An industrial study on the differences between pre-release and post-release bugs

Renaud Rwemalika*, Marinos Kintis*, Mike Papadakis*, Yves Le Traon* and Pierre Lorrach†
*SnT, University of Luxembourg, Luxembourg
†BGL BNP Paribas, Luxembourg
*{firstname.surname}@uni.lu
†pierre.lorrach@bgl.lu

***Abstract*—Software bugs constitute a frequent and common issue of software development. To deal with this problem, modern software development methodologies introduce dedicated quality assurance procedures. At the same time researchers aim at developing techniques capable of supporting the early discovery and fix of bugs. One important factor that guides such research attempts is the characteristics of software bugs and bug fixes.**

In this paper, we present an industrial study on the characteristics and differences between pre-release bugs, i.e. bugs detected during software development, and post-release bugs, i.e. bugs that escaped to production. Understanding such differences is of paramount importance as it will improve our understanding on the testing and debugging support that practitioners require from the research community, on the validity of the assumptions of several research techniques, and, most importantly, on the reasons why bugs escape to production.

To this end, we analyze 37 industrial projects from BGL BNP Paribas and document the differences between pre-release bugs and post-release bugs. Our findings suggest that post-release bugs are more complex to fix, requiring developers to modify several source code files, written in different programming languages, and configuration files, as well. We also find that approximately 82% of the post-release bugs involve code additions and can be characterized as 'omission' bugs. Finally, we conclude the paper with a discussion on the implications of our study and provide guidance to future research directions.

I. Introduction

Issues caused by software defects are a common source of business failures and economic losses. According to the Software Fail Watch of Tricentis [1] in 2017, software bugs resulted in $1.7 trillion of industrial revenue losses and, more importantly, the number of bugs reported in 2017 increased by 10 percent compared to 2016. These numbers highlight the importance of a good understanding on the nature of software bugs and their root causes. To answer these questions, additional studies are needed to better deal with bugs and reduce their impact.

Source code analysis techniques are typically developed and evaluated using some bug instances [2], which should reflect specific characteristics and assumptions around the targeted bugs. Thus, bug finding and removal techniques, such as software testing [3], static analysis [4], fault localization [5] and program repair [6], are developed according to the characteristics, nature and fixes of the used bug datasets, e.g. SIR [7], Defects4j [8] and Bugs.jar [9]. This is a good first step towards developing feasible and effective techniques. However, our perception, understanding and assessment of these techniques is strongly connected to the characteristics of the bugs involved in these datasets.

Moreover, researchers in the field of code analysis need to position and evaluate their work with respect to specific tasks, targeted scenarios and working assumptions. For example, research on advanced software testing techniques should focus on post-release bugs (bugs that escaped the basic testing process), which are hard to reveal, instead of pre-release bugs. Similarly, fault localization and bug repair techniques should focus on in-field bugs, where partial information is available. Another parameter, often ignored, is that fixing post release bugs is much more harmful to companies, exposing failures to their customers. Therefore, researchers need detailed information on the properties of the selected bugs according the techniques under consideration.

To understand the bug characteristics and their consequences on software testing and debugging techniques, we perform an extensive study on pre- and post-release bug characteristics of our industrial partner, BGL BNP Paribas, a leading banking company. We focus on 'critical' systems, developed in Java. These systems have been audited and tested using both unit and system (end-to-end) test practices. Our aim is to understand the nature of common kinds of real software issues by performing a fine-grained analysis on the recorded bugs and their patches.

In contrast to the majority of previous research that is based on Open Source projects, our study focuses on real industrial systems. We deemed this endeavor as important since the software development workflow and nature of the Open Source projects does not necessarily offer a faithful representation of the bug occurrences in large companies. Indeed, industrial code development often differs in architecture (e.g. system composed of a multitude of services), process (quality assurance (QA) teams separated from development teams) and people involved. Moreover, in most Open Source projects it is hard to illustrate the concept of pre- and post-release bugs in a meaningful way.

We extract and study both quantitatively, using source code metrics, and qualitatively, code context, the properties of the industrial pre- and post-release bug patches. Our industrial partner has established quality assurance teams and procedures, making the distinction between pre- and post-release bugs meaningful. As such, our analysis can help interpreting the feasibility of existing methods/studies, judging the representativeness of dataset used in previous work, help positioning and choosing appropriate pre- or post-release bug data and increases the general understanding of the industrial software issues.

Perhaps the most interesting result from our study is that we find evidence that most of the bug patches we analyzed, especially the post-release ones, involve a large number (approximately 82%) of additions. This finding suggests that the related bugs fall in the category of the so-called 'omission' bugs [10]. This is particularly important for software testing researchers since omission bugs are a limitation of the widely used and researched code-based software testing techniques (e.g., code coverage) [11]. As code-based testing is driven by the existing code, targeting the coverage of codebases, it is hard to reveal issues related to code that is not there [11].

Another finding regards the scope of the changes required to fix post-release bugs. While fixes requiring complex changes that spread across multiple files exist, the majority of the bugs are fixed locally (with changes applied to the same unit) and in conditional statements. This is good news for testing research and specifically unit testing, as it provides evidence that unit testing could be adequate for targeting such post-release bugs (bugs causes spread across different units being harder to triggered by unit testing).

One other interesting finding regards the existence of configuration bugs and the relative differences between pre- and post-release bugs. We find that 45.97% and 66.69% of the pre- and post-release bugs require changes on configuration files. We find that post-release bugs require chunks of changes twice as big as pre-release ones and these changes mainly involve control flow modifications (while the pre-release ones involve interface changes). Regarding the locations of fixes, we found that both pre- and post-release locations are similar indicating that the difference are mainly on the nature of bugs than in their location.

Overall, more research is needed in this important area to fully understand this fundamental aspect of software bugs, and we certainly do not claim to have completely answered all questions in this paper. We do, however, believe that our findings significantly improve the understanding of bugs nature, the structural differences between pre- and post-release bugs, and the areas where source code analysis techniques should focus on.

Our primary contribution is to raise the awareness of the research community for the need to distinguish and control between pre- and post-release bugs, and to present the results of an industrial empirical study demonstrating significant differences. The most important finding from this control study is the evidence that almost all post-release bugs are 'local' and the apparent existence of the so-called 'omission' bugs. These findings suggesting that future research should focus on unit-based (local) analysis techniques targeting this particular class of bugs, which is, as we discussed, fundamentally different from the rest bug types.

II. Background

Software development usually adopts dedicated procedures, such as code reviews, testing and verification, to minimize and prevent software issues. To successfully develop and promote research on these fields we need to understand the nature of software bugs. To do so, we focus on code-based software failures that are the most costly and are responsible for the majority of the industrial post release problems. But, what exactly is a software bug?

According to the definition provided by the IEEE Standard Classification for Software Anomalies [12], a defect is an imperfection or deficiency in a work product where that work product does not meet its requirements or specification and needs to be either repaired or replaced. A failure, hereinafter referred to as as bug, is defined as an event in which a system or system component does not perform a required function within specified limits. Bugs are usually caused by the incorrect understanding of the software requirements, by missing requirements, changed requirements, by incorrect system or environment configuration, by the complexity of the software forgotten cases etc. In most cases, lack of appropriate testing procedures, or the semantic size of the bugs makes bugs appearing in production systems.

To reduce bugs instances, it is important to understand the categories of defect types, their semantics and the activities with respect to the different software development stages [13]. To deal with this issue, Orthogonal Defect Classification identifies 8 types of bugs [13], i.e., Function, Interface, Checking, Assignment, Timing/Serialization, Build/Package/Merge, Documentation and Algorithm, determines the defect type and defect triggering condition distributions with the ultimate goal of measuring testing and verification processes. Along these lines, we perform an extended analysis of the properties involved in industrial pre- and post-release bug patches.

III. Research Questions

Our aim is to investigate the properties of pre- and post-release bugs. To do so, we resort on analyzing the bug fixes and their context (code surrounding the points modified to fix bugs). This merely means that we observe what needs to be changed to fix the encountered issues. This analysis intends to capture the semantic properties of the bugs and shape their syntactic profile [13], [14]. Therefore, we start our study by investigating:

RQ1: *How complex are industrial pre-release bugs?*

We consider as pre-release bugs those that were detected during the quality assurance process. We deem this analysis important, as it reveals the nature of the bugs that are detected by the current procedures. Having investigated pre-release bugs we turn on to post-release ones. Thus, we ask:

RQ2: *How complex are industrial post-release bugs?*

Unlike pre-release bugs, post-release bugs are the ones that escape the quality assurance process and are released to production. They are typically reported by customers observing a faulty behavior of the system.

We consider post-release bug analysis important as it reveals the nature of the most important bugs. Thus, understanding the characteristics of these bugs complement our knowledge and sheds light on the target of future research. Of course, studying the relative differences between pre- and post-release bugs, can make such an analysis insightful and helpful. Hence, we ask:

RQ3: *What are the similarities and differences between pre- and post-release bugs?*

Pre- and post-release bugs are two important and interesting categories; pre-release bugs exhibit the effectiveness of our testing practices whereas post-release ones their weaknesses. Where RQ1 and RQ2 focused on separately discussing the characteristics of these categories, RQ3 puts them in perspective and analyzes their similarities and differences. Similarities between them show the common bug characteristics that software testing techniques can exploit to target both bug categories, whereas their differences shows how testing techniques need to adapt to target post-release bugs more effectively.

The above analysis shapes the syntactic profile of bug fixes. This analysis can reveal semantic properties of the involved bugs and can guide the development and improvement of test techniques [2]. However, it does not evaluate the ability of existing static analysis tools to detect them. Static analysis forms the common basis of most of existing code analysis techniques. Its understanding and its effectiveness in both the cases of pre- and post-release bugs is particularly important. Therefore, we investigate:

RQ4: *How effective are static analysis tools in detecting pre- and post-release bugs?*

Finally, we conclude our analysis with an investigation of the effectiveness of static analysis techniques to detect pre- and post-release bugs. Static analysis could be one of the first lines of defense against bugs. This research question studies whether there is a difference between the effectiveness of such tools based on the category of bugs studied.

IV. Methodology

A. Dataset

In our study, we mine and analyze bug fixes of 37 Java projects developed by our partner BGL BNP Paribas, from which we isolated 3623 pre-release bug fixes and 250 post-release bug fixes. Pre-release bug fixes are defined as bug fixes addressing a bug found by the quality assurance process employed at the company during software development while post-release bug fixes refer to bugs that escaped the quality control process and were released to production. Those bugs are typically reported by customers and recorded in a dedicated, issue tracking system where they are assigned a unique ID, a description and a status.

The projects under study are components of the software suite, Service Oriented Architecture, developed by our partner to support their business activities. Indeed, as it is typically the case in large companies, different teams work on different software projects/services and each service communicates with others in an orchestrated environment that allows different services to be developed using different technologies and follow a different development pace.

Our partner has several quality assurance procedures in place to verify the correctness of its software. To this end, before a new functionality is released to production it has to meet several quality requirements. In a first phase, quality gates, defined in SonarQube, ensure that the test coverage of unit and integration tests is acceptable and that all tests pass. Furthermore, SonarQube produces a report on the number of vulnerabilities and potential defects relying on a static analysis of the source code. Next, the program is deployed in a QA environment where the business requirements are tested using manual and automated acceptance (end-to-end) tests. Once all quality requirements have been met, the system will be release to production. Such a thorough process explains why post-release bug fixes are so few, accounting for only 0.06% of our entire dataset.

The bug fixes presented in this work were mined using the version control system used at BGL BNP Paribas, namely Git. We systematically mine each commit and extract the relevant commits following the method presented in the next subsection. The outcome of this step consists in two sets of bug fixes, namely pre-release bug fixes and post-release bug-fixes. Each data point is composed of the buggy version of the program, the fixed version of the program and the *changeset* (bug fix) between the two versions.

B. Bug fix Identification

a) Post-release bugs: When a post-release bug is reported by a customer, an incident report is opened in a dedicated issue tracking system and assigned a unique ID. By policy, every time a post-release bug is fixed, its ID from the issue tracking system must be incorporated in the commit(s) message and these commits must contain only changes related to that particular bug. Leveraging on this practice, to collect the post-release bug fixes, we collect all the commits containing an ID from the issue tracker in their commit message and flag them as post-release bug fix[1]. We extract from this process 250 post-release bug fixes.

b) Pre-release bugs: As is the case at many companies, documenting and reporting pre-release bugs is not standardized at our partner. To circumvent this problem, we used the method introduced by Mockus and Votta [15] to isolate pre-release bug fixes. The approach consists in isolating specific keywords in the commit messages, potentially giving information about some bug fixing activity. In this work, we search for the following keywords in the commit messages: "*fix(es—ed)*", "*repair(ed)*", "*defect(s)*" and their French equivalents. From this set, we exclude all the post-release commits. We end up with 3623 pre-release bug fixes.

Finally, because our work targets bug fixes, for both sets, we remove all changes relative to testing. To do so, any changes in files containing the word *Test* in their path or their name are removed from from the dataset.

[1]The exact period analyzed is not reported for confidentiality reasons.

C. Bug fix analysis

For each bug fix described in the previous section, we extract the triplet:

- **buggy version**: Code of the version containing the bug
- **fixed version**: Code of the version where the bug is fixed
- **changeset**: The set of differences between the buggy version and the fixed version

In the remainder of this section, we describe the steps we followed in order to build a profile for the bug fixes.

1) Files Modified: While most previous open source bug benchmarks, such as Defects4J [16], focus solely on bug fixes performed on Java files, bug fixes are not restricted to such files. Industrial projects can be written in different programming languages and include various types of files, e.g. configuration files. Thus, a bug fix can affect many types of files. In order to understand which files are affected in our study, we extract the extensions of the files changed in each *changeset*. Relying on the file extension, we define the following categories:

- **Java**: Files with the extension *java*. Because the analysis focuses on Java projects we consider Java files separately than the other type of source files.
- **Sources**: Files containing source code written in other programming languages such as JavaScript (*js*), TypeScript (*ts*) or even files used by templating framework such as FreeMaker Java Template Engine (*ftl*) or JavaServer Pages (*jsp*).
- **Configuration**: In Java projects, typical configuration files are the Project Object Model files for Maven projects (*pom.xml*) and Gradle files (*gradle*) in projects built using Gradle.
- **Others**: Sometimes, other type of files are modified that are not directly related to the program such as *.ignore* files used by Git or Markdown (*md*) files used for documentation. This type of edits usually is not directly related to the bug fix and can be considered as noise.

In the remaining of our discussion, we restrict our analysis solely to Java files. While it would be interesting to have a multi-language analysis, tools and techniques to analyze code focus typically on a single language. Because the tooling and theoretical background does not allow us to conduct such a study, we focus solely on Java code. However, as our results suggest, this is a valid direction to move forward, since changes in other languages remain marginal.

2) Patch Size: The size of a patch remains one of the easiest way to assess its complexity. Using the *changeset*, we compute how many lines of code are added, deleted and modified. Since the *changeset* obtained directly from *diff* tools does not support modifications, we consider a modification as a line or group of lines added, directly followed by deletion(s), or vice versa. In this way, we avoid overestimating the number of edits. For instance consider Listing 1, a naïve way to count the number of edits is to say that we have 2 lines added and 2 lines removed, which would lead to 4 lines edited. In our work, we consider this pattern as 2 lines modified[2].

Listing 1. Patch fixing bug ClOSURE-19 from Defects4J

```
Node parent = n.getParent();
-if (parent.isVar()) {
- if (n.hasChildren() && n.getFirstChild().isQualifiedName()) {
+if (parent.isVar() &&
+ n.hasChildren() && n.getFirstChild().isQualifiedName()) {
String name = n.getString();
```

3) Patch dispersion: Wang *et al.* [17] observe that more than half of real world bugs are fixed when multiple program locations are edited together. In this work, we define as an entity a level of localization that can be observed in the code. It can be either a class, a method or a chunk. We define a chunk as a continuous sequence of changes, consisting of the combination of addition, modification and deletion of lines. The number of chunks is computed by counting the number of continuous blocks of changes in the *changeset*, i.e. edits affecting consecutive lines in a file.

To compute the number of classes and methods changes, we rely on *Change Distiller* [18]. *Change Distiller* first extracts the abstract syntax tree (AST) of both versions (buggy version and fixed version) of the code and using a tree differencing algorithm described by Chawathe *et al.* [19], the tool computes the tree edit operations, which are the operations required to go from the AST of the buggy version to the one of the fixed version. The tree edit operations coupled with the information contained in both ASTs are used to classify the source code changes. Leveraging on the data structure generated by the tool, we extract for each bug fix the unique methods and classes changed by the patch.

4) Change Pattern: While the preceding points focused on coarse-grained analysis, this section provides a fine-grained analysis of the changes observed between the buggy and the fixed version at the level of the statement and declaration change. To this end, once again we use *Change Distiller* to extract 48 change patterns from the Java code edits. For both the buggy and the fixed version, *Change Distiller* generates an AST that are fed to their change distilling algorithm. The algorithm performs a fine-grained changes extraction which relies on the tree edits presented in the previous section and the information contained in the ASTs.

The output of the change distilling algorithm is all the changes between two versions distributed among 48 categories. In order to provide a higher level vision of the changes, we group them in 6 categories similar to what Gall *et al.* [20] present in their work. We describe those categories in Table I.

5) Static Analysis: While the other metrics in this work focus on the structure of the bugs, static analysis offers a "behavioral" analysis on the bugs. Typically, static analysis can detect defects that can be classified as functional and maintainability issues, each of them being subdivided into

[2]For confidentiality reasons all code samples are taken from Defects4J and not from our dataset.

TABLE I
CHANGE TYPES

Change Type	Explanation
Conditions	Changes impacting the execution flow of the program.
Statements	Changes impacting direct calls and the ordering of statements.
Comments	Edits of the comments or the JavaDoc.
Interfaces	Edits of the interface of a class or the signature of a method.
Parameters	Modifications of the parameters and attributes of a method.
Return types	Addition, deletion or modification of the type in a return statement.

Algorithm 1 Collecting violations

```
Input: $V_{bv} \in$ violations buggy version
Input: $V_{fv} \in$ violations fixed version
Output: violations addressed by bug fix: $V_{final}$
1: $V_{final} \leftarrow \emptyset$
2: for each $v_{bv} \in V_{bv}$ do
3:     if $v_{bv} \notin V_{fv}$ then
4:         $V_{final} \leftarrow v_{bv}$
5:     end if
6: end for each
```

categories to produce specific warnings [21]. In this work, we use the term violation to denote the potential defects spotted by static analysis.

To perform our analysis, we use SpotBugs, which is the successor of FindBugs [22], a state-of-the-art Java static analysis tool often adopted by Java projects. On the official website of the project [23], we can see the following categories of violations supported by the tool:

- **Bad Practice**: Violation of good practices, also known as code smells. While these categories usually do not detect faults, they point to code improvements that can lead to the avoidance of bugs.
- **Correctness**: Apparent coding mistake which leads to unexpected behavior of the program, e.g. the presence of an infinite loop.
- **Internationalization** Flaws present typically in string encoding, resulting in improper string conversions.
- **Malicious code vulnerability** Code vulnerable to attacks from untrusted third parties.
- **Multithread correctness** Flaws leading to improper behavior during the synchronization of threads, locks and volatiles.
- **Performance** Practices leading to inefficient code execution.
- **Security** Flaws in the consumption of input generated from outside parties that can potentially create remotely exploitable security vulnerabilities.

To collect violations addressed by bug fixes, we run SpotBugs on the buggy version and the fixed version of each bug fix. Having both sets, violations from the buggy version, V_{bv}, and from the fixed version, V_{fv}, we exploit the difference to determine which ones were addressed by the bug fix. To do so, for each violation collected from the buggy version, v_{bv}, we check if it is still in the fixed version. In the case it is not, we tag the violation as addressed by the bug fix and include it in our result set, V_{final}, as shown in Algorithm 1. Given the resource-intensive nature of the process, we focus our analysis on 5 projects randomly selected from the 37 projects used for the study.

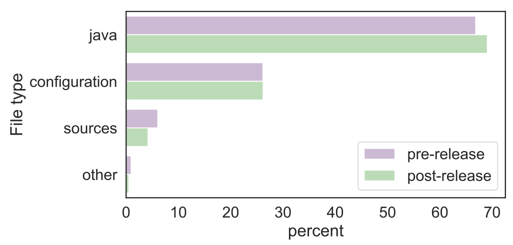


Fig. 1. Extensions of files edited in pre- and post-release patches.

V. EMPIRICAL RESULTS

The following sections answer the four research questions formulated in Section III. Sections V-A, V-B, V-C and V-D build a profile of the patches in order to assess their complexity. The profile of the patches allows to address RQ1, RQ2 and RQ3. Section V-E provides answers to RQ4.

A. File edit

Even though all the projects analyzed in this work are Java projects, other types of files might be modified while fixing bugs. In this subsection, we present results on which types of files are affected by the bug fixes, using the categories described in Section IV-C1: Java, Sources, Configuration and Others.

Figure 1 shows the proportion of files edited across all the pre- or post-release bug fixes. Not surprisingly Java files represent the large majority of file edits consisting of 66.84% and 69.09% of the total amount of edited files for the pre- and post-release bug fixes respectively. In the second place we find configuration files with 26.15% and 26.20% for pre- and post-release bug fixes respectively.

In terms of type of files edited, we see that both pre- and post-release bug fixes present the same trends. The majority of edits are performed to Java files followed by project configuration files.

Finally, we identified two other categories: *Sources*, which includes all the sources files which are not written in Java and *Others* which are file types that are neither code nor configuration. 6.04% for pre-release and 4.19% for post-release of the files edited are *Sources* files, code not written in Java. This is explained by the presence of web services serving JavaScript, CSS and HTML code to the client. The *others* category includes documentation files, CSV files or files used by Git. Not surprisingly, this category represents less that 1% of files edits both cases (0.95% in pre-release and 0.53% in post-release).

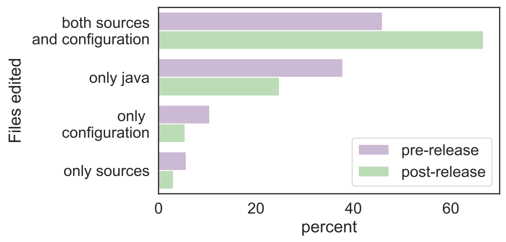


Fig. 2. Type of files edited per pre- and post-release patch.

While Figure 1 shows the proportion of file edits over all the patches, Figure 2 presents the file edits distribution per bug fix. We can see that in the case of post-release patches the proportion of edits occurring both in the source files and configuration files (66.69%) is significantly larger than in the case of pre-release patches (45.97%). Both values remains nonetheless quite high and are in both cases the most represented category. This result highlights the importance of considering modifications to configuration files as a potential fix edit when analyzing patches, creating bug datasets and in debugging research. These findings corroborate the results of Yin *et al.* [24] who show that misconfiguration issues are prevalent in both open-source and commercial projects.

Interestingly, patches containing only Java files represent only of 37.83% in the case of pre-release patches and 24.80% for post-release. This suggests that the state-of-the-art tools focusing solely on Java source files would have only been successful, in the best case, in about a quarter to a third of the bug fixes observed at BGL BNP Paribas.

Developers at our partner did not modify any source files in 10.51% of the pre-release bug fixes and in 5.43% for the post-release patches. These results are in accordance to previous results and corroborate previous findings. For instance, Zhong *et al.* [25] reports similar results, showing that 10% of the reported bugs were fixed without modifying any sources files.

In the rest of our discussion, we focus solely on Java files (66.84% and 69.09% of the total amount of edited files for the pre- and post-release bug fixes respectively) allowing to us to put our findings in perspective with existing work from the literature.

Bug fixes are most of the time composed of a combination of sources files and configuration files. Only 37.83% of the patches purely composed of Java files edits in pre-release. This value drops to 24.80% for post-release patches.

B. Patch Size

Figure 3 shows the number of added, modified and removed lines in Java files for each bug fix. Contrary to popular believe, the prevalent edit action in bug fixing is not modifying, but adding lines. Indeed, we can observe that the main edit action in both pre- and post-release patches is line additions with median values of 2 and 9 lines respectively, followed by modifications, with a median value of 2 and 4 and finally

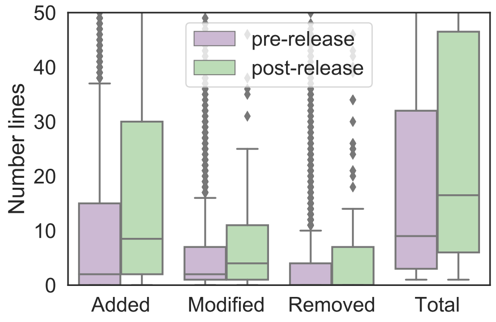


Fig. 3. Number of lines edited per pre- and post-release patches.

TABLE II
COMPARISON OF NUMBER OF LINES EDITED FOR PRE- AND POST-RELEASE DISTRIBUTIONS

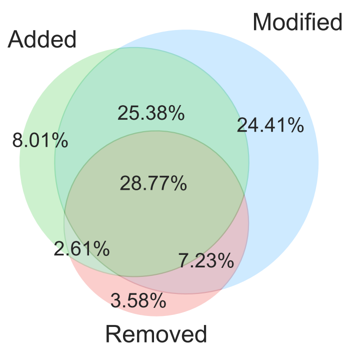
	Added	**Modified**	**Removed**	**Total**
p**-value Mann-Whitney U**	0.000	0.001	0.261	0.000
$\hat{\mathbf{A}}_{12}$	0.62	0.56	0.51	0.60

deletions of lines with a median value of 0 in both cases. Lastly, we observe that the median values for the total number of line changes is 8.5 in the case of pre-release patches and 16.5 in the case of post-release ones.

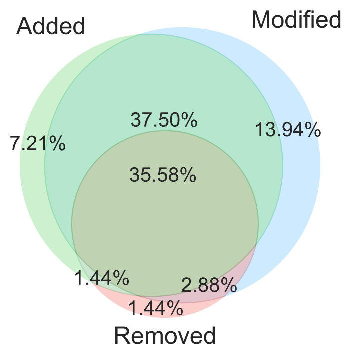
To further validate our findings, we perform inferential statistical analysis on our results. Table III presents the p-values for the Mann–Whitney U test and the effect size measures $\hat{A}_{12}$. We see that for the number of added lines and total number of lines edited, H_0 is rejected, therefore the population are different with a medium size effect ($\hat{A}_{12} > 0.56$).

Figure 4 displays the proportion of patches containing at least one added, removed or modified line or a combination of them. We can observe that in the case of pre-release patches, the number of patches containing only modified lines (24.41%) is significantly larger than for the post-release patches (13.94%). Furthermore we can observe that 64.77% of the pre-release patches and 81.73% of the post-release patches contain at least one line addition. This shows that the majority of the patches involve adding missing behaviors to the program.

These findings lead to two important observations. First, the prevalence of additions suggest that there were several lines of code missing in the buggy code, i.e. omission bugs were present. Omission bugs are an important category of bugs, which automated coverage-based test generation techniques cannot target – we cannot cover what is not there. Our results bring to light the prevalence of such bugs and suggest that research solutions should focus on this category. Secondly, our findings indicate that post-release patches are significantly larger than pre-release patches, requiring considerably more additions and modifications of lines. Thus, debugging techniques targeting limited-size patches or program locations could be ineffective in targeting post-release bugs.



(a) Pre-release patches



(b) Post-release patches

Fig. 4. Proportions of patches containing added, removed or modified lines.

TABLE III
COMPARISON OF DISPERSION FOR PRE- AND POST-RELEASE DISTRIBUTIONS

	Chunks	**Methods**	**Classes**	**Files**
p**-value Mann-Whitney U**	0.000	0.066	0.008	0.203
$\hat{\mathbf{A}}_{12}$	0.58	0.53	0.55	0.52

Post-release patches are larger with a median of 16.5 lines edited than pre-release patches with a median of only 8.5 lines edited. In both cases, the prevalent type of edits is line addition with 81.73% for post-release patches and 64.77% for pre-release patches.

C. Patch dispersion

As shown in Figure 5, the bug fixes are usually localized to a single method for a single class in a single file. However, the median values for the number of edited chunks is higher in the case of post-release, with a median value of 4 chunks than in pre-release patches with a median value of 2 chunks. In the case of pre-release patches, methods and classes have a lower quartile equal to zero which means that at least 25% of the issues didn't involve the modification of a class or a method.

Using the results presented in Table III, we observe that only the number of chunks has a medium effect size and rejects H_0 for the Mann–Whitney U test, suggesting that the distribution are different. The other results show a small effect size and weaker results for the Mann–Whitney U test.

The results suggest that in general, bug fixes are rather localized for both pre- and post-release but the post-release patches are more dispersed than the pre-release ones in terms of number of chunk changes.

67.54% of the pre-release bug fixes and 62.98% in post-release contain changes affecting at most one method in one class. Such focused changes might imply that the bug they fix could have been caught at the unit test level.

Putting those results in perspective with the ones presented by Sobreira *et al.* [14] on Defects4J, we observe a greater

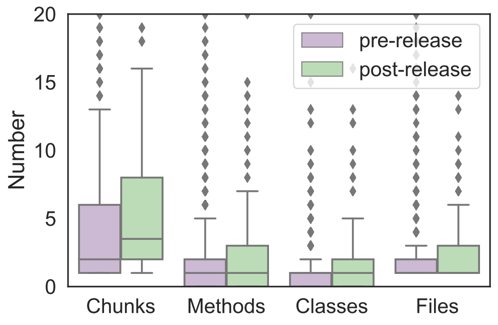


Fig. 5. Dispersion of pre- and post-release patches.

dispersion. Indeed, Defects4J changes are more localized, especially in term of chunks. However, in both pre- and post-release bug fixes as well as in Defects4J, we observe medians values for number files, classes and methods of 1. Finally, looking at the variance of the results, we see that pre-release bug fixes follow the same trend as in Defects4J, while post-release bug fixes have a great variance.

While fixes requiring changes that spread across multiple files, classes and methods exists, the majority of the bugs are fixed locally, typically, in the same method. Unit testing usually target the method level as a unit, unit tests could be adequate to target bugs both in pre- and post-release, being able to trigger them.

While post-release patches are more dispersed than pre-release patches, they both are generally localized to a single method.

D. Edit pattern

Using the tool *Change Distiller*, we extracted the edit actions from the bug fixes. Figure 6 shows the results of the frequency of each category defined in Table I. We can observe that the dominating category is "Statements" for both pre- and post-release patches with 64.10% and 53.25% respectively.

In the case of pre-release bug fixes, the second most important category represent the structural changes, referred

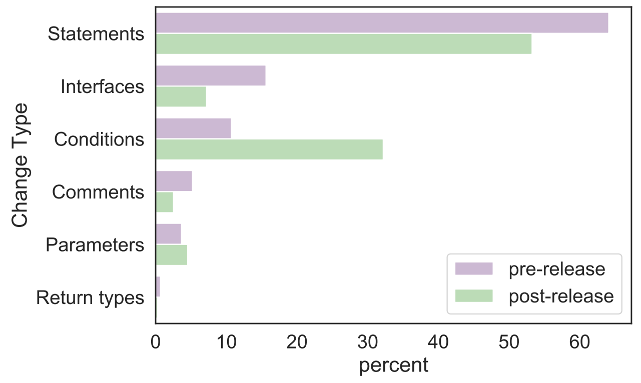


Fig. 6. Frequency of edit action in pre- and post-release patches.

to as "Interface" appearing 15.62% of the time. The fact that this number is about twice as large as for the post-release bug fixes, 7.21%, can be explain by the fact that in post-release contracts are supposed to be fixed and developers are shier to changes in interfaces.

Furthermore, we see that modifications of the control flow by changes represented by the "Conditions" category are much more frequent in post-release patches, accounting for 32.20% of the total, than in pre-release fixes with only 10.72%. One explanation for this result might come from the fact that in post-release, we observe wrong behaviors due to complex data flow, which are managed by the control logic of the program.

Sobreira *et al.* [14] conducted a similar analysis on the Defects4J dataset collecting the patterns manually. Similar to our results they observe a prevalence of changes in method calls and assignment, what we call in our work "Statements Changes". They also show that changes appear to a large extent in conditional statement, similar to what we observe in post-release but not in pre-release. Zhong *et al.* [25] extract change patterns using the same tools as in this study, *Change Distiller*. Their results show that the changes in statements (i.e. *ExpressionStatement*, *VariableDeclaraionStatement*, *SingleVariableDeclaration*) is the predominant type of changes, similar to our results. However, their findings differ from ours mainly in two categories: changes made to comments and changes made on return statements. While in our case both these categories are anecdotal (less than 5% for comments and less than 1% for return statements), in their work, they account for a large proportion of the changes. The difference in term of comment can be explained by the nature of the projects, open source *vs* industrial, where in corporation settings, documentation is often externalize to separate documentation. Zhao *et al.* [26] conducted a similar analysis on C projects. While their results are not directly comparable, some interesting observations can be made for general trends. Indeed, we can see that in their study, the two most prevalent change patterns are *Changes on function call (CFC)* and *Changes on assignment statements (CAS)*, which would correspond in our study to the category "Statements". The category *Change on branch statements (CBS)* is the third most common change pattern and would contribute to in our case to the "Conditions" category. Those results are similar to the findings made by Sobreira *et al.* on the Defects4J dataset.

TABLE IV
PERCENTAGE OF PATCHES ADDRESSING SPOTBUGS VIOLATIONS

Violation	Pre-release	Post-release
Bad Practices	3.68%	0.00%
Correctness	2.63%	0.00%
Malicious code vulnerability	1.05%	0.00%
Multithreaded correctness	0.52%	0.00%
Performance	2.37%	0.00%
Security	0.00%	0.00%
Style	3.42%	0.00%
No violation	94.21%	100.00%

In conclusion, we observe that statement changes, such as addition, modification and removal of function calls and assignment operations is the most common type of changes across all studies, while other types of changes differs from one dataset to the next. However, changes in conditional/logic blocks remains well represented in many studies, in contradiction to what our results might suggest for pre-release patches.

Adding, modifying and removing a statement, typically a function call or an assignment operation, is the most common pattern among pre- and post-release bug fixes, performed 64.10% and 53.25% of the time, respectively. Pre-release bug fixes tend to involve more design-related changes, e.g., changing interfaces, method signatures, etc., whereas post-release patches involve more control flow modifications, e.g, modification of conditional statement or addition of a conditional block.

E. Static analysis

To answer our fourth research question, we run the static analysis tool SpotBugs and report how many violations are fixed between the buggy version and the fixed version, hence showing how many patches addressed a violation. Because of time and computation budget constraints, we restrict our analysis to 5 randomly selected projects from our pool of 37 projects. The results are reported in Table IV.

Pre-release patches almost never address SpotBugs violations, with only 5.79% of them fixing a violation. The patches that fixed violation mainly addressed bad practices (3.68%) and bad style (3.42%) which can suggest that some refactoring happened as well during the bug fix. Correctness violations are only addressed in 2.63% of the bug fixes, 1.63% for the code vulnerabilities and drop to 0% for the security issues. Such low number can in part be explained by the integration of static analysis tooling directly integrated in modern integrated development environment (IDE), such as Eclipse and IntelliJ, used by the developers at the company. These IDEs provide direct feedback before changes are committed to the version control system, thus making them invisible to our study.

More surprisingly, no violations were fixed by post-release patches. This can be explained by the fact that once the program has been cleared by the quality process for production, such violations are typically reported, in particular with the use of continuous monitoring tools such as SonarQube, and have already been addressed.

Listing 2. Patch fixing bug CHART-6 from Defects4J

```
1 int  index  =  this . plot . getIndexOf( this ) ;
2 CategoryDataset  dataset  =
3    this . plot . getDataset (index) ;
4 -if(dataset != null){
5 +if(dataset == null){
6    return   result ;
7 }
8 int   seriesCount  =  dataset . getRowCount();
```

Listing-2 shows an example of a patch in the Chart project from the Defects4J dataset where SpotBugs successfully exposed a violation leading to a null pointer exception. In the buggy version, if line 2 returns a null value, the program will raise a null pointer exception when it will reach line 8. The fix consists in changing the condition (lines 4-5) and return if *dataset* is null.

Listing 3. Patch fixing bug CHART-10 from Defects4J

```
public  boolean  equals (Object  obj ){
   /*...*/
- return super.equals(obj);
+ SuperList that = (ShapeList)obj;
+ int listSize = size();
+ for(int i = 0; i ¡ listSize; i++){
+ if(!ShapeUtilities.equals(Shape)get(i),
+ (Shape)that.get(i)){
+ return false;
+ }
+ }
+ return true;
}
```

However, in listing-3 we show another bug fix from the Chart program where static analysis was enable to spot any violation. In this example, the overloaded *equals* function of an object was not comparing properly, requiring additional point of comparison that the base class provided. SpotBugs fails to capture the bug because the fault is affecting a functionality specific to the program. Detecting functional faults might be quite a challenge, if not impossible, for static analysis tools.

Static analysis tools are typically used in fault localization and automatic program repair to spot potential defects. However, we see that in our experiments, only 2.63% of the fixes addressed a correctness issue flagged by static analysis. This results confirms the disbelief of one of the practitioners interviewed by Kochlar *et al.* [27] who answered on the question about the importance of fault localization: *"I'm well aware of what static analysis can do and very few hard bugs would be solved with it."*.

Correctness violations were only addressed in 2.63% of the pre-release bug fixes and in 1.63% for the code vulnerabilities. 94.21% didn't address any violation in pre-release while none of the post-release bug fixes addressed a violation.

VI. Threats to Validity

In this section, we discuss the internal and external threats to the validity of our results. Internal validity is the extent to which conclusions can be drawn from the causal effect of our data and analysis. External validity is the extent to which the conclusion can be generalized to other settings.

The main threat to the construct validity in empirical analysis of bug fixes is the quality of patches selected. Indeed, as stated by Kim *et al.* [28], identifying fixes is hard because they are often "polluted" by other activities such as refactoring. Our strategy to extract pre-release bugs is sensitive to this problem but this is not the case for post-release ones. Given that the patch size of post-release bug fixes was greater, we expect this threat not to negatively affect our results.

Our result confirm this hypothesis where we see that documentation files are edited. Furthermore, the high percentage of architecture bug fixes can suggest the presence of refactoring activities in those patches. However, comparing our results to a clean bugs dataset, Defects4J, we observe similar trends and our conclusions still hold.

In terms of external validity, the threat includes the generality of our findings. In this study, we analyzed 37 projects originating from the same company, BGL BNP Paribas, therefore, from a single domain. Although, the company is a large player and representative of their field, there is no guarantee that our results would generalize to other companies, evolving in other domains. To address this problem, we put parts of our findings in perspective with previous studies conducted on open source projects, and our results confirm previous work.

VII. Related Work

Mining bug fixes by extracting them from software revision histories in order to extract insights is a common practice that is often found in the literature [17], [25], [26], [29]–[31]. However, these studies differ from ours in the subjects they analyze, large monolithic projects, mostly from the open source world. In this work, we present results from components of a software oriented architecture used by a large corporation to support their business activities.

In their work, Zhao *et al.* [26] perform a detailed analysis of bug-fixing changes occurring in 17 version of 11 well-known open-source systems divided across three domains. They describe 5 types of changes further divided into 9 subtypes. They conclude that interface changes are the most frequent changes and that the frequency of change types is similar across the system they studied. Similarly, Pan *et al.* [29] introduced 27 bug fix patterns over 9 categories. They use their bug extractor tool to analyzing seven Java projects.

Their results show that about half of the bug fixes fall into well known categories.

Wang *et al.* [17] conduct an empirical study on four open source projects. In their work, they present a detailed analysis of multi-entity bug fixes, i.e. classes, methods, fields, and present their frequency, composition and semantic meaning. They conclude that the majority of bug fixes involve multi-entity changes. Furthermore, they show that some recurring change patterns exist in all the projects under study.

Liu *et al.* [32] conduct a fine-grained study of 16,450 bug fix commits from seven open source Java projects, focusing at the expression level. They show results offering new ways to further improve APR tools focusing on fault localization. Similarly, Zhong *et al.* [25] perform an empirical study on more than 9000 real world bug fixes from six popular Java projects. Using their tool BugStat, they compare characteristic of manual fixes with automatic program repair and come up with 15 findings contributing at improving the state-of-the-art of automatic program repair.

Sobreira *et al.* [14] perform a similar type of analysis, but focus their work on a curated benchmark for Java bug fixes, Defects4J [16]. While the set of bug fixes is substantially smaller than in other studies, with only 395 bugs, it present the advantage of being carefully cleaned and curated. The study shows that the median size of Defects4J patches is 4 lines. Our results show much larger values (8.5 for pre-release bug fixes and 16.5 for post-release ones) that could be explained by the noise introduced by other activities such as refactoring or documentation activities (JavaDoc, code comments). However, we observe in both studies the same trends: the most common type of change is line addition, followed by line modification and line removal in both studies has a median value of 0.

On another hand, far fewer studies [33]–[35] tackled the question of comparing post-release and pre-release bug fixes. One of the reason lies in the lack of a clear definition as what would be a pre-release vs a post-release bug fix in open source projects, while access to industrial data is much scarcer for the research community.

Li *et al.* [34] preform an analysis of the reports submitted to the Windows Error Reporting (WER) for the Windows 7 operating system. They discriminate the incidents between ones submitted for beta version (pre-release) and the ones submitted for release versions (post-release). Using the incident reports, the authors build usage characteristic profiles and compare their distribution in pre-release incident and post-release incident. Their result show that usage environments and usage scenarios can differ between pre-release and post-release machines, leading to misleading field defect predictions.

Dey *et al.* [36] conduct study where they focus on the post-release faults in order to model the relationship between software usage and software crashes. Using that model the build a quality measure for software releases. They looked at software usage, release specific information and the number of application crash of 169 Android applications. Their results suggest that the number of new user and release start date to be the determining factor to predict the number of exception reported.

Finally Abou Zeinab *et al.* [35] propose an analysis of the *BugZilla* reports over 15 years for the Eclipse projects comparing the activity before and after release. They observe that more effort is spent for handling bugs before an upcoming release, but over time, especially with the adoption of shorter release cycles, the workload before and after releases tend to be increasingly balanced.

VIII. Conclusion

Software bugs, albeit dangerous and costly for practitioners play an important role in the research community as they drive the creation and evaluation of Software Engineering advances. Thus, researchers have focused on understanding such characteristics and their implications on software testing and debugging research. Unfortunately, industrial studies on such characteristics are scarce.

To bridge this gap, this paper presents an extensive industrial study on the characteristics of pre-release and post-release bugs, i.e., bugs found during software development and after their release to production, respectively. More precise, we analyze 37 Java projects with 250 post-release bug fixes and 3,623 pre-release ones and study their similarities and differences.

Our study shows that pre-release bugs are usually composed of edits in both configuration and source code files. These edits are mainly additions, with typically 9 lines added per bug fix. The changes are usually 'local' and can be localized to 2 chunks in one method. Post-release bugs are mostly composed of line additions in both code and configuration files. In source code files, we typically observe that about 16 lines of code are edited, which happens on 3 to 4 chunks in a single method.

We also show, that both pre- and post-release bugs appear in similar types of files and locations (mostly localized in a single method). However, post-release bugs are typically larger, in term of number of lines edited, than pre-release ones and more disperse across the code base. In both cases the most common change pattern is a statement change where the post-release bugs mainly involve control flow modifications while pre-release ones involve design changes such interface and modifications on parent classes.

Finally, our experiment shows that approximately 5% of the pre-release bugs addressed a violation exhibited by static analysis, where none were addressed by post-release bug fixes. This results suggest that changes in both pre- and post-release bug fixes are more behavioral than structural or too complex for static analysis tools to catch.

IX. Acknowledgements

This work is partially funded by Alphonse Weicker Foundation and by the Luxembourg National Research Fund[3]

[3]references C17/IS/11686509/CODEMATES and AFR PHD 11278802

References

[1] Tricentis. (2018) Software fail watch: 5th edition. Accessed: 2019-04-02. [Online]. Available: https://www.tricentis.com/news/tricentis-software-fail-watch-finds-3-6-billion-people-affected-and-1-7-trillion-revenue-lost-by-software-failures-last-year/

[2] M. Papadakis, D. Shin, S. Yoo, and D. Bae, "Are mutation scores correlated with real fault detection?: a large scale empirical study on the relationship between mutants and real faults," in *Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*, 2018, pp. 537–548.

[3] P. Ammann and J. Offutt, *Introduction to software testing*. Cambridge University Press, 2008.

[4] D. Hovemeyer and W. Pugh, "Finding bugs is easy," in *Companion to the 19th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2004, October 24-28, 2004, Vancouver, BC, Canada*, 2004, pp. 132–136.

[5] J. A. Jones and M. J. Harrold, "Empirical evaluation of the tarantula automatic fault-localization technique," in *20th IEEE/ACM International Conference on Automated Software Engineering (ASE 2005), November 7-11, 2005, Long Beach, CA, USA*, 2005, pp. 273–282.

[6] C. Le Goues, T. Nguyen, S. Forrest, and W. Weimer, "Genprog: A generic method for automatic software repair," *IEEE Trans. Software Eng.*, vol. 38, no. 1, pp. 54–72, 2012.

[7] H. Do, S. Elbaum, and G. Rothermel, "Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact," *Empirical Software Engineering*, vol. 10, no. 4, pp. 405–435, Oct 2005.

[8] R. Just, D. Jalali, and M. D. Ernst, "Defects4j: A database of existing faults to enable controlled testing studies for java programs," in *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, ser. ISSTA 2014. New York, NY, USA: ACM, 2014, pp. 437–440.

[9] R. K. Saha, Y. Lyu, W. Lam, H. Yoshida, and M. R. Prasad, "Bugs.jar: A large-scale, diverse dataset of real-world java bugs," in *Proceedings of the 15th International Conference on Mining Software Repositories*, ser. MSR '18. New York, NY, USA: ACM, 2018, pp. 10–13.

[10] J. M. Voas and G. McGraw, *Software Fault Injection: Inoculating Programs Against Errors*. New York, NY, USA: John Wiley & Sons, Inc., 1997.

[11] T. T. Chekam, M. Papadakis, Y. L. Traon, and M. Harman, "An empirical study on mutation, statement and branch coverage fault revelation that avoids the unreliable clean program assumption," in *Proceedings of the 39th International Conference on Software Engineering, ICSE 2017, Buenos Aires, Argentina, May 20-28, 2017*, 2017, pp. 597–608.

[12] ISDW Group, "IEEE Standard Classification for Software Anomalies," *IEEE Std 1044-2009 (Revision of IEEE Std 1044-1993)*, vol. 1044, no. 2, pp. 1–23, 2010.

[13] R. Chillarege, I. S. Bhandari, J. K. Chaar, M. J. Halliday, D. S. Moebus, B. K. Ray, and M. Wong, "Orthogonal defect classification - A concept for in-process measurements," *IEEE Trans. Software Eng.*, vol. 18, no. 11, pp. 943–956, 1992.

[14] V. Sobreira, T. Durieux, F. Madeiral, M. Monperrus, and M. de Almeida Maia, "Dissection of a bug dataset: Anatomy of 395 patches from Defects4J," in *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, mar 2018, pp. 130–140.

[15] A. Mockus and L. G. Votta, "Identifying Reasons for Software Changes Using Historic Databases," in *Proceedings of the International Conference on Software Maintenance*. IEEE Computer Society, 2000, pp. 120–130.

[16] R. Just, D. Jalali, and M. D. Ernst, "Defects4J: a database of existing faults to enable controlled testing studies for Java programs," in *Proceedings of the 2014 International Symposium on Software Testing and Analysis - ISSTA 2014*, San Jose, CA, USA, 2014, pp. 437–440.

[17] Y. Wang, N. Meng, and H. Zhong, "An Empirical Study of Multi-entity Changes in Real Bug Fixes," in *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, no. 1. IEEE, sep 2018, pp. 287–298.

[18] B. Fluri, M. Wuersch, M. PInzger, and H. Gall, "Change Distilling:Tree Differencing for Fine-Grained Source Code Change Extraction," *IEEE Transactions on Software Engineering*, vol. 33, no. 11, pp. 725–743, nov 2007.

[19] S. S. Chawathe, A. Rajaraman, H. Garcia-Molina, and J. Widom, "Change detection in hierarchically structured information," in *Proceedings of the 1996 ACM SIGMOD international conference on Management of data - SIGMOD '96*. New York, New York, USA: ACM Press, 1996, pp. 493–504.

[20] H. C. Gall, B. Fluri, and M. Pinzger, "Change Analysis with Evolizer and ChangeDistiller," *IEEE Software*, vol. 26, no. 1, pp. 26–33, jan 2009.

[21] M. Beller, R. Bholanath, S. McIntosh, and A. Zaidman, "Analyzing the State of Static Analysis: A Large-Scale Evaluation in Open Source Software," *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, vol. 1, pp. 470–481, 2016.

[22] D. Hovemeyer and W. Pugh, "Finding bugs is easy," vol. 39, no. 12, p. 92, dec 2004.

[23] "SpotBugs bug description," https://spotbugs.readthedocs.io/en/latest/bugDescriptions.html, accessed: 2019-04-06.

[24] Z. Yin, X. Ma, J. Zheng, Y. Zhou, L. N. Bairavasundaram, and S. Pasupathy, "An empirical study on configuration errors in commercial and open source systems," in *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles - SOSP '11*. New York, New York, USA: ACM Press, 2011, p. 159.

[25] H. Zhong and Z. Su, "An Empirical Study on Real Bug Fixes," in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 1. IEEE, may 2015, pp. 913–923.

[26] Y. Zhao, H. Leung, Y. Yang, Y. Zhou, and B. Xu, "Towards an understanding of change types in bug fixing code," *Information and Software Technology*, vol. 86, pp. 37–53, 2017.

[27] P. S. Kochhar, X. Xia, D. Lo, and S. Li, "Practitioners' expectations on automated fault localization," in *Proceedings of the 25th International Symposium on Software Testing and Analysis - ISSTA 2016*. New York, New York, USA: ACM Press, 2016, pp. 165–176.

[28] S. Kim, H. Zhang, R. Wu, and L. Gong, "Dealing with noise in defect prediction," in *Proceeding of the 33rd international conference on Software engineering - ICSE '11*. New York, New York, USA: ACM Press, 2011, pp. 481–490.

[29] K. Pan, S. Kim, and E. J. Whitehead, "Toward an understanding of bug fix patterns," *Empirical Software Engineering*, vol. 14, no. 3, pp. 286–315, jun 2009.

[30] B. Livshits and T. Zimmermann, "DynaMine," in *Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering - ESEC/FSE-13*, vol. 30, no. 5. New York, New York, USA: ACM Press, 2005, p. 296.

[31] H. Zhong and N. Meng, "Towards reusing hints from past fixes: An exploratory study on thousands of real samples," *Empirical Software Engineering*, vol. 23, no. 5, pp. 2521–2549, 2018.

[32] K. Liu, D. Kim, A. Koyuncu, L. Li, T. F. Bissyande, and Y. Le Traon, "A Closer Look at Real-World Patches," in *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, sep 2018, pp. 275–286.

[33] P. L. Li, M. Ni, S. Xue, J. P. Mullally, M. Garzia, and M. Khambatti, "Reliability assessment of mass-market software: Insights from windows vista®," *Proceedings - International Symposium on Software Reliability Engineering, ISSRE*, pp. 265–270, 2008.

[34] P. L. Li, R. Kivett, Z. Zhan, S.-e. Jeon, N. Nagappan, B. Murphy, and A. J. Ko, "Characterizing the differences between pre- and post- release versions of software," in *Proceeding of the 33rd international conference on Software engineering - ICSE '11*. New York, New York, USA: ACM Press, 2011, p. 716.

[35] K. Abou Zeinab, E. Constantino, and T. Mens, "A Longitudinal Analysis of Bug Handling Across Eclipse Releases," in *35th IEEE International COnference on Sotftware Maintenance and Evolution (ICSME)*. Cleveland, USA: IEEE Computer Society, 2019.

[36] T. Dey and A. Mockus, "Modeling Relationship between Post-Release Faults and Usage in Mobile Software," in *Proceedings of the 14th International Conference on Predictive Models and Data Analytics in Software Engineering - PROMISE'18*. New York, New York, USA: ACM Press, 2018, pp. 56–65.