilities (both self-assigned and delegated) 30 days before the disclosure (-30∼0 days). Therefore, for a given vulnerability, an affected vendor should be able to release the fix within 30 days after the disclosure of the vulnerability (0∼30 days). There might be some exceptional policies, such as,

[25] https://www.enisa.europa.eu/news/coordinated-vulnerability-disclosure-towards-a-common-
[26] https://digital-strategy.ec.europa.eu/en/library/cyber-resilience-act

for a given vulnerability, a vendor having more than 5 variants of the affected products should release the fixes for all affected variants within 90 days after the disclosure (see Figure 5.12).

5.7 Threats to Validity

5.7.1 Internal Validity

We calculate the delays Diff_{pd} and Diff_{pp} based on the collected security advisories from tenable.com because the open-source vulnerability database (OSVDB) was shutdown permanently in 2016.[27] Although tenable.com might miss some security advisories, our data set includes 82% of the 39,409 collected vulnerabilities (cf. Section 5.4.2) that affected application and operating-system types of products (i.e., software-related products) to provide insightful information of current practices that are used by top companies. We encourage future studies to explore other data sources to generalize our results.

We only study the application and operating-system types of products according to the CPE configurations in RQ2 and RQ3, due to the lack of security advisories related to hardware type of vulnerabilities in tenable.com. However, vulnerabilities in the hardware type of product are also severe in the context of software supply chains. For example, attackers gained complete control via the Starbleed (CVE-2019-14626)[28] vulnerability in FPGA (Field Programmable Gate Arrays) chips, which can be found in many safety-critical applications.

Another internal threat to validity is that our study focuses on vulnerabilities with a

[27] https://www.securityweek.com/osvdb-shut-down-permanently
[28] https://nvd.nist.gov/vuln/detail/CVE-2019-14626

valid CVE ID and excluded those (9,836) with an invalid CVE ID.[29] These CVE IDs with an invalid status may still correspond to vulnerabilities to a certain extent, but were invalid due to a variety of reasons. For example, one CVE should be split into several CVEs to reflect the risks of these CVEs, or the definition of a CVE is controversial from different CNAs, ending up with a dispute status.

5.7.2 External Validity

Because we collected data from the NVD and `tenable.com` databases, our results may not generalize to other CNAs. However, our results reveal the current practices of the studied CNAs, who are experts in security and also software vendors on vulnerability coordination and fix releases. These practices shed light into the deeper understanding of how fast the CNAs react to vulnerabilities. Therefore, we acknowledge that future studies are required to add more data source to reach more general conclusion.

5.8 Conclusion

In this work, we investigate the efficiency of vulnerability coordination and fix releases w.r.t the disclosure date from a set of 39,409 vulnerabilities and the top 13 CNAs that have played several roles in multi-party coordination on fixing vulnerabilities. Using the dataset, we analyze influential factors of vulnerability fix releases and disclosure, including the assignor of a vulnerability, affected product types and the number of affected vendors. In addition, we extensively measure the efficiency of patch releases and patch coherence in the multi-party coordination.

[29] `https://cve.mitre.org/cve/list_rules_and_guidance/correcting_counting_issues.html`

Our findings have revealed that the size of the window of vulnerability exploitation in the CNAs is usually small (with fixes released before the disclosure), but large for delegated vulnerabilities in operating-system products and multi-affected vulnerabilities affecting both operating-system and application products (with fixes released after disclosure). Additionally, the patch coherence for vulnerabilities varies from a median of 35 days up to more than one year, due to weaknesses and pairs of the affected CNAs.

These insights indicate that the window size of potential vulnerability exploitation for a given vulnerability in a vulnerable product is not significantly large. However, given the poor patch coherence and the context of software supply chains, both software vendors and users are still at high-risk for a long period. CNAs could design better disclosure policies for various kinds of vulnerabilities and a better fix release process. Researchers could investigate additional practices to improve such policies and processes, such as ways to patch vulnerable products that are affected by vulnerabilities that already have an available fix.

CHAPTER 6

Coordination of co-evolving software projects in the WordPress plugin-based ecosystem

One can extend the features of a software system by installing a set of additional components called plugins. WordPress, as a typical example of such plugin-based software ecosystems, is used by millions of websites and has a large number (i.e., 54,777) of available plugins. These plugin-based software ecosystems are different from traditional ecosystems (e.g., NPM dependencies) in a sense that there is a high coupling between a platform and its plugins, compared to traditional ecosystems for which components might not necessarily depend on each other (e.g., NPM libraries do not depend on a specific version of NPM or a specific version of a client software system). The high coupling between a plugin and its platform and other plugins causes incompatibility issues that occur during the co-evolution of a plugin and its platform as well as other plugins. In fact, incompatibility issues represent a major challenge when upgrading WordPress or its plugins. According to our study of the top 500 most-released WordPress plugins, we observe that incompatibility issues represent the 3^{rd} major cause for bad releases, which are rapidly (within the next 24 hours) fixed via urgent releases. 32% of these incompatibilities are between a plugin and WordPress while 19% are between peer plugins. In this chapter, we study how plugins co-evolve with the underlying platform as well as other plugins, in an effort to understand the practices that are related support such co-evolution and reduce incompatibility issues. In particular, we investigate how plugins support the latest available

versions of WordPress, as well as how plugins are related to each other, and how they co-evolve. We observe that a plugin's support of new versions of WordPress with a large amount of code change is risky, as the releases which declare such support have a higher chance to be followed by an urgent release compared to ordinary releases. Although plugins support the latest WordPress version, plugin developers omit important changes such as deleting the use of removed WordPress APIs, which are removed a median of 873 days after the APIs have been removed from the source code of WordPress. Plugins introduce new releases that are made according to a median of 5 other plugins, which we refer to as peer-triggered releases. A median of 20% of the peer-triggered releases are urgent releases that fix problems in their previous releases. The most common goal of peer-triggered releases is the fixing of incompatibility issues that a plugin detects as late as after a median of 36 days since the last release of another plugin. Our work sheds light into the co-evolution of WordPress plugins with their platform as well as peer plugins in an effort to uncover the practices of plugin evolution, so WordPress can accordingly design approaches to avoid incompatibility issues.

An earlier version of this chapter is published in the ACM Transactions on Software Engineering and Methodology Journal (TOSEM) (Lin et al., 2022a).

6.1 Introduction

A plugin-based software ecosystem is an ecosystem that is constituted of a platform that can be extended with a set of "plugins". WordPress, as a typical example of a plugin-based ecosystem, has more than 54,777 plugins that are available to end-users. For example, one can install the *FV Flowplayer Video Player* plugin to display videos on his WordPress website. The richness of these plugins makes WordPress one of the most popular content management platforms. In fact, WordPress is used by 61.8% of all the websites, whose content management system is known.[1] WordPress is also used to build well known websites such as the New York Post, USA Today, and TIME.com websites.[2]

Plugin-based software ecosystems are different from traditional ecosystems. For instance, the relation between the plugins and the WordPress platform is symbiotic. The features that are provided by the platform are often limited and need to be extended through the plugins. The plugins, on the other side, cannot be used without the platform. In addition, plugins interfere with each other through their shared platform. That is contrary to traditional ecosystems, for which a component can often evolve independently from the other components. In addition, components in traditional ecosystems might never interfere with each other if they do not share any resources or do not invoke each other. Thus, there exists a high coupling between plugins and the platform as well as the different plugins for plugin-based software ecosystems.

[1] https://w3techs.com/technologies/overview/content_management
[2] https://wpvip.com/case-studies/

Such a high coupling leads to incompatibility issues that occur when a plugin co-evolves with the platform or with other plugins. For example, a plugin might introduce a release that is incompatible with the platform or with other plugins. Given the richness of plugins, the freedom to intermix plugins, and their varying release cycles, the users of WordPress often must cope with incompatibility issues, which discourage users from upgrading their WordPress and plugins.[3] For example, a user's website crashed after upgrading to the latest WordPress version due to an incompatibility between a plugin and that latest WordPress version.[4] In other words, the plugin did not co-evolve correctly with its platform.

A large amount of research has examined incompatibility issues in traditional ecosystems, such as project dependencies (Bavota et al., 2013; Lungu et al., 2010; Espinha et al., 2014; Xavier et al., 2017) (i.e., project dependency refers to a relation between two working applications that contain a set of source files, configurations, assets, etc.), out-of-date dependencies (Mirhosseini and Parnin, 2017; Robbes et al., 2012), library dependencies (Raemaekers et al., 2012; Kula et al., 2018b), and package dependencies (Decan et al., 2017b; Kikas et al., 2017) (i.e., package dependency refers to a package consisting of a group of related classes and source files that require another package in order to work). However, a few research efforts (Sayagh et al., 2017; Eshkevari et al., 2014) investigated the incompatibility issues in plugin-based software ecosystems, which have higher chances for incompatibility issues compared to traditional ecosystems, as we noted earlier.

Through a preliminary study of the top 500 WordPress plugins with the largest number of releases, we observe that incompatibility issues represent the 3rd major cause

[3] https://wordpress.org/news/2009/10/plugin-compatibility-beta/
[4] https://wordpress.org/support/topic/urgent-plugin-update-for-4-7-1/

of bad releases, which are rapidly addressed via urgent releases. Note that this type of releases is also studied by prior work (Hassan et al., 2017; Lin et al., 2017), which also defined the urgent releases as the releases whose goal is to fix errors that were introduced in a previous release that was recently published (within 24 hours before the urgent release). 32% of these incompatibility issues occur between plugins and the WordPress platform while 19% are between peer plugins.

The goal of this chapter is to shed light into the co-evolution of plugins with their platform and other plugins in an effort to understand the practices of such co-evolution, so WordPress can better design approaches to avoid incompatibility issues. We focus our study on the WordPress ecosystem since it is one of the most successful and widely used plugin-based ecosystems. In particular, we address the following research questions:

RQ1: When plugins support a new version of WordPress?

> Plugins announce their supports for a newly released version of WordPress after a median of 38 days, with a lag of one release behind the latest available version of WordPress. However, each plugin does not have a consistent lag for every WordPress version. Plugin developers use a new WordPress API as fast as 1 day after its introduction and delete the use of removed APIs after a median of 873 days.

RQ2: How plugins support a new version of WordPress?

> 89% of the studied plugins have at least one release that does not support the latest version of WordPress. These releases have a higher chance of being followed by an urgent release (153%) compared to ordinary releases. Furthermore, plugin releases that announce support to a new version of WordPress with a large amount of code changes are more likely to be followed by an urgent release.

RQ3: How WordPress plugins co-evolve?

> Plugins ship new releases that are triggered due to a median of 5 other plugins (i.e., peer-triggered releases). 60% of the pair of plugins that have a peer-triggered release do not explicitly depend on each other. Peer-triggered releases have a lower chance of being followed by an urgent release (66%) and a median of 20% of the peer-triggered releases are urgent releases. Fixing incompatibility issues is the most common goal (46%) for peer-triggered releases.

Our results provide an understanding on how a plugin co-evolves with its platform and other peer plugins. Such an understanding provides WordPress developers with better understanding of the co-evolution in plugin-based software ecosystems in order to avoid incompatibility issues.

The chapter is organized as follows. Section 6.2 provides background information about the release process of WordPress plugins and describes the related work to our study. Section 6.3 provides our preliminary analysis on the incompatibilities that are fixed through urgent releases. Section 6.4 answers our research questions. Section 6.5 discusses threats to the validity of our observations. Finally, Section 6.6 concludes the work.

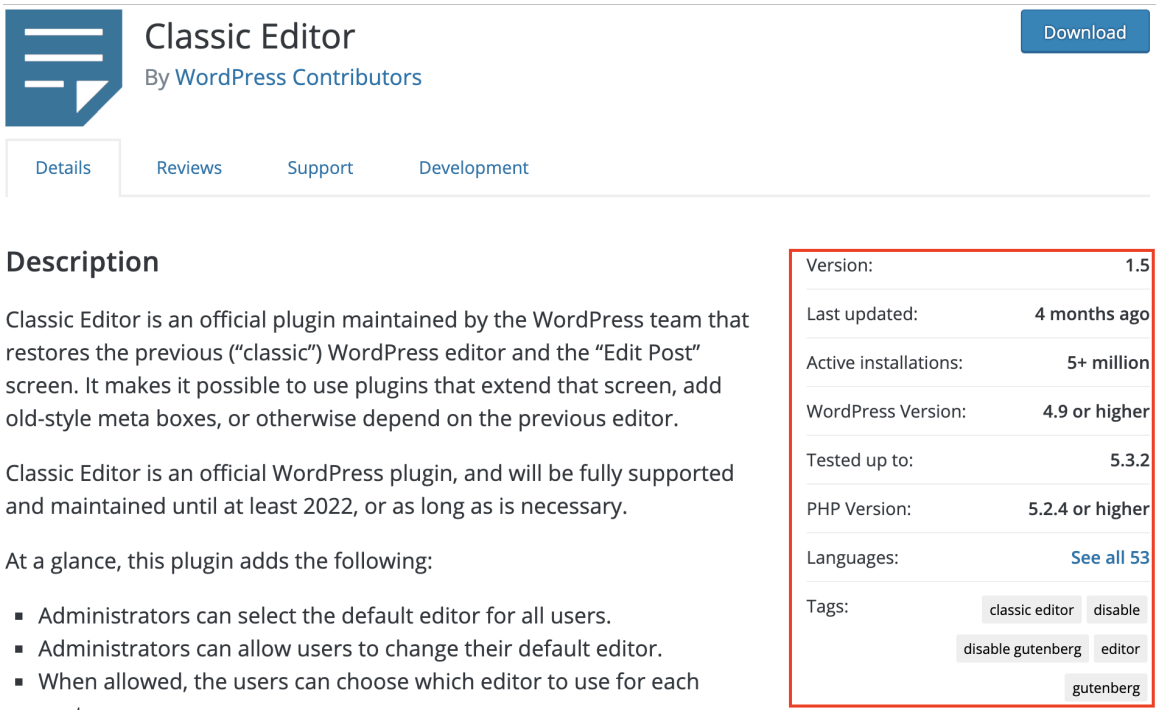
6.2 Background and Related Work

In this section, we provide background information about how plugins ship new releases and support new versions of WordPress (Section 6.2.1). We discuss related work on empirical studies of the co-evolution of different components of an ecosystem (Section 6.2.2) and incompatibility issues (Section 6.2.3).

6.2.1 How plugins support new WordPress versions

```
Line
1 === Classic Editor ===
2 Contributors: wordpressdotorg, azaozz, melchoyce, chanthaboune, alexislloyd, pento, youknowriad, desrosj, luciano-croce
3 Tags: gutenberg, disable, disable gutenberg, editor, classic editor, block editor
4 Requires at least: 4.9
5 Tested up to: 5.3
6 Stable tag: 1.5
7 Requires PHP: 5.2.4
8 License: GPLv2 or later
9 License URI: http://www.gnu.org/licenses/gpl-2.0.html
```

(a) An example of a readme file



(b) A plugin web page example. The metadata information in the red box are generated from the readme file. The information indicates the versions of the plugin, the supported WordPress version, and programming languages. Note that the *Tested-up-to* tag only defines the minor version (e.g.,5.3) tested by plugins and WordPress will automatically add the latest micro version (e.g., 5.3.2) to it.

Figure 6.1: An example of a readme file and the web page displaying the associated information in the readme file.

Each plugin has a "readme" file that contains the meta-data of the plugin, such as

the latest available version and the versions of WordPress that the plugin officially supports. The *readme* file of a plugin is used by the WordPress platform to auto-generate a web page for that plugin. For example, the latest available version of the *"Classic Editor"* plugin is *1.5*, as indicated by the "*Stable-tag*" in the *readme* file, and as displayed on the web page of the plugin, as shown in Figure 6.1. The "Tested up to" tag indicates that the plugin is supporting WordPress versions up to *5.3.2*. The *readme* file changes are recorded in the source code repository of each plugin.

6.2.2 Co-evolution of different components in ecosystems

Prior work studied the co-evolution of a project and its dependencies (Hora et al., 2018; McDonnell et al., 2013; Kula et al., 2018b; Bavota et al., 2015). Hora et al. (2018) observed that developers discover and react to API changes after a median of 34 days and 95% of the software systems never adopt new API changes. Kula et al. (2018b) identified that 81.5% of the software systems stick with outdated dependencies, due to the overhead of updating such dependencies. Bavota et al. (2015) observed that developers update to the new version of a dependency when that new version includes major changes such as new features or a large amount of bug fixes. The co-evolution of different components of an ecosystem leads to incompatibility issues, which impacts the stability and fault-proneness of an ecosystem (Bavota et al., 2014), and results in poor user experiences (Wei et al., 2016).

Another line of research is related to breaking changes (Raemaekers et al., 2014; Bogart et al., 2016; Brito et al., 2020). Raemaekers et al. (2014) studied subsequent releases of more than 20,000 libraries in the Maven repository and observed that one third of all releases introduced at least one breaking change (e.g., method removals)

that affects their client systems. Raemaekers et al. (2014) also observed that developers tend to keep up with the latest version of dependent libraries with a median of 0 release lag of the versions between the library version included in a project and the latest available version of the library. Bogart et al. (2016) observed the npm ecosystem focuses strongly on signaling change through semantic versioning for developers to manage their dependencies, while developers bear to update their dependencies when breaking changes are made in the dependencies in the R ecosystem. Even breaking changes are costly in terms of interruptions and rework, each software ecosystem makes such braking changes for various purposes. For example Bogart et al. (2016) reported that the most important reason for breaking changes is technical debt, rather than bugs or new features in the Eclipse, R and npm ecosystems. Brito et al. (2020) reported that another important reason for breaking changes is to implement new features to simplify source code and improve maintainability in Java libraries of popular GitHub projects.

6.2.3 WordPress incompatibilities

A few prior studies focused on the incompatibility issues in the WordPress ecosystem (Sayagh et al., 2017; Eshkevari et al., 2014; Sayagh and Adams, 2015; Nguyen et al., 2014). Sayagh et al. (Sayagh and Adams, 2015; Sayagh et al., 2017) observed that plugins share a large percentage of the platform configuration options. For example, 79% of all WordPress configurable constants and 85.16% of all WordPress database options are used by at least two different plugins, which suggests potential conflicts between plugins. Nguyen et al. (2014) leveraged a source code analysis technique to identify incompatibility errors between different WordPress plugins. Eshkevari et al. (2014) levareged static and dynamic source code analysis techniques to identify incompatibility issues

between two plugins. While these studies focused on how to detect and fix incompatibility issues, our study focuses on investigating the co-evolution of the plugin-platform and peer-plugins, which can guide WordPress developers to develop mechanisms that support such co-evolution to prevent incompatibility issues.

6.3 (Preliminary Question) What is the proportion of urgent releases that are due to incompatibility issues?

Motivation: The goal of this preliminary research question is to quantify the prevalence of incompatibility issues. These issues occur due to missed interactions between a plugin and the platform or other peer plugins. In this preliminary research question, we focus on the interactions that were missed and that are critical enough to lead to urgent releases. These releases are shipped within 24 hours from their respective prior release. Urgent releases are perceived to be harmful as noted by several prior studies (e.g., (Hassan et al., 2018; Lin et al., 2017)).

Approach: To study how often incompatibility issues lead to urgent releases, we first need to investigate the prevalence of urgent releases and whether the urgent releases are related to their prior releases. Then, we investigate the main causes of the urgent releases in order to quantify how often they occur due to incompatibility issues.

To do so, we investigate the releases of the top 500 most-released plugins (the **Plugin Dataset**). To obtain these plugins, we collect all the 60,592 plugins that have a *readme* file and the *"Stable-tag"* and sort these plugins based on the number of their releases, which we obtain from the historical changes to the *"Stable-tag"* in a plugin's *readme* file.

To address our preliminary research question, we first quantify the number of urgent releases for each of the top 500 most-released WordPress plugins. We then conduct a qualitative analysis on a representative random sample (confidence level = 95%, confidence interval = 5%) of 375 urgent releases. The goal of this qualitative analysis is to first identify whether urgent releases are related to their prior release. We consider that an urgent release is related to its prior release (1) when both of the release notes of the urgent and prior release mention the same information (e.g., features), or (2) when both releases change the same parts of the code (e.g., functions or classes). However, we cannot conclude if an urgent release is related to its prior release when both releases change different parts of the code. The second goal of the qualitative analysis is to quantify how often the urgent releases are caused by incompatibility issues. To do so, we use the extended Swanson's maintenance classification (Hindle et al., 2008; Swanson, 1976) to label the causes of urgent releases. We perform an iterative process that is similar to prior work (Seaman, 1999; Seaman et al., 2008) to figure out what causes an urgent release. In particular, we manually analyze the release notes and commit messages of the 375 urgent releases against their respective prior release. For example, one stated *"fixing default sort"* in the release note of version 2.12.6 of the *12-step-meeting-list*[5] plugin, which was an urgent release. The sort feature was introduced in version 2.12.5 as stated *"now able to set sort order on meetings page"* in its release note. Meanwhile, there were overlaps between the code changes made in version 2.12.6 and 2.12.6. Therefore, we determine that the urgent release 2.12.6 was to fix bugs in version 2.12.5 and label the root cause of the urgent release as a malfunction. Note that our dataset contains 16,444 urgent releases for the top 500 most-released plugins.

[5] https://wordpress.org/plugins/12-step-meeting-list/

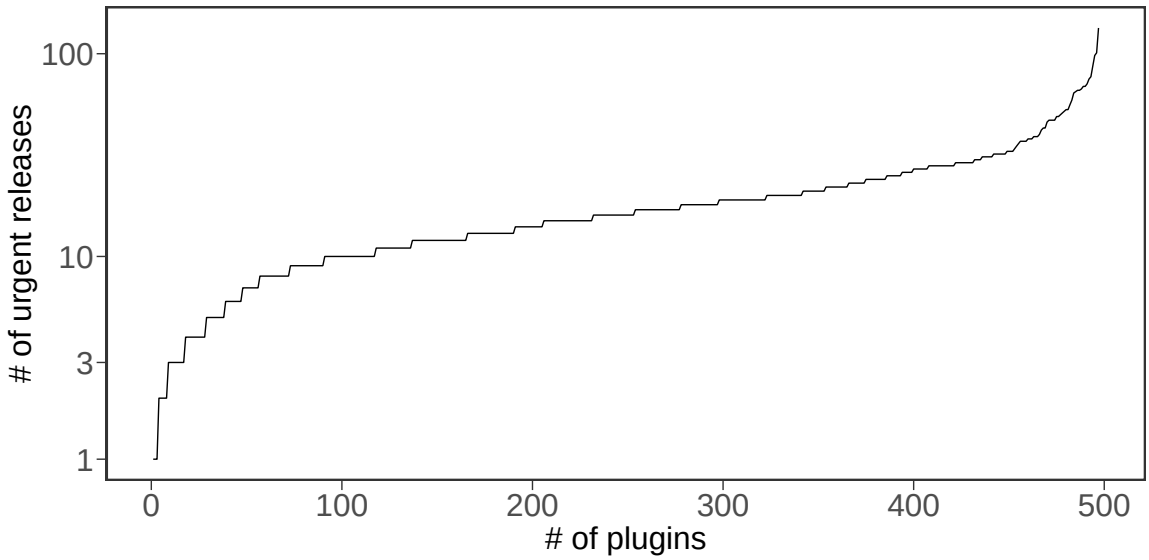


Figure 6.2: The distribution of plugins with their respective number of urgent releases. 497 out of the studied plugins have at least one urgent release, while three do not have any urgent release.

Results: 497 (99%) of the studied plugins have at least one urgent release (i.e., a release within 24 hours of its prior release). The studied plugins have a median of 24 urgent releases, as shown in Figure 6.2. 73 (14%) of the studied plugins have less than 10 urgent releases while 62 (12%) of the studied plugins have more than 30 urgent releases. We also observe that some releases are followed not just by one urgent release, but by a median of 4.5 and up to 18 urgent releases. For example, the *Wp-Easycart* plugin[6] continuously released 14 urgent releases (i.e., from 1.0.8 to 1.0.19) from July 22 to July 24, 2013. The causes of these urgent releases include bug fixes in the plugin, bugs related to WordPress (e.g., notices appearing everywhere when WordPress was in debug mode), and bugs related to other software systems (e.g., deprecated functions in PHP 5.3, did not store data in MySQL). The *Hitsteps-Visitor-Manager* plugin[7] released 5

[6] https://wordpress.org/plugins/wp-easycart/
[7] https://wordpress.org/plugins/hitsteps-visitor-manager/

Table 6.1: Causes for urgent releases based on extended Swanson categories of changes. Note that one release can address issues across several categories. Incompatibility is the 3^{rd} most common cause for urgent releases.

Category	Issues Addressed	Description	Count
Corrective	Visual	Modifying changes related to user interface. For example, ensuring that RichText links follow the global colors.	130
	Data Processing Logic	Changing the data processing logic in the operational flows. For example, fix the logic of email piping in the multi-site and single-site detection.	88
	Malfunction	Functional errors or processing failures	34
Adaptive	Incompatibility	Fixing conflicts with the platform (e.g., WordPress, browsers), other plugins, libraries, or services (e.g. API).	**69**
Perfective	Performance	Enhancing functionality by improving throughput. For example, reduce the amount of data that is loaded at the initial stage when a page is opened.	9
	Security	Fixing security vulnerabilities or avoiding the leakage of personal sensitive data. For example, a security fix to prevent malicious uploads.	8
Implementation	Improvement	Adding new features or premium service.	30
	Localization	Changes related to languages, regions, or translations. For example, updating the language files.	20
Non-functional	Documentation	Adding or revising meta data, guidelines, figures, or relevant information.	36
	Code Cleanup	Cleaning or refactoring the source code.	11
	Other	-	9

urgent releases to resolve incompatibility issues with *Woocommerce* analytics and the *Contact-Form-7* plugin[8] between June 10 and 15 in 2015.

84% of the urgent releases in our qualitatively examined urgent releases are fixing bugs in their prior release, where the 3^{rd} most common cause of these bugs (32% of the qualitatively examined urgent releases) are related to incompatibility issues. The majority of urgent releases are fixing issues that affect the operation of a plugin

[8] https://wordpress.org/plugins/contact-form-7/

and result in failures. For example, an urgent release provides a set of new settings that were missing in prior release, which displayed multiple errors on users' websites. In addition, 9% of the urgent releases are complementing features in their prior releases rather than fixing their prior releases. For example, the *Easyindex* plugin[9] stated that "*Update: Made the instructions for thumbnail preselection clearer*" to guide users on how to use the new features. Note that we are not able to classify 7% of the urgent releases, which do not have any associated release notes or precise commit messages or due to the urgent releases exploiting different classes or functions than the prior release. The 3^{rd} most common issue that plugin developers fix via their urgent releases is related to incompatibility issues, as shown in Table 6.1. For example, the *Bns-login* plugin[10] released the urgent release 1.5.1 to fix incompatibility issues with the WordPress version 3.0.1. 32% of these incompatibility issues are between the plugins and WordPress, 19% are between different plugins, 32% are between the plugins and other software systems (e.g., database, PHP), and 26% are between the plugins and third party (e.g., JQuery libraries, Google APIs).

99% of the studied plugins release at least one urgent release. The 3^{rd} most common cause for these urgent releases are done to address incompatibility issues. Among these issues, 32% of the incompatibility issues are between plugins and WordPress and 19% are between peer-plugins.

[9] https://wordpress.org/plugins/easyindex
[10] https://wordpress.org/plugins/bns-login

6.4 Understanding the Relation between WordPress and Plugin Releases

Motivated by the observation of our preliminary study, we study when (RQ1) and how (RQ2) the top 500 most-released plugins co-evolve with WordPress and how the plugins co-evolve related to each other (RQ3).

RQ1: When plugins support a new version of WordPress?

Motivation: The goal of this research question is to investigate when plugins support new versions of WordPress, and the impact of the timing of such a support on the quality of the plugin support releases (i.e., releases on which plugins support new WordPress versions). We measure such a quality by the occurrence of urgent releases, as such releases indicate that a bug is introduced in their prior releases, as discussed by prior work and our preliminary study. Our research question sheds light on the dynamics of the relation between WordPress and its plugins.

Approach: To investigate when plugins support a new version of WordPress, we measure the duration of the *"off-sync period"*, which consists of the lag between the release of a new WordPress version and when a plugin releases a version (*"support release"*) that supports that new WordPress version, as shown in Figure 6.3. As illustrated in the same Figure, we refer to the plugin releases that are shipped within the off-sync period as "*off-sync releases*" and the other plugin releases as "*in-sync releases*". We also investigate whether plugins are consistent with their support of new WordPress versions. For example, a plugin always releases a support release within 7 days after a new version of WordPress. Finally, we investigate when plugins make changes according to the

changes of WordPress APIs.

In this research question, we study the plugins' releases as well as WordPress releases. In particular, we use the **Plugin Dataset** (the set of the top 500 most-released plugins) and their associated releases, which are obtained following the approach discussed in our preliminary research question. In addition, we collect all the versions of WordPress including their release dates and notes from the WordPress release page[11] (the **WordPress Dataset**). WordPress had five major versions with each of them having 10 minor versions (e.g., 4.0 to 4.9 for the WordPress version 4) except for version 5, which is the latest major version of WordPress and which has only three minor versions at the time of our data collection. We exclude the WordPress version 1 since it has only 4 active plugins. Note that we do not consider the testing and beta versions of WordPress, such as *"beta 1"* or *"release candidate 1"*. These versions are not official and are just for testing purposes[12].

Since the duration of the off-sync periods varies from one plugin and WordPress releases to another, we classify the support releases for a new version of WordPress into 4 groups:

1. **Early adopters** are plugin releases that support WordPress Beta versions.

2. **Fast adopters** are plugin releases that support new WordPress versions within one week after the official release date of a new WordPress version.

3. **Normal adopters** are plugin releases that support a new WordPress version after one week and within 60 days from the release date of the new WordPress version.

[11] https://wordpress.org/news/category/releases/
[12] https://wordpress.org/news/2019/10/wordpress-5-3-release-candidate/

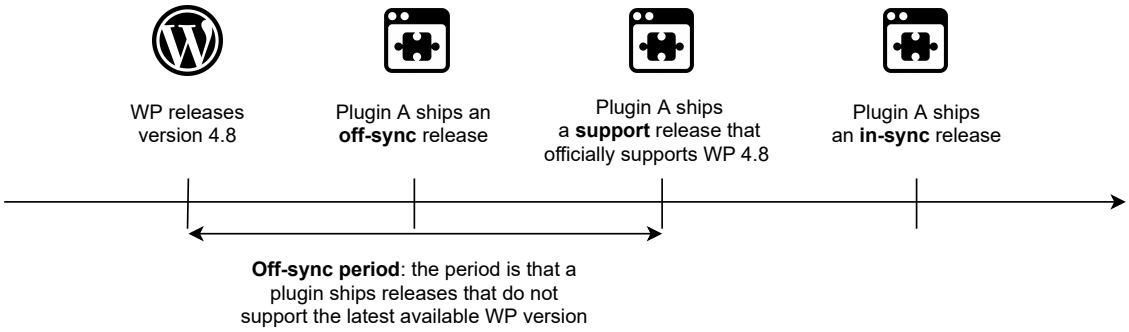


Figure 6.3: An illustration of the **"off-sync period"**, which consists of the time between the release of a new version of WordPress and when a plugin supports that new version. We refer to the release that supports a new WordPress version by **"support release"**. During the off-sync period, a plugin can release an **"off-sync release"**, which is shipped by a plugin before it declares support to the latest available version of WordPress. We refer to any release that is shipped after the support of the latest WordPress version by **"in-sync"** release.

4. **Slow adopters** are plugin releases that support a new WordPress version after at least 60 days.

We then compare which of these groups has a higher chance to be followed by an urgent release.

Finally, we investigate when plugins adopt a new or delete a removed WordPress API. To do so, we investigate which type of plugin releases (i.e., in-sync, off-sync, or support releases) change (add or delete) a WordPress API. Then, we investigate the lag between the time when a new WordPress API is added and when it is used by the plugins. Similarly, we investigate when a WordPress API is removed and when it is removed from the source code of our studied plugins.

Results: Plugins support a new version of WordPress after a median of 38 days with a lag of just one release behind the latest available WordPress version. 28% of the support releases are made within one week after the official release date of a new version of

WordPress, while 38% are made at least 60 days after a new version of WordPress. Our observation holds for version 4 and its minor versions, as shown in Figure 6.4. For instance, only 27% of the studied plugins support WordPress 4.0 within the first week and a median of 22% of the studied plugins support minor versions of WordPress 4.0 (i.e., 4.1 to 4.9) within the first week. The lag (in days) for plugins supporting a new version of WordPress is similar to the lag of updating API changes (a median of 34 days) in the Pharo ecosystem, as reported by Hora et al. (2018), while it is faster than the lag of API adoption in the Android ecosystem (average 14 months), as reported by McDonnell et al. (2013).

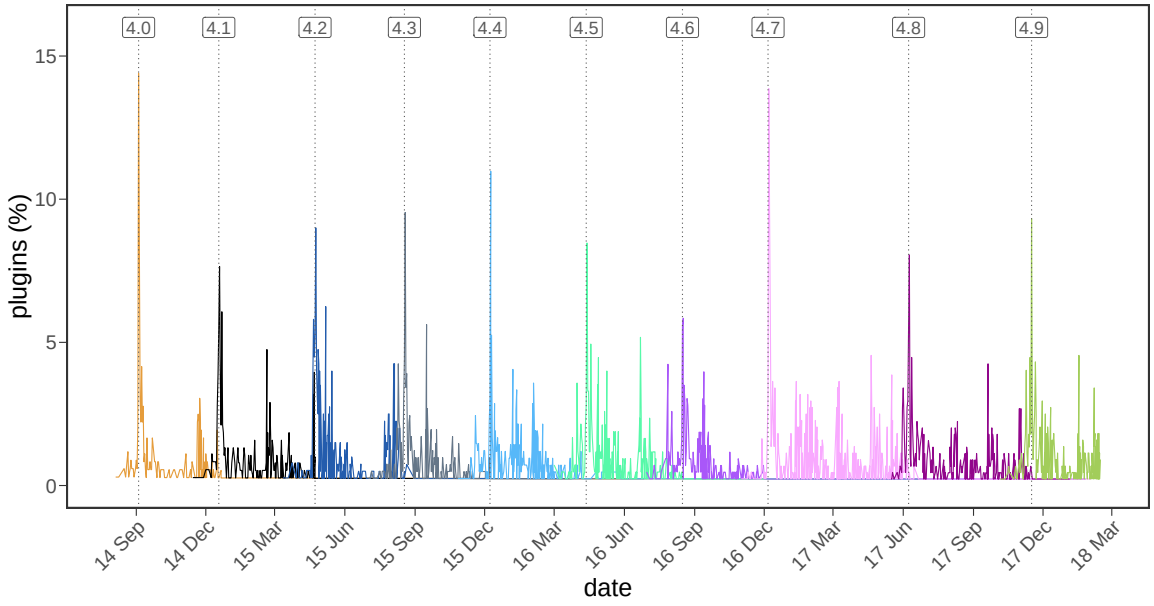


Figure 6.4: The distribution of plugins shipping a support release for a new WordPress version. Early spikes (i.e., early and fast adopters) appear across all minor versions of WordPress version 4. The vertical dashed lines indicate when WordPress released a version. Note that this plot considers just version 4 and we observe similar results for the other WordPress versions (i.e., 2, 3, and 5).

97% of the plugins have at least one support release that is made as late as after a week from the release of the latest WordPress release. For instance, only 3% of the studied plugins always support new versions of WordPress within their first week. The studied plugins support some versions of WordPress as early as within the first week, while the same plugins support other versions as late as after 60 days. For example, *Woocommerce-Checkout-Manager*[13] shipped a release to support WordPress 4.6 on August 19, 2016, which is three days after the release of version 4.6, while the release supporting WordPress 4.9 was introduced 84 days after its release date.

We do not observe any large differences among the four groups of adopters in terms of the chance to be followed by an urgent release. Early, fast, normal and slow adopters have a median of 0%, 0%, 11%, and 13%, respectively, of their releases that are followed by an urgent release. The percentage of normal and slow adopters that are followed by an urgent release is statistically significantly different (Wilcoxon test: p-value $< 2.2e^{-16}$, $\alpha = 0.01$) compared to early and fast adopters, as shown in Figure 6.5. However, this difference has a small effect size (0.26). In particular, 478 (i.e., 96%) out of the studied plugins have at least one early or fast adopter, while a median of 0% of their such adopters are followed by an urgent release.

Although WordPress plugins support new WordPress versions, the plugins make changes according to the new WordPress version after a median of 431 days. For instance, even when plugins support a new WordPress version that removes a WordPress API, the studied plugins keep using these APIs that are completely removed from the source code of WordPress for a median of 873 additional days. WordPress removes an API from its source code in a median of 10 minor versions (i.e., 1,320 days as WordPress releases a minor version in a median of 132 days) after the introduction of the

[13] https://wordpress.org/plugins/woocommerce-checkout-manager/

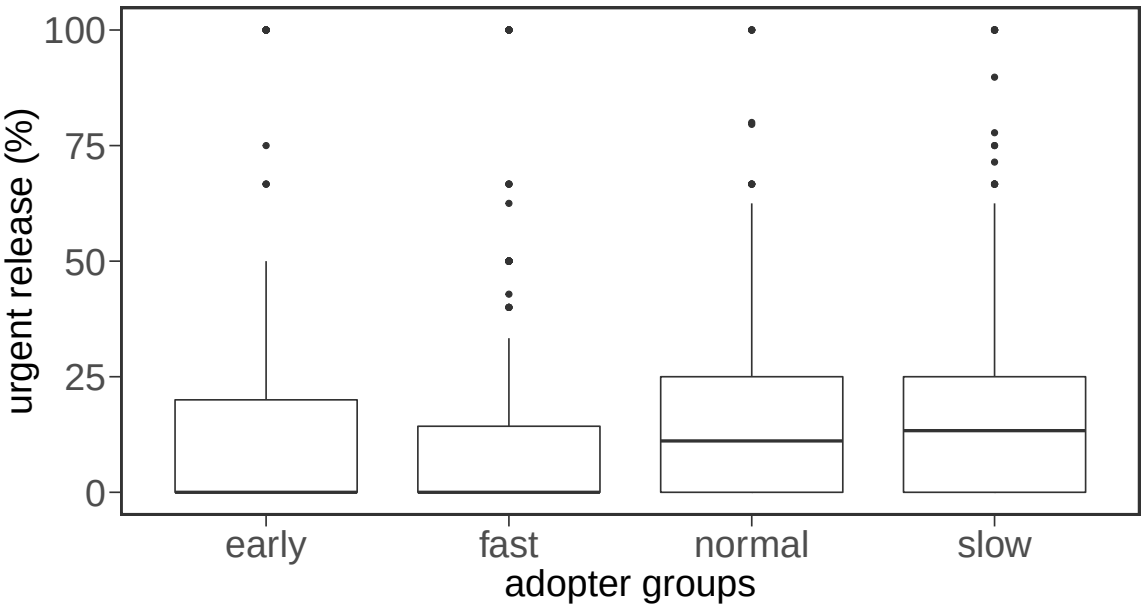


Figure 6.5: The comparison of the distribution of urgent release ratio for support releases based on their adoption speed. Slow and normal adopters are more likely to be followed by an urgent release.

API. Furthermore, most of the API related changes are not made by support releases, but by in-sync releases, as shown in Figure 6.6. For example, *post_permalink()* has been removed since WordPress 4.4[14]. The *photo-gallery* plugin[15] supported WordPress 4.4 one day after its release, while the plugin removed the call to the *post_permalink()* API (in an in-sync release) after 671 days. We also observe that newly introduced APIs are used as fast as 1 day from their addition by WordPress. Finally, 25% of the WordPress APIs were never used by any WordPress plugin. We randomly selected 50 APIs from these APIs to further investigate whether they need special settings (e.g., configuration settings) before using them. 76% of these 50 APIs are to manipulate particular information of a tiny element (e.g., get the last name of the author of the current

[14] https://developer.wordpress.org/reference/functions/post_permalink/
[15] https://wordpress.org/plugins/photo-gallery/

post, determine if a string is well-formatted) on a website. The others (24%) are for administrators to control and maintain their websites, such as getting the size of a directory recursively, handling warnings and notices in each sub-domain of multi-sites. In general, plugins only use these APIs when they support certain functionality (e.g., multi-site management).

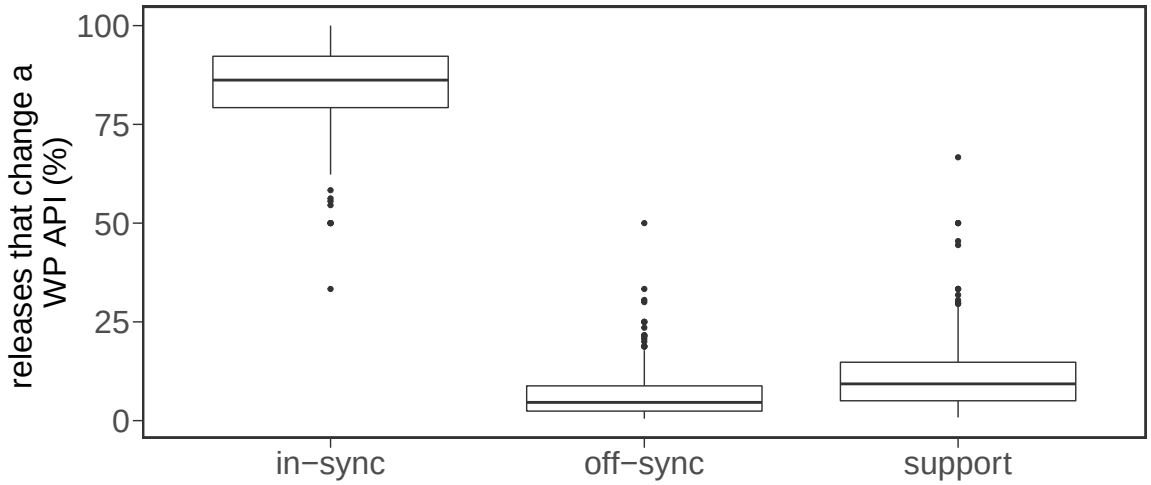


Figure 6.6: The comparison of the percentage of releases that change a WordPress API among in-sync, off-sync and support releases. Plugin developers are more likely to make changes related to the associated WordPress APIs in in-sync releases compared to support releases.

Plugins support a new version of WordPress after a medium of 38 days with a lag of just one release behind the latest available WordPress version. Such a lag period is inconsistent even for the same plugin. Moreover, plugin developers use a new WordPress API as fast as 1 day after its introduction and delete removed APIs after a median of 873 days.

RQ2: How plugins support a new version of WordPress?

Motivation: The goal of this research question is to investigate how plugins support new WordPress releases, so WordPress developers can accordingly design tools to help plugins better support new WordPress versions and reduce incompatibility issues. In particular, this research question investigates how plugins support new WordPress versions from three perspectives: we investigate whether plugins release changes (aka., off-sync releases as shown in Figure 6.3) **before** supporting new WordPress versions, and how likely these off-sync releases will contain issues that will be fixed by an urgent release. We secondly investigate the amount of code changes **during** the support of new WordPress releases. We also investigate **post-support** by studying how likely support releases as well as releases that make a change according to the modification of a WordPress API will be followed by an urgent release.

Approach: To study the practices that are **before the support** of new WordPress versions, we quantify the amount of off-sync releases that plugins have. In particular, we count for each plugin the number of releases that exist between the release of each WordPress version and the plugin's support release for the same new WordPress version. We also compare how often these off-sync releases are followed by an urgent release compared to in-sync releases (i.e., releases that are made after supporting new WordPress versions).

Second, we investigate **the support itself** by studying their code changes. In particular, we compare the amount of code changes of support releases, in-sync, and off-sync releases. We also compare the impact of the amount of code changes on the quality of support releases. The quality is quantified by the chance of a support release to be followed by an urgent release. In particular, we compare the support releases with a

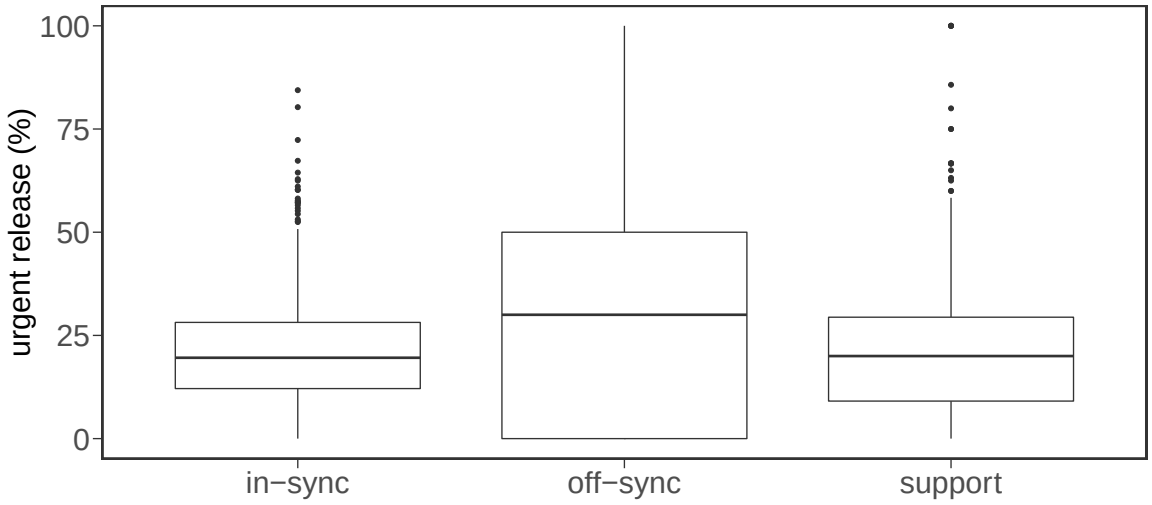


Figure 6.7: The comparison of the distribution of urgent release ratio among in-sync, off-sync and support releases. Off-sync releases are more likely to be followed by an urgent release compared to support and in-sync releases.

large amount of code change (i.e., the top 25% of our studied support releases in terms of code changes) and support releases with a few code changes (i.e., the bottom 25% of our studied support releases). Since plugins do not always use new WordPress APIs or delete removed WordPress APIs in support releases, we also quantify the quality of releases that change (add or delete) a WordPress API. Similarly to the other practices, we measure the quality with the number of urgent releases.

Finally, we study the quality of support releases in comparison with ordinary releases by measuring how likely such releases will be followed by urgent releases. Similarly, we study the quality of releases that make a change according to a WordPress API (e.g., releases that add a new WordPress API or delete a removed WordPress API).

Result: The majority of the studied plugins (i.e., 89%) have at least one off-sync release. Such releases are more likely to be followed by an urgent release (153%) compared to in-sync releases. The studied plugins have a median of 6 off-sync releases.

31% of these off-sync releases are made within one week after a new version of WordPress, while 17% are made even after 60 days. Figure 6.7 shows the boxplots that compare off-sync and in-sync releases in terms of the urgent release ratio (the percentage of the releases that are followed by an urgent release for a given plugin). The difference of the urgent release ratio between off-sync and in-sync releases is statistically significant (Wilcoxon test: p-value $= 2.07e^{-7}$, $\alpha = 0.01$), with a medium effect size (0.49). An example from the *Caldera-Forms* plugin[16] shows how off-sync releases can introduce incompatibility issues with WordPress. For instance, the plugin had an off-sync release that was made 13 days after the official release of WordPress 5.0. The off-sync release was followed by 4 urgent releases that fix incompatibility issues related to WordPress 5.0, as stated in the release notes of the urgent releases (e.g., *"FIXED: Unable to continue with Freemius opt-in after update to WordPress 5.0."*).

While a median of 20% of support releases are followed by an urgent release, a median of 5% of support releases are urgent releases that fix their previous releases. 271 (54%) out of the studied plugins have at least one of its support releases that is also an urgent release. Figure 6.8 indicates that 351 plugins have less than 10% of their support releases that are urgent releases, while 2 plugins have more than 50%. For example, the *planso-forms* plugin[17] shipped an urgent release (i.e., 1.3.2) to support WordPress 4.2 due to an incompatibility issue, as noted in the release note of version 1.3.2: "*Checked for compatibility with WordPress 4.2. Changed the dynamic date and time variables to obey the WordPress settings.*"

Support releases with a large amount (i.e., top 25%) of code changes are more likely to be followed by an urgent release compared to support releases with a few

[16] https://wordpress.org/plugins/caldera-forms
[17] https://wordpress.org/plugins/planso-forms

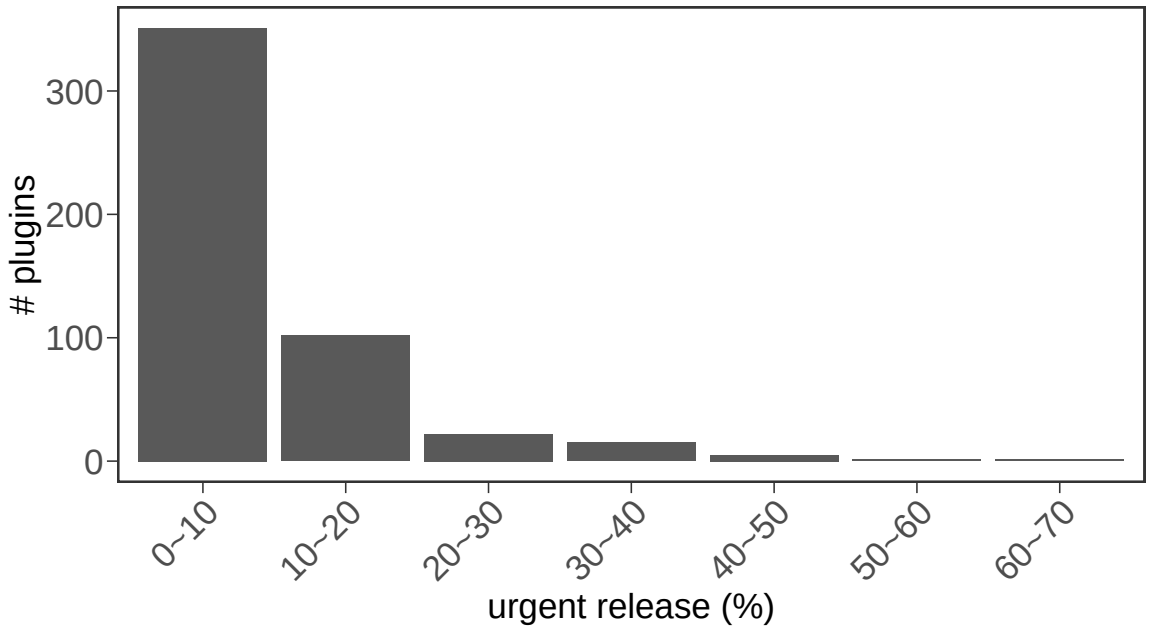


Figure 6.8: The number of plugins that have a certain percentage of its support releases that are urgent releases. For example, 102 plugins have 10-20% of their support releases that are urgent releases.

code changes (bottom 25%). In particular, plugin developers modify a median of 33 lines of code for support releases, while they modify a median of 22 and 13 lines of code for in-sync and off-sync releases, respectively. Figure 6.9 indicates that the difference of the urgent release ratio between support releases with large changes and few changes is statistically significant (Wilcoxon test: p-value $< 2.2e^{-16}$, $\alpha = 0.01$), with a medium effect size (0.6).

Releases with code changes related to WordPress APIs are more likely to be followed by an urgent release compared to ordinary releases (that do not have changes related to WordPress APIs). Figure 6.10 presents the difference between these two kinds of releases is statistically significant (Wilcoxon test: p-value $= 0.005$, $\alpha = 0.01$), with a small effect size (0.15). Although plugin developers are more likely to make

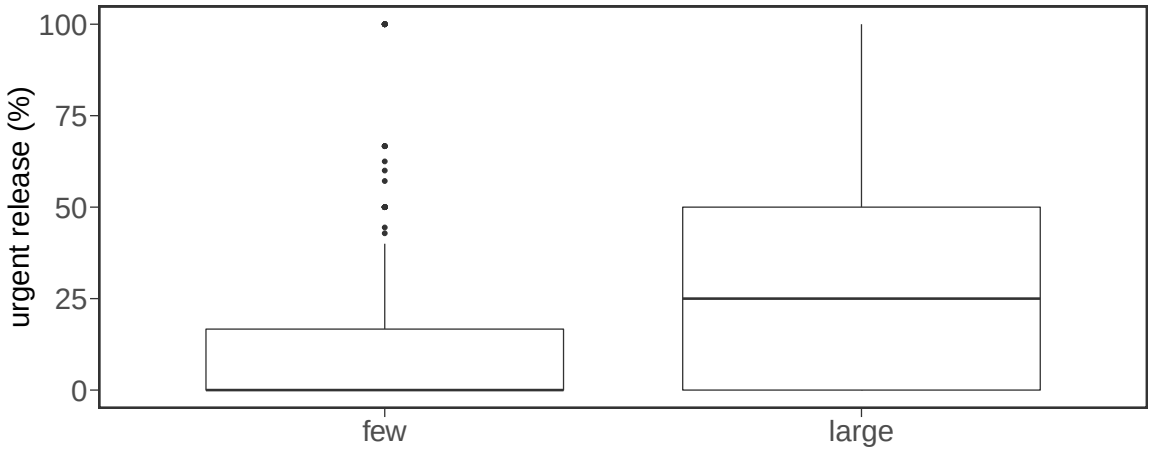


Figure 6.9: The comparison of urgent release ratio for support releases with large v.s. few changes. Support releases with a large amount of code changes are more likely to be followed by an urgent release.

changes related to WordPress APIs in the in-sync releases (Figure 6.6), the ratio of such in-sync releases that are followed by an urgent release (i.e., median of 20%) is lower than support releases that have changes related to WordPress APIs (i.e., median of 31%). Moreover, a median of 22% of in-sync releases that have changes to WordPress APIs are also urgent releases that fix their previous releases.

The majority (89%) of the studied plugins have at least one off-sync release. The possibility of such off-sync releases being followed by an urgent release is 153% times higher than in-sync releases. Support releases with a large amount of code changes are risky since they are more likely to be followed by an urgent release.

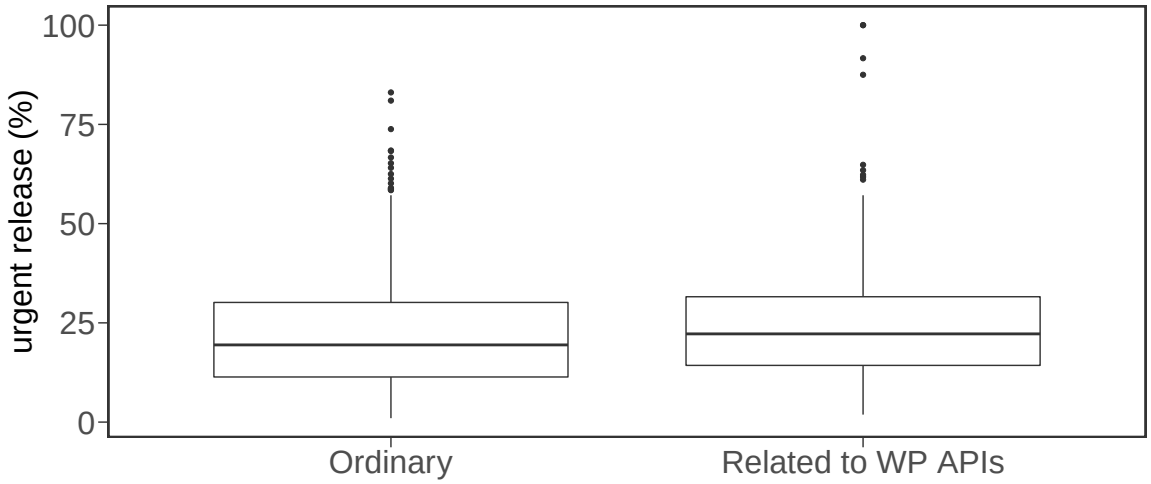


Figure 6.10: The comparison of urgent release ratio for releases with a change related to WordPress APIs v.s. ordinary releases (no changes related to WordPress APIs). Releases with changes related to WordPress APIs are more likely to be followed by an urgent release compared to ordinary releases (that not have changes related to WordPress APIs).

RQ3: How WordPress plugins co-evolve?

Motivation: The goal of this research question is to investigate the co-evolution of plugins. Incompatibility issues do not occur solely between a plugin and its platform, but also between different plugins as discussed in our preliminary study. For example, a user noted an incompatibility issue between the *Jetpack* plugin and the *Yoast-SEO* plugin 2 days after the release of version 7.0 of the *Jetpack* plugin.[18] The problem was resolved once the user reverted the upgrade of *Jetpack* by downgrading to the version 6.9. Therefore, this research question investigates the relation between plugins by studying when a plugin releases a change according to another plugin (aka., *peer-triggered releases*), the quality (in terms of following urgent releases) of such a change, and what resources can be shared between plugins so incompatibilities occur. Our findings can

[18] https://wordpress.org/support/topic/jetpack-7-0-incompatible-with-yoast-seo/

provide a clearer understanding of the co-evolution of plugins, so WordPress can accordingly develop solutions that minimize the incompatibilities between WordPress plugins.

Approach: To investigate how plugins release new updates according to other plugins, we identify related plugins from the **Plugin DataSet**, which is obtained from our preliminary study. Since there is no metadata that indicates which plugin is related to which other plugin, we leverage the description of plugins and their release notes to identify related plugins. In particular, we identify that two plugins are related from one plugin's description or from the release notes of a plugin. A plugin is explicitly related (i.e., **explicit relation**) to another plugin when they extend each other's features according to one of the two plugins' descriptions. For example, the *ecwid-shopping-cart* plugin[19] generates shopping carts for users to place their orders on an e-commerce store. The plugin supports multilingual storefronts by leveraging multilingual plugins (e.g., the *polylang* plugin) to translate the information (e.g., name, description) of products. Therefore, the *ecwid-shopping-cart* plugin has an explicit relation with each of the multilingual plugins. In other cases, two plugins might interfere via an **implicit relation** when they are installed together leading to incompatibility issues. For example, Figure 6.11 shows that there is a relation between the *All-In-One-Wp-Security-and-Firewall* plugin and the *Woocommerce* plugin. We refer to the releases that mention other plugins as *"peer-triggered releases"*. Similar to RQ1 and RQ2, we first measure the quality of peer-triggered releases by calculating the amount of their following urgent releases.

We also investigate a lower-bound time required to fix an incompatibility between

[19] https://wordpress.org/plugins/ecwid-shopping-cart/

plugins. In particular, we investigate the time when plugins ship new releases in response to incompatibilities that were introduced by other plugins. Since we cannot identify for a plugin A that introduces an incompatibility issue that was fixed by a plugin B, we compute the lag between the release of the plugin B that fixes compatibility issues to the closest previous release of the plugin A, as a lower-bound estimation of the time that incompatibility issues between plugins persist.

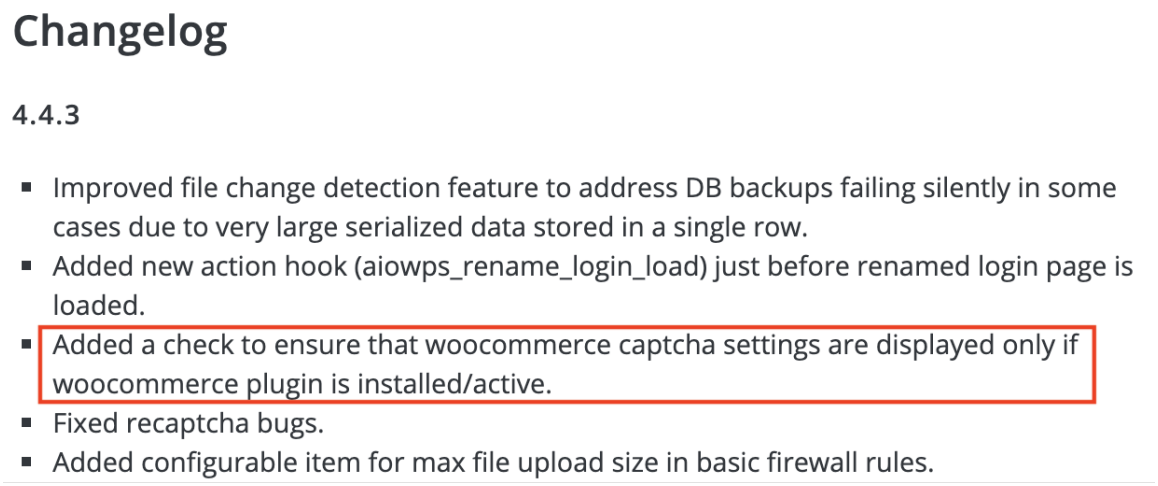


Figure 6.11: An example of a pair of plugins, the *All-In-One-Wp-Security-and-Firewall* plugin and the *Woocommerce* plugin, based on the release note of the the *All-In-One-Wp-Security-and-Firewall* plugin.

After defining a set of related pairs of plugins, we qualitatively investigate what resources plugins could share leading to incompatibility issues. To do so, we conduct a qualitative analysis for a statistically representative random sample (confidence level = 95%, confidence interval = 5%) of 240 peer-triggered releases to provide a catalog of the rationale for two plugins to be related. Among the 240 peer-triggered releases, we manually label each peer-triggered release that addresses incompatibility issues by investigating the release note of that release as well as its source code changes. We label the main causes for incompatibility issues between WordPress plugins in terms

of resources that are shared between plugins. For example, one stated that *"fixed issue setting Featured flag and Catalog Visibility in woocommerce 3.0 and later"* in the release note of version 2.41 of the *pw-bulk-edit* plugin,[20] a plugin for editing product metadata (e.g., name or price) on an e-commerce website. We determine the root cause of the incompatibility issue between the *pw-bulk-edit* and *woocommerce* plugins as layout, since the two plugins shared the same elements (e.g., labels, buttons) on a web page. The developers of the *pw-bulk-edit* plugin tweaked the invoked function for retrieving and displaying information in the elements of the web page in version 2.41.

Result: Plugins have a median of 6 peer-triggered releases. The plugins release changes according to a median of 5 other plugins (not all of these related plugins are in the top 500 most most-released plugins). Figure 6.12 illustrates that the studied plugins are related to one other plugin and as much as 78 other plugins (not all of these related plugins are in the top 500 most most-released plugins). For example, the *Squirrly-SEO* plugin[21] is related to 78 other plugins since it offers smart tools of search engine optimization that collect information from web pages where those plugins are installed to increase the ranking of their websites. In addition, the top three plugins having the largest number of relations with other plugins are *Woocommerce*,[22] *Gutenberg*,[23] and *Buddypress*.[24] *Woocommerce* is a popular eCommerce solution built on WordPress, *Gutenberg* is a flexible editor for users to create fancy content (e.g. media, photo gallery) easily on a website, and *Buddypress* provides a suite of components to integrate a modern, robust, and sophisticated social network into a

[20] https://en-ca.wordpress.org/plugins/pw-bulk-edit/
[21] https://wordpress.org/plugins/squirrly-seo/
[22] https://wordpress.org/plugins/woocommerce/
[23] https://wordpress.org/plugins/gutenberg/
[24] https://wordpress.org/plugins/buddypress/

website. We also observe that 94% of the pairs of related plugins do not share any developers in common.

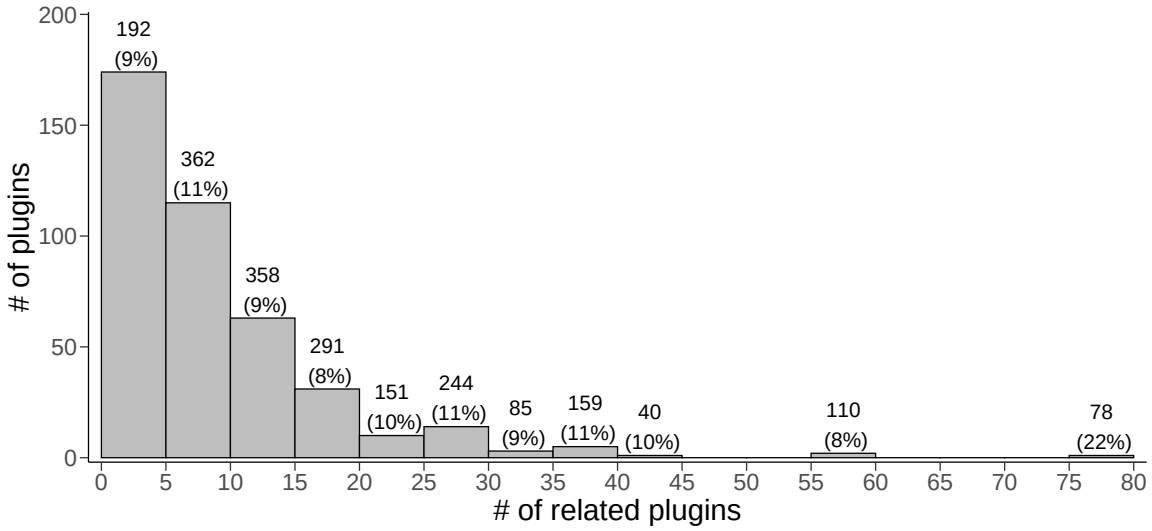


Figure 6.12: The distribution of the number of related plugins for a given plugin. The numbers at the top of each bin indicate the unique number of related plugins and the percentage of these related plugins that are in the top 500 most-released plugins.

Peer-triggered relations do not exist just between the plugins that directly depend on each other. In particular, 60% (144 out of 240) of the qualitatively studied plugins relations are between plugins that do not extend the features of each other or do not directly depend on each other (implicit relations). For example, *Super-Socializer* integrates login, share, and comments from social media into a website.[25] This plugin breaks the features of *Buddypress*, which is a competitor. Hence, the *Super-Socializer* developers have resolved the incompatibility issue and stated that *"[Bugfix] BuddyPress XProfile fields were being wiped out on Social Login, if social profile fields were*

[25] https://wordpress.org/plugins/super-socializer/

not mapped to XProfile fields from plugin options". On the other side, *Woocommerce-Predictive-Search*[26] is a plugin that extends *Woocommerce* with an auto-complete feature so there is an explicit relation between the *Woocommerce-Predictive-Search* and *Woocommerce* plugin.

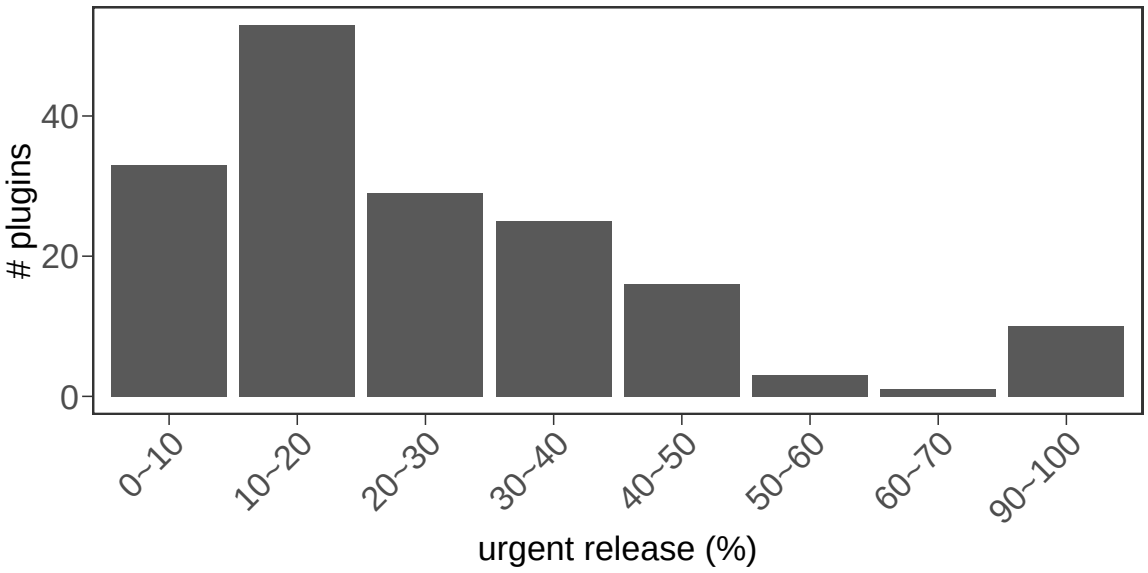


Figure 6.13: The number of plugins that have a certain percentage of its peer-triggered releases that are urgent releases. For example, 53 plugins have 10-20% of its peer-triggered releases that are urgent releases.

Peer-triggered releases are not risky, since peer-triggered releases are less likely to be followed by an urgent release, compared to ordinary releases (that are not associated with other plugins). Figure 6.14 shows that the ratio of urgent releases for peer-triggered releases is statistically significantly (Wilcoxon test: p-value = $1.66e^{-8}$; α = 0.01; effect size is small (0.21)) lower than the ratio of urgent releases that are associated with ordinary releases (that are not associated with other plugins). In addition, a

[26] https://wordpress.org/plugins/woocommerce-predictive-search

median of 20% of peer-triggered releases are urgent releases that fix their previous releases. Figure 6.13 indicates the distribution of plugins that have a certain percentage of its peer-triggered releases that are urgent releases. 170 (34%) out of the studied plugins have at least one peer-triggered release that is an urgent release. For example, one mentioned that "*FIXED: Block was not showing in the editor if Gutenberg was not active and WordPress 5.0.1+*" in the release note of an urgent release for the *caldera-forms* plugin.[27]

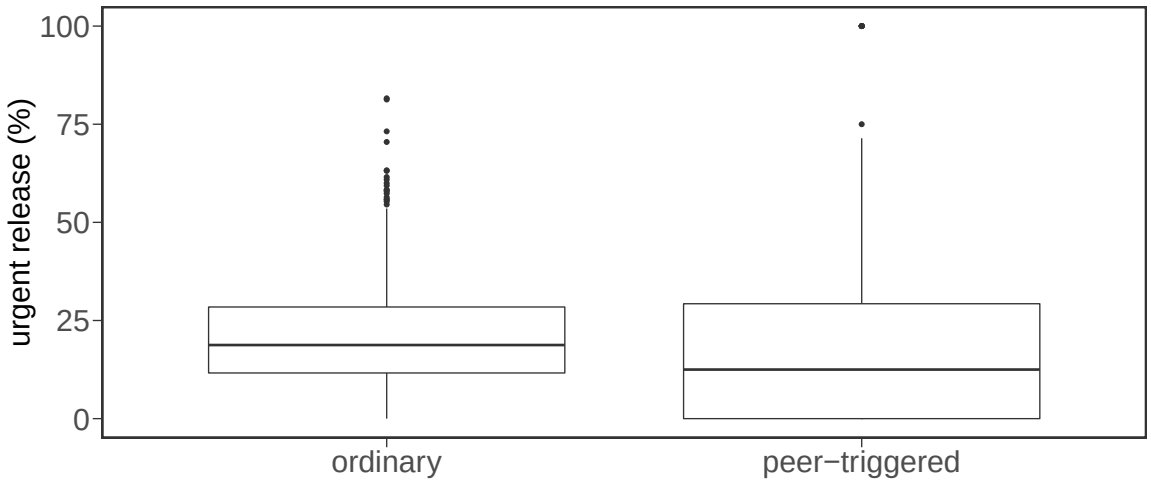


Figure 6.14: The comparison of urgent release ratio for ordinary v.s. peer-triggered releases. Peer-triggered releases are less likely to be followed by an urgent release.

The most common goal of peer-triggered releases is fixing incompatibility issues between peer plugins, as shown in Table 6.2. In fact, 46% of these releases are related to incompatibility issues between plugins. In addition, plugins extend their features by using other plugins, as discussed earlier in the above for the explicit relations between plugins. Plugins also face incompatibility issues in these explicit relations, for which developers know in advance the plugins on which they depend. For example,

[27] https://wordpress.org/plugins/caldera-forms

Table 6.2: The types of relations that exist between a pair of related plugins. Note that a relation can have multiple reasons.

Reason	**Count(%)**
Conflicts between plugins	110 (46%)
Leverage features from other plugins to extend its features	80 (33%)
Adjust/Add functions to offer additional settings for other plugins (e.g. options to implement *Yoast-SEO* breadcrumbs)	72 (30%)
Multiple plugins to form a feature (e.g., master-slave)	25 (10%)
Integrated newly created plugins to benefit to users (e.g., merge settings from different plugins together)	23 (10%)
Protective measurements to avoid conflicts or errors with plugins having a similar feature	8 (7%)

as stated in the release notes of the *Affiliates-Manager* plugin,[28] "*Fixed an issue that was preventing affiliates manager from creating new affiliates during WooCommerce checkout*".

Developers take a long time, i.e., a median of 36 days (lower-bound estimation), before fixing incompatibility issues. Based on our qualitatively studied incompatibilities, we observe that plugin developers take a median of 39 days or 27 days (lower-bound estimation) to fix an incompatibility between a pair of plugins having an explicit or implicit relation, respectively, as shown in Figure 6.15.

Sharing six types of resources is responsible for incompatibility issues between peer plugins, according to our investigation on the main causes for 110 incompatibilities. Table 6.3 presents the six types of resources, which are workflow, layout, data, library, name, and cache that plugins share with each other along with examples. While sharing the workflow and layout is expected because plugins work together to form a web page, sharing data, libraries, names, and caches between plugins is surprising because each plugin is designed for different purposes. Even when WordPress provides

[28] https://wordpress.org/plugins/affiliates-manager/

Table 6.3: The types of resources that are shared between each pair of plugins that show a conflict.

Shared Resource	**Description**	**Example**	**Reason**	**Count(%)**
Data	Data related objects that are used in the operations of a web page (e.g., rendering, forms) are changed or removed by another plugins	"Fix – *Yoast SEO* was overriding Sunshine's own title tags with wrong data for individual galleries and images, not anymore" Wordpress.org (2020f)	Two plugins (i.e., *yoast-seo* and *sunshine*) change data in the same global variable.	58 (53%)
Workflow	The order of invoking functions is impacted when a plugin is installed with other plugins, such as one function is invoked twice.	"Fixed: If you click on the small icon from the *WooCommerce* product editor to insert shortcode, it shows popup header twice" Wordpress.org (2020c)	Two plugins added two textboxes that are sharing the same css-class "*css-textbox*". One of the two plugin added a javascript handler for the mouse-over on any textbox tagged with the *css-textbox* class style. Thus, the handler was executed for both textboxes, as illustrated in Figure 6.16a.	45 (41%)
Layout	HTML elements (e.g., buttons, images) are not located at their proper positions in a web page	"Fixed: Input width and box shadow options are not applied on file type input in *Contact Form 7* widget" Wordpress.org (2020d)	One plugin changes the css class of an element that was created by another plugin after certain behaviors (e.g., mouse over), making users cannot interact with the element any more. See Figure 6.16b for more details.	35 (32%)
Library	A collection of files, programs, and code that assists developers to reduce the implementation time and can be used by different plugins	"Fixed Stripe library conflict if other *Stripe* plugin is installed" Wordpress.org (2020e)	Several plugins include the *stripe-gateway* library in the same web page. When the web page loads the same library at the 2nd time, the variables and functions in the library conflict with the ones that have been loaded by the first plugin.	11 (10%)
Name	The same source code names (e.g., classes, variables, configurations) are repeated across plugins	"Renamed configuration page tab classes to avoid conflicts with other plugins" Wordpress.org (2020a)	Css styles apply to several elements (e.g., labels) on a web page that are sharing the same name but are added by two different plugins.	6 (5%)
Cache	A asset is stored in cache for a web page to enhance the performance of the web page	"Avoid using relative asset URLs which may break caching plugins" Wordpress.org (2020b)	Caching plugins cannot cache the assets without their full urls, leading to errors.	3 (2%)

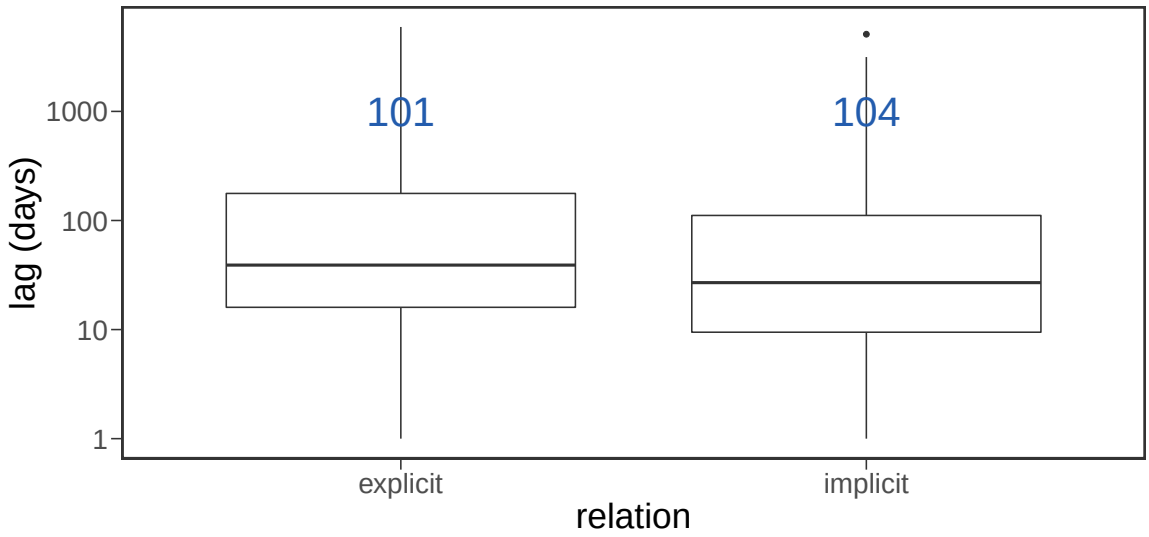
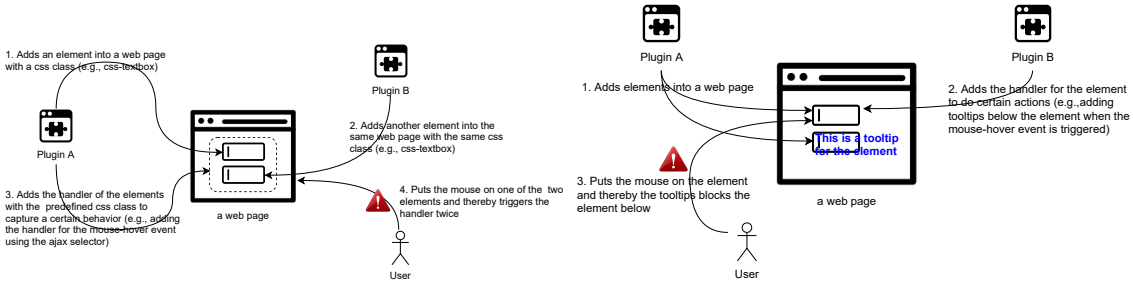


Figure 6.15: The incompatibility last (lower-bound estimation) between pairs of plugins with explicit or implicit relations. The blue number above each boxplot represents the number of pairs of plugins are considered in each relation.

a best practices guideline about the naming conventions[29] of these resources to avoid the incompatibilities, plugins still share these resources.

Our studied plugins have a median of 6 peer-triggered releases. The first important goal of these peer-triggered releases is fixing incompatibility issues, which remain unfixed for 39 days and 27 days for explicitly and implicitly related plugins, respectively. The studied plugins have a median of 20% of their peer-triggered releases are urgent releases. These incompatibilities occur when two plugins share at least one of six types of resources.

[29] https://developer.wordpress.org/plugins/plugin-basics/best-practices/



(a) The explanation of the conflict between two plugins while sharing **workflow** together (the 2nd example).

(b) The explanation of the conflict between two plugins while sharing **layout** together (the 3rd example).

Figure 6.16: The explanation for the examples of sharing resources between plugins in Table 6.3.

6.5 Threats to Validity

6.5.1 Internal Validity

An internal threat to validity is related to the identification of related plugins. For instance, we might learn more possible reasons for incompatibility issues in case we are able to identify all the possible related pairs of plugins. However, our approach that leverages the plugins' descriptions and release notes to identify related plugins allows us to observe that plugins can accidentally interact even if they do not extend each other. Finding more possible relations between plugins will enforce that conclusion. In addition, our approach to identify related plugins allows us to collect a large number of incompatibilities between plugins, which we qualitatively study to identify the possible causes of incompatibility issues. Finding more relations between the WordPress plugins can help identify more possible causes for incompatibility issues. Thus, we encourage future studies to investigate approaches that help identify all possible relations between WordPress plugins.

We also do not generalize our results to the WordPress changes that can be triggered by the plugins. As the plugins are designed to extend the functionality of WordPress instead of the opposite, our study focuses on how the plugins react to the changes of WordPress and whether there is any relation between the plugins themselves. While there is no systematic way to identify whether a WordPress change is triggered by a plugin, we observed certain examples where a plugin developer suggested WordPress certain improvements,[30] which was integrated into WordPress after several versions (from 2.9 to 4.4). Thus, we encourage future work to investigate how WordPress reacts to the suggestions made by the plugins.

6.5.2 External Validity

Similarly to prior studies (Eshkevari et al., 2014; Kyaw et al., 2015; Moslehi et al., 2018; Nguyen et al., 2014; Sayagh and Adams, 2015; Sayagh et al., 2017) that focused just on WordPress, we do not generalize our findings to other content management systems (e.g., Drupal and Joomla). However, WordPress is the most popular CMS platform and has been used by more than 42% of the websites whose content management system (i.e., more than 65% of market share) is known (W3techs.com, 2020), compared to other popular CMSs, such as 1.5% for Drupal and 2.1% for Joomla. In addition, WordPress has a structured historical repository that allows us to collect data for analyzing the co-evolution between the plugins and their platform. That is not the case for Joomla, as it does not have any hosted code repositories for its plugins so we cannot analyze the plugins changes according to the changes to the Joomla platform. Drupal has a

[30] https://core.trac.wordpress.org/ticket/11334

completely different release strategy for plugins, compared to WordPress. The developers of Drupal plugins specify a version of Drupal when they create a plugin. In other words, each Drupal plugin only works for a particular version of Drupal, which is completely different from WordPress and its plugins. In fact, users can install any version of WordPress plugins on any version of WordPress. Hence, the results we learn from our study might not be applicable to Drupal. That said, Drupal and Joomla have many incompatibility issues between the platform and plugins (e.g., the *config-terms* plugin was incompabilitible with Drupal 9.2[31], the *admin-tools* plugin[32] was incompatible with Joomla 4.0.3[33]) and we encourage future studies to investigate the incompatibilities in these other CMSs. Nevertheless, a large number of posts on incompatibilities between WordPress and its plugins or between plugins with each other are reported on the WordPress support forum, indicating how common the incompatibility issues are.[34] For example, one stated an incompatibility between a plugin and WordPress 5.3. "*When I click on Add New Post, Blank White screen appears.*"[35]

6.6 Chapter Summary

A plugin-based software ecosystem, such as WordPress, is an ecosystem that consists of a platform that can be extend by additional components, that are called plugins. Differently from traditional ecosystems, the platform and the plugins of a plugin-based software ecosystem have a symbiotic relation. The platform needs the plugins, which

[31] https://www.drupal.org/project/config_terms/issues/3200627
[32] https://extensions.joomla.org/extension/admin-tools/
[33] https://www.akeeba.com/support/admin-tools/Ticket/35795:extensions-not-compatible-with-joomla-4-0-3.html
[34] https://wordpress.org/search/incompatibility
[35] https://wordpress.org/support/topic/incompatible-with-wp-5-3-gutenberg/

in turn require the platform. Such a high coupling between the platform and the plugins increases the chances for incompatibility issues. In fact, users hesitate to upgrade their WordPress and plugins due to incompatibility issues. According to our preliminary study, plugins often miss relations with the platform or other plugins, which cause important incompatibility issues that are as important to be rapidly addressed via urgent updates. 32% and 19% of the incompatibilities that are fixed through urgent updates are between a plugin and its platform and between a pair of plugins, respectively.

In this chapter, we conduct an empirical study that investigates how plugins ship releases in a plugin-based software ecosystem, so we can get a clear understanding of how plugins and the platform as well as peer plugins co-evolve. Such understanding can guide future studies and WordPress developers better support such co-evolution by designing mechanisms and evaluating their impact on reducing incompatibility issues. For example, WordPress can design mechanisms for notifying that a new version will be released and which APIs will be removed in the new version. These mechanisms can also warn plugin developers when they accidentally share resources (e.g., unique css classes, variable names) with other plugins. Our findings suggest that (RQ1) even if plugins support the latest WordPress versions, they do so after a long period. Even worse, they make changes according to the WordPress APIs after a long period (i.e., a median of 431 days). During such an-out-of sync period, plugins can manifest incompatibility issues with their platform. (RQ2) Plugins release risky changes that are out-of-sync with WordPress releases, whereas the support releases and especially those with a large source code modification are risky. Finally, (RQ3) incompatibility issues occur not just between plugins that extend each other's features, and these incompatibilities occur when plugins accidentally share six types of resources with each other.

CHAPTER 7

Conclusion and future work

DUE to the rise of open source software ecosystems and the distribution of developers around the world, coordination is inevitable to maintain a sustainable and thriving software ecosystem. However, most prior works considered coordination on project-based ecosystems. In this thesis, we propose to leverage data from large-scale software ecosystems to understand current coordination practices for software quality assurance and provide actionable information to practitioners. In particular, we study four specific coordination phenomena: upstream bug coordination, vulnerability coordination, multi-party coordination on fixing vulnerabilities, and coordination of co-evolving software projects. We then perform an empirical analysis of such data to investigate how developers coordinate behind these phenomena, cost of the coordination and impact in the ecosystem.

7.1 Thesis Contributions

Based on our empirical observations, we highlight the main contributions of this thesis below:

- **Fixing high-severity upstream bugs.** In Chapter 3, we observed that upstream packages are dominant across the studied distributions. A similar percentage of high-severity upstream bugs are explicitly recorded to be fixed by upstream (less than 14%) and by distribution maintainers (at least 13%). However, the vast majority (at least 73%) of high-severity upstream bugs are unknown who fixed them, suggesting a lack of traceability between high-severity upstream bugs and their fixes and potentially impacting the quality assurance activities in the distributions. The cost of fixing a high-severity bug in an upstream package by the distribution maintainers requires a longer needed time when users wait for the fix and more reference links per comment, compared to fixing high-severity bugs in in-house packages. However, the number of participants per day and the number of comments per day are similar. In addition, our explanatory models suggest that maintainers search for bug fixing history in the upstream projects before starting to fix an upstream bug.

- **Coordinating vulnerability fixes.** In Chapter 4, we observed that the reported vulnerabilities are Common Security Vulnerabilities (CSVs) to both the studied distributions. Distribution maintainers collaborate closely across distribution boundaries before vulnerability fixes are available, but less afterward, i.e., integrating and testing the fixes on their own. Distribution maintainers take less time

to integrate vulnerability fixes for CSVs than testing the fixes. In addition, we observed that Debian maintainers are more likely to integrate their own patches for CSVs, while Fedora maintainers tend to integrate new upstream releases.

- **Multi-party coordination on fixing vulnerabilities.** In Chapter 5, we observed that the top 13 CNAs with the highest number of assigned CVEs usually assign CVEs for vulnerabilities in their own products, i.e., self-assigned vulnerabilities. However, the vulnerabilities assigned by other CVEs, i.e., delegated vulnerabilities, are more risky, especially those that affecting operating-system related products and those that affecting both operating-system and application related products. Additionally, the patch coherence for vulnerabilities varies from a median of 35 days up to more than one year.

- **Release coordination of co-evolving software projects.** In Chapter 6, we observed that, in a plugin-based ecosystem, even if plugins support the latest available versions of WordPress, they do so after a long period. Even worse, they make changes according to a long period (i.e., a median of 431 days). During such an out-of-sync period, plugins can manifest incompatibilities with their platform. Moreover, incompatibilities occur not only just between plugins and the platform but also between peer plugins when they accidentally share six types of resources with each other.

7.2 Future Research

In the following, we list future research avenues that can leverage our results.

- *Future research could investigate automated tools to improve the traceability and*

transparency of coordination activities between upstream projects and downstream distributions. We observe the vast majority (at least 73%) of the history of coordination activities are not explicitly recorded as both upstream developers and distribution maintainers have their own freedom w.r.t. including reference fixes/bug ids in their own issue tracking systems. Such tools could help not only assist distribution maintainers on maintaining their packages but also enable the opportunities for researchers to improve the coordination model.

- *Future research could investigate how derivative distributions fix bugs that are inherited from their parent distributions.* Each Linux distribution could have its children that inherit its packages in which the parent distribution is responsible for fixing bugs in those packages. For example, Ubuntu is one of Debian's children. In this thesis, we do not investigate the impact of quality assurance around coordination activities on derivative distributions.

- *Future research could explore vulnerability coordination mechanisms that could reduce the risk of vulnerability exploitation.* We observe that embargoed high-severity vulnerabilities have quicker fixes for users than non-embargoed ones. However, embargoing a vulnerability and fixing it in secret requires additional costs, so Fedora applies the embargo policy only on high-severity vulnerabilities, while Debian does not apply the policy at all. As we observe that distribution maintainers collaborate closely across distribution boundaries but less afterward, it is important to understand why distribution maintainers are not able to reuse peer fixes or develop tools to assist them to do so.

- *Future research should be performed for delegated vulnerabilities as they are at*

high-severity and the release of their fixes is slower than self-assigned ones. In the thesis, we study the release speed of delegated vulnerabilities in terms of affected product types and the affected number of software vendors. Future research could explore on the other aspects (e.g., patch complexity).

Bibliography

(2022). CNA. last accessed: 2022-05-02.

(2022). CWE. last accessed: 2022-05-02.

(2022). National vulnerability database. last accessed: 2022-05-02.

Adams, B., Kavanagh, R., Hassan, A. E., and German, D. M. (2016). An empirical study of integration activities in distributions of open source software. *Empirical Software Engineering (EMSE'16)*, 21(3):960–1001.

Al Sabbagh, B. and Kowalski, S. (2015). A socio-technical framework for threat modeling a software supply chain. *IEEE Security & Privacy*, 13(4):30–39.

Alfadel, M., Costa, D. E., and Shihab, E. (2021). Empirical analysis of security vulnerabilities in python packages. In *2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER'21)*, pages 446–457. IEEE.

Algarni, A. and Malaiya, Y. (2014). Software vulnerability markets: Discoverers and buyers. *International Journal of Computer, Information Science and Engineering*, 8(3):71–81.

Alhazmi, O. H. and Malaiya, Y. K. (2008). Application of vulnerability discovery models to major operating systems. *IEEE Transactions on Reliability*, 57(1):14–22.

Alipour, A., Hindle, A., and Stroulia, E. (2013). A contextual approach towards more accurate duplicate bug report detection. In *2013 10th Working Conference on Mining Software Repositories (MSR'13)*, pages 183–192. IEEE.

Allodi, L. (2017). Economic factors of vulnerability trade and exploitation. In *Proceedings of the 2017 ACM SIGSAC conference on computer and communications security (CCS'17)*, pages 1483–1499.

Anbalagan, P. and Vouk, M. (2009). On predicting the time taken to correct bug reports in open source projects. In *Proceedings of the 25th IEEE International Conference on Software Maintenance (ICSM'09)*, pages 523–526. IEEE.

Anderson, R. (2002). Security in open versus closed systems—the dance of boltzmann, coase and moore. Technical report, Technical report, Cambridge University, England.

Arora, A., Krishnan, R., Nandkumar, A., Telang, R., and Yang, Y. (2004). Impact of vulnerability disclosure and patch availability-an empirical analysis. In *Third Workshop on the Economics of Information Security*, volume 24, pages 1268–1287.

Ashraf, U., Mayr-Dorn, C., Mashkoor, A., Egyed, A., and Panichella, S. (2021). Do communities in developer interaction networks align with subsystem developer teams?

an empirical study of open source systems. In *Proceedings of the 16th ACM/IEEE International Conference on Global Software Engineering (ICGSE'21)*, pages 61–71. IEEE.

Barbour, L., Khomh, F., and Zou, Y. (2011). Late propagation in software clones. In *Proceedings of the 27th IEEE International Conference on Software Maintenance (ICSM'11)*, pages 273–282. IEEE.

Barr, E. T., Harman, M., Jia, Y., Marginean, A., and Petke, J. (2015). Automated software transplantation. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis (ISSTA'15)*, pages 257–269.

Bavota, G., Canfora, G., Di Penta, M., Oliveto, R., and Panichella, S. (2013). The evolution of project inter-dependencies in a software ecosystem: The case of Apache. In *Proceedings of the 29th IEEE International Conference on Software Maintenance (ICSM'13)*, pages 280–289. IEEE.

Bavota, G., Canfora, G., Di Penta, M., Oliveto, R., and Panichella, S. (2015). How the apache community upgrades dependencies: an evolutionary study. *Empirical Software Engineering (EMSE'15)*, 20(5):1275–1317.

Bavota, G., Linares-Vasquez, M., Bernal-Cardenas, C. E., Di Penta, M., Oliveto, R., and Poshyvanyk, D. (2014). The impact of api change-and fault-proneness on the user ratings of android apps. *IEEE Transactions on Software Engineering (TSE'14)*, 41(4):384–407.

Bettenburg, N. and Hassan, A. E. (2013). Studying the impact of social interactions on software quality. *Empirical Software Engineering (EMSE'13)*, 18(2):375–431.

Bettenburg, N., Premraj, R., Zimmermann, T., and Kim, S. (2008). Duplicate bug reports considered harmful... really? In *2008 IEEE International Conference on Software Maintenance (ICSM'08)*, pages 337–345. IEEE.

BHARGAVA, G. and CHANDINI, M. (2018). Then and now: On the maturity of the cybercrime markets.

Bhattacharya, P. and Neamtiu, I. (2011). Bug-fix time prediction models: can we do better? In *Proceedings of the 8th Working Conference on Mining Software Repositories (MSR'11)*, pages 207–210.

Bilge, L. and Dumitraş, T. (2012). Before we knew it: an empirical study of zero-day attacks in the real world. In *Proceedings of the 2012 ACM conference on Computer and communications security (CCS'12)*, pages 833–844.

Bird, C. (2011). Sociotechnical coordination and collaboration in open source software. In *Proceedings of the 27th IEEE International Conference on Software Maintenance (ICSM'11)*, pages 568–573. IEEE.

Bird, C. and Zimmermann, T. (2012). Assessing the value of branches with what-if analysis. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering (FSE'12)*, pages 1–11.

Blocken, B. (2014). 50 years of computational wind engineering: past, present and future. *Journal of Wind Engineering and Industrial Aerodynamics*, 129:69–102.

Bogart, C., Kästner, C., Herbsleb, J., and Thung, F. (2016). How to break an api: cost negotiation and community values in three software ecosystems. In *Proceedings of*

the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, pages 109–120.

Boisselle, V. and Adams, B. (2015). The impact of cross-distribution bug duplicates, empirical study on debian and ubuntu. In *2015 IEEE 15th International Working Conference on Source Code Analysis and Manipulation (SCAM'15)*, pages 131–140. IEEE.

Brindescu, C., Ahmed, I., Jensen, C., and Sarma, A. (2020). An empirical investigation into merge conflicts and their effect on software quality. *Empirical Software Engineering (EMSE'20)*, 25(1):562–590.

Brito, A., Valente, M. T., Xavier, L., and Hora, A. (2020). You broke my code: understanding the motivations for breaking changes in apis. *Empirical Software Engineering*, 25(2):1458–1492.

Brito, G., Hora, A., Valente, M. T., and Robbes, R. (2016). Do developers deprecate APIs with replacement messages? a large-scale analysis on Java systems. In *Proceedings the 23rd IEEE International Conference on Software Analysis, Evolution, and Reengineering (SANER'16)*, volume 1, pages 360–369. IEEE.

Brito, G., Hora, A., Valente, M. T., and Robbes, R. (2018). On the use of replacement messages in API deprecation: An empirical study. *Journal of Systems and Software (JSS'18)*, 137:306–321.

Brun, Y., Holmes, R., Ernst, M. D., and Notkin, D. (2011). Proactive detection of collaboration conflicts. In *Proceedings of the 19th ACM SIGSOFT symposium and the*

13th European Conference on Foundations of Software Engineering (FSE'11), pages 168–178.

Brun, Y., Holmes, R., Ernst, M. D., and Notkin, D. (2013). Early detection of collaboration conflicts and risks. *IEEE Transactions on Software Engineering (TSE'13)*, 39(10):1358–1375.

Canfora, G., Cerulo, L., Cimitile, M., and Di Penta, M. (2011). Social interactions around cross-system bug fixings: the case of FreeBSD and OpenBSD. In *Proceedings of the 8th Working Conference on Mining Software Repositories (MSR'11)*, pages 143–152.

Canfora, G., Di Sorbo, A., Forootani, S., Pirozzi, A., and Visaggio, C. A. (2020). Investigating the vulnerability fixing process in OSS projects: Peculiarities and challenges. *Computers & Security*, 99:102067.

Cataldo, M. and Herbsleb, J. D. (2012). Coordination breakdowns and their impact on development productivity and software failures. *IEEE Transactions on Software Engineering (TES'12)*, 39(3):343–360.

Cataldo, M., Herbsleb, J. D., and Carley, K. M. (2008). Socio-technical congruence: a framework for assessing the impact of technical and work dependencies on software development productivity. In *Proceedings of the Second ACM-IEEE international symposium on Empirical software engineering and measurement*, pages 2–11.

Cataldo, M., Wagstrom, P. A., Herbsleb, J. D., and Carley, K. M. (2006). Identification of coordination requirements: Implications for the design of collaboration and awareness tools. In *Proceedings of the 20th Anniversary Conference on Computer Supported Cooperative Work (CSCW'06)*, pages 353–362.

Chen, X., Zhao, Y., Cui, Z., Meng, G., Liu, Y., and Wang, Z. (2019). Large-scale empirical studies on effort-aware security vulnerability prediction methods. *IEEE Transactions on Reliability*, 69(1):70–87.

Chinthanet, B., Kula, R. G., McIntosh, S., Ishio, T., Ihara, A., and Matsumoto, K. (2021). Lags in the release, adoption, and propagation of npm vulnerability fixes. *Empirical Software Engineering (EMSE'21)*, 26(3):1–28.

Christey, S. and Martin, B. (2013). Buying into the bias: Why vulnerability statistics suck. *BlackHat, Las Vegas, USA, Tech. Rep*, 1.

Claes, M., Mens, T., Di Cosmo, R., and Vouillon, J. (2015). A historical analysis of debian package incompatibilities. In *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories (MSR'15)*, pages 212–223. IEEE.

Cogo, F. R., Oliva, G. A., and Hassan, A. E. (2019). An empirical study of dependency downgrades in the npm ecosystem. *IEEE Transactions on Software Engineering (TES'19)*, 47(11):2457–2470.

Crowston, K. and Scozzi, B. (2008). Bug fixing practices within free/libre open source software development teams. *Journal of Database Management (JDM'08)*, 19(2):1–30.

da Costa, D. A., Abebe, S. L., McIntosh, S., Kulesza, U., and Hassan, A. E. (2014). An empirical study of delays in the integration of addressed issues. In *Proc. of the 30th Int'l Conf. on Software Maintenance and Evolution (ICSME'14)*, pages 281–290.

da Costa, D. A., McIntosh, S., Treude, C., Kulesza, U., and Hassan, A. E. (2018). The

impact of rapid release cycles on the integration delay of fixed issues. *Empirical Software Engineering (EMSE'18)*, 23(2):835–904.

Datta, S. (2018). How does developer interaction relate to software quality? an examination of product development data. *Empirical Software Engineering (EMSE'18)*, 23(3):1153–1187.

Davies, J., Zhang, H., Nussbaum, L., and German, D. M. (2010). Perspectives on bugs in the debian bug tracking system. In *2010 7th IEEE Working Conference on Mining Software Repositories (MSR'10)*, pages 86–89. IEEE.

Debian.org (2020a). Debian - list of package categories. last accessed: 2020-12-01.

Debian.org (2020b). Developer's reference. https://www.debian.org/doc/manuals/developers-reference/index.en.html. last accessed: 2020-12-01.

Debian.org (2020c). Information regarding the bug processing system for package maintainers and bug triagers. https://www.debian.org/Bugs/Developer. last accessed: 2020-12-01.

Debian.org (2020d). Managing packages. https://www.debian.org/doc/manuals/developers-reference/pkgs.html. last accessed: 2020-12-01.

Debian.org (2021a). Debian continuous integration. https://ci.debian.net/doc/. last accessed: 2021-06-02.

Debian.org (2021b). Quilt for debian maintainers. https://perl-team.pages.debian.net/howto/quilt.html. last accessed: 2021-12-01.

Debian.org (2021c). Reasons to use debian. https://www.debian.org/intro/why_debian. last accessed: 2021-06-02.

Debian.org (2021d). Upstream guide. https://wiki.debian.org/UpstreamGuide. last accessed: 2021-12-01.

Debian.org (2021e). Vulnerability disclosure policy. https://www.debian.org/security/disclosure-policy. last accessed: 2021-06-02.

Decan, A., Mens, T., and Claes, M. (2017a). An empirical comparison of dependency issues in oss packaging ecosystems. In *Proceedings of the 24th IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER'17)*, pages 2–12. IEEE.

Decan, A., Mens, T., and Claes, M. (2017b). An empirical comparison of dependency issues in OSS packaging ecosystems. In *Proceedings of the 24th IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER'17)*, pages 2–12. IEEE.

Decan, A., Mens, T., Claes, M., and Grosjean, P. (2016). When github meets cran: An analysis of inter-repository package dependency problems. In *Proceedings of the 23rd IEEE International Conference on Software Analysis, Evolution, and Reengineering (SANER'16)*, volume 1, pages 493–504. IEEE.

Decan, A., Mens, T., and Constantinou, E. (2018a). On the evolution of technical lag in the npm package dependency network. In *Proceedings of the 34th IEEE International Conference on Software Maintenance and Evolution (ICSME'18)*, pages 404–414. IEEE.

Decan, A., Mens, T., and Constantinou, E. (2018b). On the impact of security vulnerabilities in the npm package dependency network. In *Proceedings of the 15th International Conference on Mining Software Repositories (MSR'18)*, pages 181–191.

Desouza, K. C., Awazu, Y., and Baloh, P. (2006). Managing knowledge in global software development efforts: Issues and practices. *IEEE software*, 23(5):30–37.

Ding, H., Ma, W., Chen, L., Zhou, Y., and Xu, B. (2017). An empirical study on downstream workarounds for cross-project bugs. In *2017 24th Asia-Pacific Software Engineering Conference*, pages 318–327. IEEE.

Dolan-Gavitt, B., Hulin, P., Kirda, E., Leek, T., Mambretti, A., Robertson, W., Ulrich, F., and Whelan, R. (2016). Lava: Large-scale automated vulnerability addition. In *2016 IEEE Symposium on Security and Privacy (SP'16)*, pages 110–121. IEEE.

Duc, A. N., Cruzes, D. S., Ayala, C., and Conradi, R. (2011). Impact of stakeholder type and collaboration on issue resolution time in oss projects. In *IFIP International Conference on Open Source Systems*, pages 1–16. Springer.

Ellison, R. J., Goodenough, J. B., Weinstock, C. B., and Woody, C. (2010). Evaluating and mitigating software supply chain security risks. Technical report, Carnegie-Mellon Univ Pittsburgh PA Software Engineering Inst.

Eshkevari, L., Antoniol, G., Cordy, J. R., and Di Penta, M. (2014). Identifying and locating interference issues in php applications: the case of wordpress. In *Proceedings of the 22nd International Conference on Program Comprehension (ICPC'14)*, pages 157–167.

Espinha, T., Zaidman, A., and Gross, H.-G. (2014). Web API growing pains: Stories from client developers and their code. In *2014 Software Evolution Week-IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering (CSMR-WCRE'14)*, pages 84–93. IEEE.

Fan, Y., Xia, X., Lo, D., and Hassan, A. E. (2018). Chaff from the wheat: Characterizing and determining valid bug reports. *IEEE Transactions on Software Engineering (TSE'18)*, 46(5):495–525.

Farhang, S., Kirdan, M. B., Laszka, A., and Grossklags, J. (2019). Hey google, what exactly do your security patches tell us? a large-scale empirical study on android patched vulnerabilities. *arXiv preprint arXiv:1905.09352*.

Fedora.org (2020a). Anitya - upstream release monitoring system. https://release-monitoring.org/. last accessed: 2020-12-01.

Fedora.org (2020b). Bug status workflow. https://fedoraproject.org/wiki/BugZappers/BugStatusWorkFlow. last accessed: 2020-12-01.

Fedora.org (2020c). Join the package collection maintainers. https://fedoraproject.org/wiki/Join_the_package_collection_maintainers#Create_a_Fedora_Account. last accessed: 2020-12-01.

Fedora.org (2020d). Packaging guidelines. https://docs.fedoraproject.org/en-US/packaging-guidelines/#changelogs. last accessed: 2020-12-01.

Fedora.org (2020e). Upstream release monitoring. https://fedoraproject.org/wiki/Upstream_release_monitoring. last accessed: 2020-12-01.

Fedora.org (2021a). Fedora - creating a patch. https://docs.fedoraproject.org/en-US/Fedora_Draft_Documentation/0.1/html/Documentation_Guide/sect-workflow-patching.html. last accessed: 2021-12-01.

Fedora.org (2021b). Security bugs. https://fedoraproject.org/wiki/Security_Bugs. last accessed: 2021-06-02.

Fedora.org (2021c). Staying close to upstream projects. https://fedoraproject.org/wiki/Staying_close_to_upstream_projects. last accessed: 2021-12-01.

Feutrill, A., Roughan, M., Ross, J., and Yarom, Y. (2020). A queueing solution to reduce delay in processing of disclosed vulnerabilities. In *2020 Second IEEE International Conference on Trust, Privacy and Security in Intelligent Systems and Applications*, pages 1–11. IEEE.

First.org (2021). Guidelines and practices for multi-party vulnerability coordination and disclosure. https://www.first.org/global/sigs/vulnerability-coordination/multiparty/guidelines-v1.1. last accessed: 2021-06-02.

First.org (2022). Guidelines and practices for multi-party vulnerability coordination and disclosure. https://www.first.org/global/sigs/vulnerability-coordination/multiparty/guidelines-v1.0. last accessed: 2022-05-02.

Foundjem, A. and Adams, B. (2021). Release synchronization in software ecosystems. *Empirical Software Engineering*, 26(3):1–50.

Freedesktop.org (2013). Upstream guidelines for linux distributions. https://www.freedesktop.org/wiki/Distributions/Packaging/WhyUpstream/. last accessed: 2021-12-01.

Frei, S., May, M., Fiedler, U., and Plattner, B. (2006). Large-scale vulnerability analysis. In *Proceedings of the 2006 SIGCOMM workshop on Large-scale attack defense (LSAD'06)*, pages 131–138.

Frei, S., Tellenbach, B., and Plattner, B. (2008). 0-day patch exposing vendors (in) security performance. *BlackHat Europe*.

Fruhwirth, C. and Mannisto, T. (2009). Improving CVSS-based vulnerability prioritization and response with context information. In *Proceedings of the 3rd International symposium on empirical software engineering and measurement*, pages 535–544. IEEE.

Gharehyazie, M., Posnett, D., Vasilescu, B., and Filkov, V. (2015). Developer initiation and social interactions in oss: A case study of the Apache software foundation. *Empirical Software Engineering (EMSE'15)*, 20(5):1318–1353.

Giger, E., Pinzger, M., and Gall, H. (2010). Predicting the fix time of bugs. In *Proceedings of the 2nd International Workshop on Recommendation Systems for Software Engineering*, pages 52–56.

Github.com (2020). Github - securing the world's software. https://octoverse.github.com/static/github-octoverse-2020-security-report.pdf. last accessed: 2021-06-02.

Goyal, P., Parmar, V., Rishi, R., et al. (2011). Manet: vulnerabilities, challenges, attacks, application. *IJCEM International Journal of Computational Engineering & Management*, 11(2011):32–37.

Grieco, G., Grinblat, G. L., Uzal, L., Rawat, S., Feist, J., and Mounier, L. (2016). Toward large-scale vulnerability discovery using machine learning. In *Proceedings of the Sixth ACM Conference on Data and Application Security and Privacy*, pages 85–96.

Guimarães, M. L. and Silva, A. R. (2012). Improving early detection of software merge conflicts. In *Proceedings of the 34th International Conference on Software Engineering (ICSE'12)*, pages 342–352. IEEE.

Guo, P. J., Zimmermann, T., Nagappan, N., and Murphy, B. (2010). Characterizing and predicting which bugs get fixed: an empirical study of microsoft windows. In *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering-Volume 1 (ICSE'10)*, pages 495–504.

Gupta, S. and Gupta, B. B. (2017). Detection, avoidance, and attack pattern mechanisms in modern web application vulnerabilities: present and future challenges. *International Journal of Cloud Applications and Computing (IJCAC17)*, 7(3):1–43.

Guzzo, R. A. and Shea, G. P. (1992). Group performance and intergroup relations in organizations.

Harer, J. A., Kim, L. Y., Russell, R. L., Ozdemir, O., Kosta, L. R., Rangamani, A., Hamilton, L. H., Centeno, G. I., Key, J. R., Ellingwood, P. M., et al. (2018). Automated software vulnerability detection with machine learning. *arXiv preprint arXiv:1803.04497*.

Harrell Jr, F. E., Lee, K. L., Califf, R. M., Pryor, D. B., and Rosati, R. A. (1984). Regression modelling strategies for improved prognostic prediction. *Statistics in medicine*, 3(2):143–152.

Hassan, S., Bezemer, C.-P., and Hassan, A. E. (2018). Studying bad updates of top free-to-download apps in the google play store. *IEEE Transactions on Software Engineering (TSE'18)*.

Hassan, S., Shang, W., and Hassan, A. E. (2017). An empirical study of emergency updates for top Android mobile apps. *Empirical Software Engineering (EMSE'17)*, 22(1):505–546.

Hauge, Ø., Ayala, C., and Conradi, R. (2010). Adoption of open source software in software-intensive organizations–a systematic literature review. *Information and Software Technology*, 52(11):1133–1154.

Herraiz, I., Shihab, E., Nguyen, T. H., and Hassan, A. E. (2011). Impact of installation counts on perceived quality: A case study on debian. In *2011 18th Working Conference on Reverse Engineering*, pages 219–228. IEEE.

Hindle, A., German, D. M., and Holt, R. (2008). What do large commits tell us? a taxonomical study of large commits. In *Proceedings of the 2008 international working conference on Mining software repositories (MSR'08)*, pages 99–108.

Hooimeijer, P. and Weimer, W. (2007). Modeling bug report quality. In *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering (ASE'07)*, pages 34–43.

Hora, A., Robbes, R., Valente, M. T., Anquetil, N., Etien, A., and Ducasse, S. (2018). How do developers react to api evolution? a large-scale empirical study. *Software Quality Journal*, 26(1):161–191.

Hu, H., Zhang, H., Xuan, J., and Sun, W. (2014). Effective bug triage based on historical bug-fix information. In *2014 IEEE 25th International Symposium on Software Reliability Engineering*, pages 122–132. IEEE.

Huang, Z., DAngelo, M., Miyani, D., and Lie, D. (2016). Talos: Neutralizing vulnerabilities with security workarounds for rapid response. In *2016 IEEE Symposium on Security and Privacy (SP'16)*, pages 618–635. IEEE.

Huelsenbeck, J. P. and Crandall, K. A. (1997). Phylogeny estimation and hypothesis testing using maximum likelihood. *Annual review of ecology and systematics*, 28(1):437–466.

Jansen, S., Finkelstein, A., and Brinkkemper, S. (2009). A sense of community: A research agenda for software ecosystems. In *2009 31st International Conference on Software Engineering-Companion Volume*, pages 187–190. IEEE.

Jeong, G., Kim, S., and Zimmermann, T. (2009). Improving bug triage with bug tossing graphs. In *Proceedings of the 7th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering (FSE'09)*, pages 111–120.

Jiang, Y., Adams, B., and German, D. M. (2013). Will my patch make it? and how fast? case study on the linux kernel. In *2013 10th Working Conference on Mining Software Repositories (MSR'13)*, pages 101–110. IEEE.

Jiarpakdee, J., Tantithamthavorn, C., Dam, H. K., and Grundy, J. (2020). An empirical study of model-agnostic techniques for defect prediction models. *IEEE Transactions on Software Engineering (TSE'20)*.

Joh, H. and Malaiya, Y. K. (2011). Defining and assessing quantitative security risk measures using vulnerability lifecycle and cvss metrics. In *Proceedings of the 2011 International Conference on Security and Management (SAM'11)*, volume 1, pages 10–16.

Jovanovic, N., Kruegel, C., and Kirda, E. (2006). Pixy: A static analysis tool for detecting web application vulnerabilities. In *2006 IEEE Symposium on Security and Privacy (SP'06)*, pages 6–pp. IEEE.

Just, S., Premraj, R., and Zimmermann, T. (2008). Towards the next generation of bug tracking systems. In *2008 IEEE symposium on visual languages and human-centric computing*, pages 82–85. IEEE.

Kakimoto, T., Kamei, Y., Ohira, M., and Matsumoto, K. (2006). Social network analysis on communications for knowledge collaboration in OSS communities. In *Proceedings of the International Workshop on Supporting Knowledge Collaboration in Software Development (KCSD'06)*, pages 35–41. Citeseer.

Kamei, Y., Fukushima, T., McIntosh, S., Yamashita, K., Ubayashi, N., and Hassan, A. E. (2016). Studying just-in-time defect prediction using cross-project models. *Empirical Software Engineering (EMSE'16)*, 21(5):2072–2106.

Kamei, Y., Matsumoto, S., Monden, A., Matsumoto, K.-i., Adams, B., and Hassan, A. E. (2010). Revisiting common bug prediction findings using effort-aware models.

In *Proceedings of the 26th IEEE International Conference on Software Maintenance (ICSM'10)*, pages 1–10. IEEE.

Kamei, Y., Shihab, E., Adams, B., Hassan, A. E., Mockus, A., Sinha, A., and Ubayashi, N. (2012). A large-scale empirical study of just-in-time quality assurance. *IEEE Transactions on Software Engineering (TSE'12)*, 39(6):757–773.

Kikas, R., Gousios, G., Dumas, M., and Pfahl, D. (2017). Structure and evolution of package dependency networks. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR'17)*, pages 102–112. IEEE.

Kim, S. and Whitehead Jr, E. J. (2006). How long did it take to fix bugs? In *Proceedings of the 2006 international workshop on Mining software repositories (MSR'06)*, pages 173–174.

Klinke, A. and Renn, O. (2002). A new approach to risk evaluation and management: Risk-based, precaution-based, and discourse-based strategies 1. *Risk Analysis: An International Journal*, 22(6):1071–1094.

Kraut, R. E. and Streeter, L. A. (1995). Coordination in software development. *Communications of the ACM*, 38(3):69–82.

Kula, R. G., German, D. M., Ouni, A., Ishio, T., and Inoue, K. (2018a). Do developers update their library dependencies? *Empirical Software Engineering (EMSE'18)*, 23(1):384–417.

Kula, R. G., German, D. M., Ouni, A., Ishio, T., and Inoue, K. (2018b). Do developers update their library dependencies? *Empirical Software Engineering (EMSE'18)*, 23(1):384–417.

Kwan, I., Schroter, A., and Damian, D. (2011). Does socio-technical congruence have an effect on software build success? a study of coordination in a software project. *IEEE Transactions on Software Engineering (TSE'11)*, 37(3):307–324.

Kyaw, A. K., Sioquim, F., and Joseph, J. (2015). Dictionary attack on wordpress: Security and forensic analysis. In *2015 Second International Conference on Information Security and Cyber Forensics (InfoSec)*, pages 158–164. IEEE.

Lamkanfi, A., Demeyer, S., Giger, E., and Goethals, B. (2010). Predicting the severity of a reported bug. In *Proceedings of the 7th IEEE Working Conference on Mining Software Repositories (MSR'10)*, pages 1–10. IEEE.

Lee, D., Rajbahadur, G. K., Lin, D., Sayagh, M., Bezemer, C.-P., and Hassan, A. E. (2020). An empirical study of the characteristics of popular minecraft mods. *Empirical Software Engineering (EMSE'20)*, 25(5):3396–3429.

Lee, J., Hong, S., and Oh, H. (2018). Memfix: static analysis-based repair of memory deallocation errors for c. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 95–106.

Li, F. and Paxson, V. (2017). A large-scale empirical study of security patches. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS'17)*, pages 2201–2215.

Li, J., Conradi, R., Slyngstad, O. P. N., Bunse, C., Khan, U., Torchiano, M., and Morisio, M. (2005). An empirical study on off-the-shelf component usage in industrial

projects. In *International Conference on Product Focused Software Process Improvement*, pages 54–68. Springer.

Li, Z., Zou, D., Xu, S., Jin, H., Zhu, Y., and Chen, Z. (2021). Sysevr: A framework for using deep learning to detect software vulnerabilities. *IEEE Transactions on Dependable and Secure Computing*.

Lin, D., Bezemer, C.-P., and Hassan, A. E. (2017). Studying the urgent updates of popular games on the steam platform. *Empirical Software Engineering (EMSE'17)*, 22(4):2095–2126.

Lin, J., Sayagh, M., and Hassan, A. E. (2022a). The co-evolution of the WordPress platform and its plugins. *ACM Transactions on Software Engineering and Methodology (TOSEM'22)*.

Lin, J., Zhang, H., Adams, B., and Hassan, A. E. (2022b). Upstream bug management in Linux distributions. *Empirical Software Engineering (EMSE'22)*, 27(6):1–41.

Lin, J., Zhang, H., Adams, B., and Hassan, A. E. (2023). Vulnerability management in linux distributions: An empirical study on debian and fedora. *Empirical Software Engineering*, 28(2):47.

Ling, C. X., Huang, J., and Zhang, H. (2003). Auc: a better measure than accuracy in comparing learning algorithms. In *Conference of the canadian society for computational studies of intelligence*, pages 329–341. Springer.

Liu, B., Meng, G., Zou, W., Gong, Q., Li, F., Lin, M., Sun, D., Huo, W., and Zhang, C. (2020). A large-scale empirical study on vulnerability distribution within projects

and the lessons learned. In *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE'20)*, pages 1547–1559. IEEE.

Lungu, M., Robbes, R., and Lanza, M. (2010). Recovering inter-project dependencies in software ecosystems. In *Proceedings of the IEEE/ACM international conference on Automated software engineering (ASE'10)*, pages 309–312.

Ma, W., Chen, L., Zhang, X., Feng, Y., Xu, Z., Chen, Z., Zhou, Y., and Xu, B. (2020a). Impact analysis of cross-project bugs on software ecosystems. In *Proceedings of the 42nd ACM/IEEE International Conference on Software Engineering (ICSE'20)*, pages 100–111.

Ma, W., Chen, L., Zhang, X., Feng, Y., Xu, Z., Chen, Z., Zhou, Y., and Xu, B. (2020b). Impact analysis of cross-project bugs on software ecosystems. In *Proceedings of the 42nd ACM/IEEE International Conference on Software Engineering (ICSE'20)*, pages 100–111.

Ma, W., Chen, L., Zhang, X., Zhou, Y., and Xu, B. (2017). How do developers fix cross-project correlated bugs? a case study on the Github scientific Python ecosystem. In *Proceedings of the 39th IEEE/ACM International Conference on Software Engineering (ICSE'17)*, pages 381–392. IEEE.

Machiry, A., Redini, N., Camellini, E., Kruegel, C., and Vigna, G. (2020). Spider: Enabling fast patch propagation in related software repositories. In *2020 IEEE Symposium on Security and Privacy (SP'20)*, pages 1562–1579. IEEE.

Maddila, C., Nagappan, N., Bird, C., Gousios, G., and van Deursen, A. (2021). Cone: A

concurrent edit detection tool for large-scale software development. *ACM Transactions on Software Engineering and Methodology (TOSEM'21)*, 31(2):1–26.

Malone, T. W. and Crowston, K. (1994). The interdisciplinary study of coordination. *ACM Computing Surveys (CSUR)*, 26(1):87–119.

Marks, L., Zou, Y., and Hassan, A. E. (2011). Studying the fix-time for bugs in large open source projects. In *Proceedings of the 7th International Conference on Predictive Models in Software Engineering (PROMISE'11)*, pages 1–8.

McDonnell, T., Ray, B., and Kim, M. (2013). An empirical study of api stability and adoption in the android ecosystem. In *2013 IEEE International Conference on Software Maintenance (ICSM'13)*, pages 70–79. IEEE.

McIntosh, S., Kamei, Y., Adams, B., and Hassan, A. E. (2016). An empirical study of the impact of modern code review practices on software quality. *Empirical Software Engineering (EMSE'16)*, 21(5):2146–2189.

Mendez, C., Padala, H. S., Steine-Hanson, Z., Hilderbrand, C., Horvath, A., Hill, C., Simpson, L., Patil, N., Sarma, A., and Burnett, M. (2018). Open source barriers to entry, revisited: A sociotechnical perspective. In *Proceedings of the 40th International Conference on Software Engineering (ICSE'18)*, pages 1004–1015.

Meneely, A. and Williams, L. (2011). Socio-technical developer networks: should we trust our measurements? In *Proceedings of the 33rd International Conference on Software Engineering (ICSE'11)*, pages 281–290.

Menzies, T. and Marcus, A. (2008). Automated severity assessment of software defect

reports. In *Proceedings of the 24th IEEE International Conference on Software Maintenance (ICSM'08)*, pages 346–355. IEEE.

Merilinna, J. and Matinlassi, M. (2006). State of the art and practice of opensource component integration. In *Proceedings of the 32nd EUROMICRO Conference on Software Engineering and Advanced Applications (EUROMICRO'06)*, pages 170–177. IEEE.

Mirhosseini, S. and Parnin, C. (2017). Can automated pull requests encourage software developers to upgrade out-of-date dependencies? In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE'17)*, pages 84–94. IEEE.

Mitre.org (2021). CVE. https://cve.mitre.org/. last accessed: 2021-06-02.

Mockus, A., Fielding, R. T., and Herbsleb, J. D. (2002). Two case studies of open source software development: Apache and Mozilla. *ACM Transactions on Software Engineering and Methodology (TOSEM'02)*, 11(3):309–346.

Moslehi, P., Adams, B., and Rilling, J. (2018). Feature location using crowd-based screencasts. In *Proceedings of the 15th International Conference on Mining Software Repositories*, pages 192–202.

Nakajima, A., Watanabe, T., Shioji, E., Akiyama, M., and Woo, M. (2019). A pilot study on consumer iot device vulnerability disclosure and patch release in japan and the united states. In *Proceedings of the 2019 ACM Asia Conference on Computer and Communications Security (AsiaCCS '19)*, pages 485–492.

Nappa, A., Johnson, R., Bilge, L., Caballero, J., and Dumitras, T. (2015). The attack of

the clones: A study of the impact of shared code on vulnerability patching. In *2015 IEEE symposium on security and privacy*, pages 692–708. IEEE.

Nelson, N., Brindescu, C., McKee, S., Sarma, A., and Dig, D. (2019). The life-cycle of merge conflicts: processes, barriers, and strategies. *Empirical Software Engineering (EMSE'19)*, 24(5):2863–2906.

Nguyen, H. V., Kästner, C., and Nguyen, T. N. (2014). Exploring variability-aware execution for testing plugin-based web applications. In *Proceedings of the 36th International Conference on Software Engineering (ICSE'14)*, pages 907–918.

NVD (2021). U.S. national institute of standards and technology. CVSS information. https://nvd.nist.gov/vuln-metrics/cvss. last accessed: 2021-06-02.

Ohira, M., Hassan, A. E., Osawa, N., and Matsumoto, K.-i. (2012). The impact of bug management patterns on bug fixing: A case study of eclipse projects. In *Proceedings of the 28th IEEE International Conference on Software Maintenance (ICSM'12)*, pages 264–273. IEEE.

Ohm, M., Plate, H., Sykosch, A., and Meier, M. (2020). Backstabber's knife collection: A review of open source software supply chain attacks. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA'20)*, pages 23–43. Springer.

OSS-security.org (2021). Operating system distribution security contact lists. https://oss-security.openwall.org/wiki/mailing-lists/distros. last accessed: 2021-06-02.

Ozment, A. and Schechter, S. E. (2006). Milk or wine: does software security improve with age? In *USENIX Security Symposium*, volume 6.

Panichella, S., Bavota, G., Di Penta, M., Canfora, G., and Antoniol, G. (2014). How developers' collaborations identified from different sources tell us about code changes. In *2014 IEEE International Conference on Software Maintenance and Evolution (ICSME'14)*, pages 251–260. IEEE.

Piantadosi, V., Scalabrino, S., and Oliveto, R. (2019). Fixing of security vulnerabilities in open source projects: A case study of apache http server and apache tomcat. In *2019 12th IEEE Conference on software testing, validation and verification (ICST'19)*, pages 68–78. IEEE.

Raemaekers, S., Van Deursen, A., and Visser, J. (2012). Measuring software library stability through historical version analysis. In *Proceedings of the 28th IEEE International Conference on Software Maintenance (ICSM'12)*, pages 378–387. IEEE.

Raemaekers, S., Van Deursen, A., and Visser, J. (2014). Semantic versioning versus breaking changes: A study of the maven repository. In *2014 IEEE 14th International Working Conference on Source Code Analysis and Manipulation (SCAM'14)*, pages 215–224. IEEE.

Rafique, S., Humayun, M., Hamid, B., Abbas, A., Akhtar, M., and Iqbal, K. (2015). Web application security vulnerabilities detection approaches: A systematic mapping study. In *2015 IEEE/ACIS 16th International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD'15)*, pages 1–6. IEEE.

Rajbahadur, G. K., Wang, S., Ansaldi, G., Kamei, Y., and Hassan, A. E. (2021). The impact of feature importance methods on the interpretation of defect classifiers. *IEEE Transactions on Software Engineering (TSE'21)*.

Ramsauer, R., Bulwahn, L., Lohmann, D., and Mauerer, W. (2020). The sound of silence: Mining security vulnerabilities from secret integration channels in open-source projects. In *Proceedings of the 2020 ACM SIGSAC Conference on Cloud Computing Security Workshop (CCSW'20)*, pages 147–157.

Ray, B. and Kim, M. (2012). A case study of cross-system porting in forked projects. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering (FSE'12)*, pages 1–11.

Ray, B., Kim, M., Person, S., and Rungta, N. (2013). Detecting and characterizing semantic inconsistencies in ported code. In *Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering (ASE'13)*, pages 367–377. IEEE.

Raymond, E. (1999). The cathedral and the bazaar. *Knowledge, Technology & Policy*, 12(3):23–49.

Redhat.com (2021). The hidden costs of embargoes. https://access.redhat.com/blogs/766093/posts/1976653. last accessed: 2021-06-02.

Reed, A. H. and Knight, L. V. (2010). Effect of a virtual project team environment on communication-related project risk. *International Journal of Project Management*, 28(5):422–427.

Ristov, S., Gusev, M., and Donevski, A. (2013). Openstack cloud security vulnerabilities from inside and outside. *Cloud Computing*, pages 101–107.

Robbes, R., Lungu, M., and Röthlisberger, D. (2012). How do developers react to API deprecation? the case of a smalltalk ecosystem. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering (FSE'12)*, pages 1–11.

Ruohonen, J. (2018). An empirical analysis of vulnerabilities in python packages for web applications. In *2018 9th International Workshop on Empirical Software Engineering in Practice (IWESEP'18)*, pages 25–30. IEEE.

Russell, R., Kim, L., Hamilton, L., Lazovich, T., Harer, J., Ozdemir, O., Ellingwood, P., and McConley, M. (2018). Automated vulnerability detection in source code using deep representation learning. In *2018 17th IEEE international conference on machine learning and applications (ICMLA'18)*, pages 757–762. IEEE.

Ryu, D., Choi, O., and Baik, J. (2016). Value-cognitive boosting with a support vector machine for cross-project defect prediction. *Empirical Software Engineering (EMSE'16)*, 21(1):43–71.

Sabottke, C., Suciu, O., and Dumitraș, T. (2015). Vulnerability disclosure in the age of social media: Exploiting twitter for predicting {Real-World} exploits. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 1041–1056.

Sayagh, M. and Adams, B. (2015). Multi-layer software configuration: Empirical study on wordpress. In *2015 IEEE 15th International Working Conference on Source Code Analysis and Manipulation (SCAM'15)*, pages 31–40. IEEE.

Sayagh, M., Kerzazi, N., and Adams, B. (2017). On cross-stack configuration errors. In *Proceedings of the 39th IEEE/ACM International Conference on Software Engineering (ICSE'17)*, pages 255–265. IEEE.

Schryen, G. (2009). A comprehensive and comparative analysis of the patching behavior of open source and closed source software vendors. In *2009 Fifth International Conference on IT Security Incident Management and IT Forensics (IMF'09)*, pages 153–168. IEEE.

Seaman, C. B. (1999). Qualitative methods in empirical studies of software engineering. *IEEE Transactions on software engineering (TSE'99)*, 25(4):557–572.

Seaman, C. B., Shull, F., Regardie, M., Elbert, D., Feldmann, R. L., Guo, Y., and Godfrey, S. (2008). Defect categorization: making use of a decade of widely varying historical data. In *Proceedings of the Second ACM-IEEE international symposium on Empirical software engineering and measurement (ESEM'08)*, pages 149–157.

Shahzad, M., Shafiq, M. Z., and Liu, A. X. (2012). A large scale exploratory analysis of software vulnerability life cycles. In *Proceedings of the 34th International Conference on Software Engineering (ICSE'12)*, pages 771–781. IEEE.

Shihab, E., Ihara, A., Kamei, Y., Ibrahim, W. M., Ohira, M., Adams, B., Hassan, A. E., and Matsumoto, K.-i. (2013). Studying re-opened bugs in open source software. *Empirical Software Engineering (EMSE'13)*, 18(5):1005–1042.

Shin, Y., Meneely, A., Williams, L., and Osborne, J. A. (2010). Evaluating complexity, code churn, and developer activity metrics as indicators of software vulnerabilities. *IEEE transactions on software engineering (TSE'10)*, 37(6):772–787.

Sood, A. K., Bansal, R., and Enbody, R. J. (2012). Cybercrime: Dissecting the state of underground enterprise. *IEEE internet computing*, 17(1):60–68.

Steinmacher, I., Conte, T. U., Treude, C., and Gerosa, M. A. (2016). Overcoming open source project entry barriers with a portal for newcomers. In *Proceedings of the 38th International Conference on Software Engineering (ICSE'16)*, pages 273–284.

Steinmacher, I., Wiese, I. S., Conte, T., Gerosa, M. A., and Redmiles, D. (2014). The hard life of open source software project newcomers. In *Proceedings of the 7th International Workshop on Cooperative and Human Aspects of Software Engineering (CHASE'14)*, pages 72–78.

Stock, B., Pellegrino, G., Rossow, C., Johns, M., and Backes, M. (2016). Hey, you have a problem: On the feasibility of {Large-Scale} web vulnerability notification. In *25th USENIX Security Symposium*, pages 1015–1032.

Stol, K.-J., Babar, M. A., Avgeriou, P., and Fitzgerald, B. (2011). A comparative study of challenges in integrating open source software and inner source software. *Information and Software Technology*, 53(12):1319–1336.

Storey, M.-A., Zagalsky, A., Figueira Filho, F., Singer, L., and German, D. M. (2016). How social and communication channels shape and challenge a participatory culture in software development. *IEEE Transactions on Software Engineering (TSE'16)*, 43(2):185–204.

Swanson, E. B. (1976). The dimensions of maintenance. In *Proceedings of the 2nd international conference on Software engineering*, pages 492–497.

Tabassum, S., Minku, L. L., Feng, D., Cabral, G. G., and Song, L. (2020). An investigation of cross-project learning in online just-in-time software defect prediction. In *Proceedings of 42nd IEEE/ACM International Conference on Software Engineering (ICSE'20)*, pages 554–565. IEEE.

Telang, R. and Wattal, S. (2005). Impact of software vulnerability announcements on the market value of software vendors-an empirical investigation. *Available at SSRN 677427*.

Thongtanunam, P. and Hassan, A. E. (2020). Review dynamics and their impact on software quality. *IEEE Transactions on Software Engineering (TSE'20)*.

Thongtanunam, P., McIntosh, S., Hassan, A. E., and Iida, H. (2016). Revisiting code ownership and its relationship with software quality in the scope of modern code review. In *Proceedings of the 38th international conference on software engineering (ICSE'16)*, pages 1039–1050.

Thummalapenta, S., Cerulo, L., Aversano, L., and Di Penta, M. (2010). An empirical study on the maintenance of source code clones. *Empirical Software Engineering (EMSE'10)*, 15(1):1–34.

Tourani, P. and Adams, B. (2016). The impact of human discussions on just-in-time quality assurance: An empirical study on OpenStack and Eclipse. In *Proceedings of the 23rd IEEE International Conference on Software Analysis, Evolution, and Reengineering (SANER'16)*, volume 1, pages 189–200. IEEE.

Ulan, M., Löwe, W., Ericsson, M., and Wingkvist, A. (2021). Weighted software metrics

aggregation and its application to defect prediction. *Empirical Software Engineering (EMSE'21)*, 26(5):1–34.

Van Den Berk, I., Jansen, S., and Luinenburg, L. (2010). Software ecosystems: a software ecosystem strategy assessment model. In *Proceedings of the Fourth European Conference on Software Architecture: Companion Volume*, pages 127–134.

Van der Linden, F. J., Schmid, K., and Rommes, E. (2007). *Software product lines in action: the best industrial practice in product line engineering*. Springer Science & Business Media.

W3techs.com (2020). Usage of content management systems. https://w3techs.com/technologies/overview/content_management. last accessed: 2020-05-15.

Wagstrom, P. and Datta, S. (2014). Does latitude hurt while longitude kills? geographical and temporal separation in a large scale software development project. In *Proceedings of the 36th International Conference on Software Engineering (ICSE'14)*, pages 199–210.

Wang, S. and Nagappan, N. (2019). Characterizing and understanding software developer networks in security development. *arXiv preprint arXiv:1907.12141*.

Wang, X., Sun, K., Batcheller, A., and Jajodia, S. (2019). Detecting "0-day" vulnerability: An empirical study of secret security patch in OSS. In *2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'19)*, pages 485–492. IEEE.

Wei, L., Liu, Y., and Cheung, S.-C. (2016). Taming android fragmentation: Characterizing and detecting compatibility issues for android apps. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering (ASE'16)*, pages 226–237.

Weimer, W. (2006). Patches as better bug reports. In *Proceedings of the 5th International Conference on Generative Programming and Component Engineering (GPCE'06)*, pages 181–190.

Weiss, C., Premraj, R., Zimmermann, T., and Zeller, A. (2007). How long will it take to fix this bug? In *Fourth International Workshop on Mining Software Repositories (MSR'07)*, pages 1–1. IEEE.

Wittern, E., Suter, P., and Rajagopalan, S. (2016). A look at the dynamics of the javascript package ecosystem. In *Proceedings of the 13th International Conference on Mining Software Repositories (MSR'16)*, pages 351–361.

Wordpress.org (2020a). Link library. https://wordpress.org/plugins/link-library/#developers. last accessed: 2020-05-15.

Wordpress.org (2020b). Page builder by siteorigin. https://wordpress.org/plugins/siteorigin-panels/#developers. last accessed: 2020-05-15.

Wordpress.org (2020c). Photo gallery by 10web – mobile-friendly image gallery. https://wordpress.org/plugins/photo-gallery/#developers. last accessed: 2020-05-15.

Wordpress.org (2020d). Premium addons for elementor. https://wordpress.

org/plugins/premium-addons-for-elementor/#developers. last accessed: 2020-05-15.

Wordpress.org (2020e). Simple membership. https://wordpress.org/plugins/simple-membership/#developers. last accessed: 2020-05-15.

Wordpress.org (2020f). Sunshine photo cart. https://wordpress.org/plugins/sunshine-photo-cart/#developers. last accessed: 2020-05-15.

Wu, D., Gao, D., Cheng, E. K., Cao, Y., Jiang, J., and Deng, R. H. (2019). Towards understanding android system vulnerabilities: techniques and insights. In *Proceedings of the 2019 ACM Asia Conference on Computer and Communications Security (AsiaCCS '19)*, pages 295–306.

Xavier, L., Brito, A., Hora, A., and Valente, M. T. (2017). Historical and impact analysis of API breaking changes: A large-scale study. In *Proceedings of the 24th IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER'17)*, pages 138–147. IEEE.

Xia, X., Lo, D., Wen, M., Shihab, E., and Zhou, B. (2014). An empirical study of bug report field reassignment. In *2014 Software Evolution Week-IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering (CSMR-WCRE'14)*, pages 174–183. IEEE.

Xuan, J., Jiang, H., Ren, Z., and Zou, W. (2012). Developer prioritization in bug repositories. In *Proceedings of the 34th International Conference on Software Engineering (ICSE'12)*, pages 25–35. IEEE.

Yilek, S., Rescorla, E., Shacham, H., Enright, B., and Savage, S. (2009). When private keys are public: Results from the 2008 Debian OpenSSL vulnerability. In *Proceedings of the 9th ACM Conference on Internet Measurement (IMC'09)*, pages 15–27.

Yin, Z., Yuan, D., Zhou, Y., Pasupathy, S., and Bairavasundaram, L. (2011). How do fixes become bugs? In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering (FSE'11)*, pages 26–36.

Zaman, S., Adams, B., and Hassan, A. E. (2011). Security versus performance bugs: a case study on firefox. In *Proceedings of the 8th working conference on mining software repositories (MSR'11)*, pages 93–102.

Zhang, F., Khomh, F., Zou, Y., and Hassan, A. E. (2012). An empirical study on factors impacting bug fixing time. In *2012 19th Working Conference on Reverse Engineering*, pages 225–234. IEEE.

Zhang, H., Gong, L., and Versteeg, S. (2013). Predicting bug-fixing time: an empirical study of commercial software projects. In *Proceedings of the 35th International Conference on Software Engineering (ICSE'13)*, pages 1042–1051. IEEE.

Zhang, H., Wang, S., Li, H., Chen, T.-H. P., and Hassan, A. E. (2021). A study of C/C++ code weaknesses on stack overflow. *IEEE Transactions on Software Engineering (TSE'21)*.

Zhang, Y., Yu, Y., Wang, H., Vasilescu, B., and Filkov, V. (2018). Within-ecosystem issue linking: a large-scale study of rails. In *Proceedings of the 7th International Workshop on Software Mining*, pages 12–19.

Zhao, M., Grossklags, J., and Liu, P. (2015). An empirical study of web vulnerability discovery ecosystems. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 1105–1117.

Zhou, B., Neamtiu, I., and Gupta, R. (2015). A cross-platform analysis of bugs and bug-fixing in open source projects: Desktop vs. android vs. ios. In *Proceedings of the 19th International Conference on Evaluation and Assessment in Software Engineering (EASE'15)*, pages 1–10.

Zimmermann, T., Premraj, R., Bettenburg, N., Just, S., Schroter, A., and Weiss, C. (2010). What makes a good bug report? *IEEE Transactions on Software Engineering (TSE'10)*, 36(5):618–643.