The Journal of Systems and Software 217 (2024) 112179

Contents lists available at ScienceDirect

The Journal of Systems & Software

journal homepage: www.elsevier.com/locate/jss

An empirical study on bug severity estimation using source code metrics and static analysis☆

Ehsan Mashhadi [a], Shaiful Chowdhury [b], Somayeh Modaberi [a], Hadi Hemmati [a,c,*], Gias Uddin [a]

[a] University of Calgary, Calgary, Alberta, Canada
[b] University of Manitoba, Winnipeg, Manitoba, Canada
[c] York University, Toronto, Ontario, Canada

ARTICLE INFO

Keywords:
Bug severity
Defect prediction
Code complexity metrics
Static analysis tools

ABSTRACT

In the past couple of decades, significant research efforts have been devoted to the prediction of software bugs (i.e., defects). In general, these works leverage a diverse set of metrics, tools, and techniques to predict which classes, methods, lines, or commits are buggy. However, most existing work in this domain treats all bugs the same, which is not the case in practice. The more severe the bugs the higher their consequences. Therefore, it is important for a defect prediction method to estimate the severity of the identified bugs, so that the higher severity ones get immediate attention. In this paper, we provide a quantitative and qualitative study on two popular datasets (Defects4J and Bugs.jar), using 10 common source code metrics, and two popular static analysis tools (SpotBugs and Infer) for analyzing their capability to predict defects and their severity. We studied 3,358 buggy methods with different severity labels from 19 Java open-source projects. Results show that although code metrics are useful in predicting buggy code (Lines of the Code, Maintainable Index, FanOut, and Effort metrics are the best), they cannot estimate the severity level of the bugs. In addition, we observed that static analysis tools have weak performance in both predicting bugs (F1 score range of 3.1%–7.1%) and their severity label (F1 score under 2%). We also manually studied the characteristics of the severe bugs to identify possible reasons behind the weak performance of code metrics and static analysis tools in estimating their severity. Also, our categorization shows that Security bugs have high severity in most cases while Edge/Boundary faults have low severity. Finally, we discuss the practical implications of the results and propose new directions for future research.

1. Introduction

Software maintenance is one of the most challenging and expensive phases in the software development life cycle (Kafura and Reddy, 1987). Handling bugs (including detecting, localizing, fixing, etc.) is the most typical challenge associated with the software maintenance step (Börstler and Paech, 2016; Bennett and Rajlich, 2000). Consequently, both practitioners and researchers are trying to make this tedious task as automated as possible from different aspects such as defect prediction, test generation, fault-localization, and program repair (Kondo et al., 2020; Shin et al., 2010; Tosun et al., 2010; Zhou et al., 2010; Mashhadi and Hemmati, 2021).

While there has been much research in handling bugs using different techniques like search-based (Le Goues et al., 2011), pattern-based (Long and Rinard, 2016) and ML-based techniques (Mashhadi and Hemmati, 2021), there is little research that focuses explicitly on the severe bugs. In other words, most of the research implicitly assumes that all bugs have the same importance (Shamshiri et al., 2015; Mashhadi and Hemmati, 2021; Wong et al., 2016; Pearson et al., 2017). However, the bug severity indicates the intensity of the impact the bug has on system operation (Neysiani et al., 2020). Critical bugs may cause a system to crash completely or cause non-recoverable conditions such as data loss. High-severity bugs affect major system components that prevent users from working with some parts of the system. Presumably, fixing severe bugs is typically more challenging compared to the medium or low severity bugs, where few components are affected and there is an easy workaround (Vucevic and Yaddow, 2012). Therefore, in practice, bugs with higher severity tend to be fixed sooner than other less severe bugs (Uddin et al., 2017; Kanwal and Maqbool, 2012; Saha et al., 2014).

In general, software practitioners have several means to detect bugs. These can range from QA practices such as code review (Kononenko et al., 2016; Mäntylä and Lassenius, 2008; Bacchelli and Bird, 2013),

☆ Editor: Prof. Neil Ernst.

* Corresponding author at: York University, Toronto, Ontario, Canada.
E-mail address: hemmati@yorku.ca (H. Hemmati).

https://doi.org/10.1016/j.jss.2024.112179
Received 9 June 2023; Received in revised form 23 March 2024; Accepted 27 July 2024
Available online 6 August 2024

and inspection to different testing approaches, and even development methodologies such as pair programming (Williams et al., 2000; Nawrocki and Wojciechowski, 2001; Sun et al., 2015), and Test Driven Development (TDD) (Bhat and Nagappan, 2006; Martin, 2007; Aniche et al., 2013; Rafique and Mišić, 2012). When a bug is detected, it is common for the development teams to consider the most severe/important bugs first which helps them to prevent extreme consequences. Some issue tracking systems, such as Jira (ATLASSIAN, 2023), have a specific field named Severity/Priority which is assigned during the bug reporting and helps the development teams to consider bug importance during debugging.

It is important to note that bug severity does not have a standard formal definition. The definition depends on the context of the software, the nature of the bug, the current state of the project, the ratio of affected users, potential harm to users, and many other factors. The current state of the practice to identify and record the severity of bugs is through a manual process in issue tracking systems, where the severity of a bug has its own field (with options such as *Blocker*, *Critical*, *Major*, *Minor*, and *Trivial* or sometimes with numbers ranging from zero to 20). The field is manually populated by the person (e.g., a developer, tester, or user) who documents the bug, which, however, may change by the technical team during the bug reporting review process. Thus, the whole process of bug labeling is quite expensive. To alleviate this difficulty, significant research has focused on automatically labeling bug severity to a bug report (e.g., Tian et al., 2012, 2013; Ramay et al., 2019; Lamkanfi et al., 2010).

Predicting the severity of bug reports, however, does not help in detecting bugs and their severity from the beginning of the software development life cycle (SDLC). Detecting bugs at the later stages of SDLC is much more expensive than detecting and fixing them early (Celerity, 2022). In this paper, we aim to study the feasibility of predicting software bugs and their severity using source code metrics and different static analysis tools. Code metrics have shown a great deal of success in predicting code smell (Tufano et al., 2015), maintenance effort (Polo et al., 2001) and defects (Giger et al., 2012; Ferenc et al., 2020a). Static analysis tools such as SpotBugs (SpotBugs, 2023), the successor to the popular FindBugs (FindBugs, 2015) tool, Facebook Infer (Infer, 2023), and Google Error Prone (ErrorProne, 2023) have also been successfully used in related research (Habib and Pradel, 2018; Tomassi, 2018). These tools use different techniques such as AST-based patterns or dataflow analysis to find a bug's existence and to predict the bug's type and severity.

Our approach focuses on method-level granularity in contrast to class/module-level, since many studies have shown that developers find the class/module level granularity impractical (too coarse-grained to be useful for remedial actions) (Shihab et al., 2012; Pascarella et al., 2020; Grund et al., 2021; Hata et al., 2012). We use two popular datasets: Defects4J (Just et al., 2014), and Bugs.jar (Saha et al., 2018) which contain real bugs from different open-source Java projects. We studied 19 projects containing 1668 bugs (3358 buggy methods) for our quantitative study. Furthermore, we studied 140 randomly sampled bugs from both datasets for our qualitative study to find out when and why different code metrics and the existing static analysis tools fail to distinguish bug severity.

To guide our study, we aim to answer the following research questions (RQ):

- **RQ1: Are source code metrics good indicators of buggyness and bug severity?** The results show that most of the code metrics (e.g., Lines of Code, McCabe, McClure, Nested Block Depth, Proxy Indentation, FanOut, Readability, Difficulty, and Effort) are good indicators of buggyness, but they perform quite poorly for predicting bug severity. For example, the Line of Code metric shows excellent performance in finding buggyness, but very poor performance in finding bug severity. The Halstead Difficulty and Effort metrics show good performance in identifying bug severity when compared to other metrics.

Table 1
Unifying the raw severity labels of Defects4J dataset. For example, the third row means that bugs having raw severity labels of Medium or 5 are considered as Medium bugs.

Unified Severity Label (USL)	Raw Severity Label (RSL)
Critical	Critical
High	High, Major, 3
Medium	Medium, 5
Low	Low, Trivial, Minor, 7, 8, 9

- **RQ2: What is the capability of static analysis tools in finding bugs and their severity?** Results show that the studied static analysis tools (SpotBugs, and Infer) are not yet powerful enough to find many bug types, and in many cases, they mislabel the bug severity. Based on our experiments, these tools often fail to identify numerous bugs due to insufficient built-in patterns. Additionally, they struggle to accurately assess the severity of bugs, as they assign fixed severity values to each bug type without considering the contextual factors and potential consequences that could classify them as severe or non-severe bugs.
- **RQ3: What are the characteristics of bugs with different severity values?** Results reveal that the severity of bugs is mostly related to the software's specification, which is not predictable solely based on code metrics or static analysis. Also, we found no direct relationship between method complexity and its severity value. Many low-severity bugs exist in the quite complex methods according to the code metrics, but these functions handle trivial functionalities, such as GUI, or they do not lead to a crash or unauthorized access.

The findings of this paper can help researchers and practitioners to better understand the characteristics of severe bugs and guide future research/tools on how to advance the field to better predict bug severity. We provide concrete future directions in this regard in Section 4. All the data and source code of this study are also publicly available (Mashhadi, 2023) for replication studies.

The remainder of the paper is organized as follows: We provide our dataset, and experiments setup in Section 2. Experiment motivation, design, and results are discussed in Section 3. Discussions about the results are provided in Section 4. Threats to the validity of our work are described in Section 5. Related works to this paper are described in Section 6. We conclude this paper with possible future work in Section 7.

2. Study setup

Fig. 1 summarizes our approach of collecting bug severity labels from two existing datasets (step 1), merging the raw severity labels to unified severity labels (step 2), and then formulating the research questions (step 3). We also discuss how we identify the set of buggy and not-buggy methods. Then we discuss the method-level source code metrics and the static analysis tools that we study in this paper.

2.1. (Step1) Bug datasets and severity extraction

We use Defect4J (Just et al., 2014) and Bugs.jar (Saha et al., 2018) datasets for our study which contain bugs from different popular Java projects. Defect4J is used because it has been widely used in different automated software engineering research domains such as Test Generation (Shamshiri et al., 2015), Program Repair (Martinez et al., 2017), and Fault Localization (Pearson et al., 2017). Also, this is a generic dataset containing real bugs from different projects with different domains. Bugs.jar has also been widely used in Program Repair (Saha et al., 2017) and contains many real bugs and there is a corresponding bug report for each sample in Jira (ATLASSIAN, 2023) containing the bug severity.

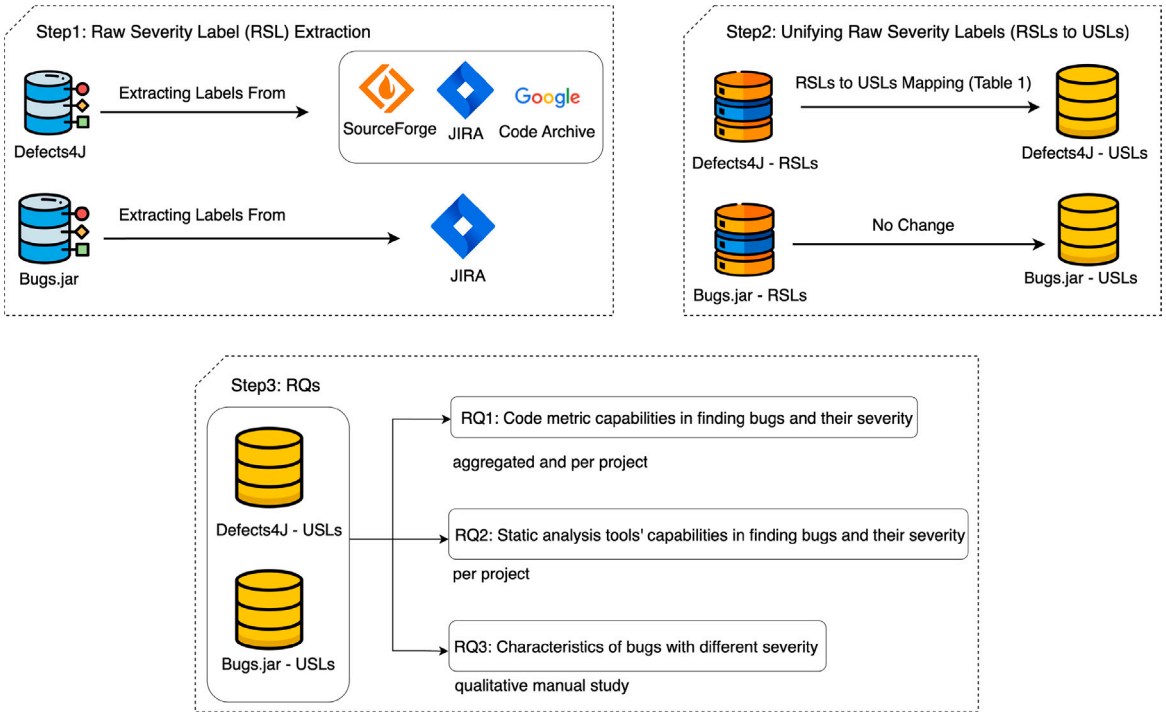


Fig. 1. Overview of the setup of our empirical study.

The latest version (2.0.0) of Defects4J contains 835 bugs from 17 Java projects. The dataset does not have any information regarding the bug severity itself, so we extract bug severity values from different issue-tracking systems. There are three software hosting services used for these projects: Jira, SourceForge, and Google Code Archive. We implemented scripts (available in our provided repository (Mashhadi, 2023)) that either use open-source SDKs or scrap the web pages to extract the severity labels. After extracting the severity values from these systems, we collected 510 bugs out of 835 bugs containing severity labels.

The Bugs.jar dataset contains 1158 bugs from eight large, popular, and diverse open-source Java projects. All of these projects use Jira as an issue management system, and each of the bugs has an assigned severity level (extracted in a similar way to the Defects4J dataset). The severity labels from this dataset are "Blocker", "Critical", "Major", "Minor", and "Trivial". Using the same technique mentioned above,

2.2. (Step2) Unifying raw severity labels (RSL to USL)

After extracting the raw severity labels (RSL) of the Defects4J dataset, we found that there are many inconsistencies between these values. Some projects use numerical values as an indicator for severity labels (smaller values indicate higher severity while larger values indicate lower severity) and others use categorical values due to the nature of their different issue-tracking systems. Therefore, we unified similar raw labels (RSL) to produce four meaningful unified severity labels (USL): "Critical", "High", "Medium", and "Low" severity labels. Table 1 shows the mapping between all RSL and the USL labels of the Defects4J dataset.

For the Bugs.jar dataset, we found that the extracted RSL values are consistent (since all of the bugs in the dataset are in the Jira issue-tracking system), so there is no need for a unifying process (RLS to USL) like what we did for the Defects4J dataset, and we considered all the RSLs as USLs.

It is important to note that, for both datasets, we have relied on human-labeled bug severity. In any bug-reporting system, some guidelines and examples exist to help practitioners and users decide what should be the label of a reported bug. For example, in Jira,[1] a severity level 1 (i.e., critical) is defined as, *A critical incident with very high impact*. The presented examples are, *A customer-facing service, like Jira Cloud, is down for all customers*; *Confidentiality or privacy is breached*; and *Customer data loss*. In contrast, a minor bug is defined as, *A minor incident with low impact*, and one of the examples is, *A minor inconvenience to customers, workaround available*. The relevant threats to validity in using human-labeled severity and their mitigation are discussed in Section 5.3.

2.3. (Step3) Research Questions (RQs)

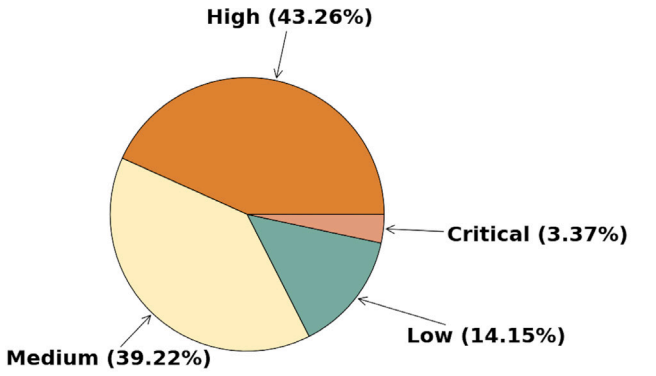
After the mentioned preprocessing step we concluded with two datasets containing the USLs (RSLs are discarded and not used anymore). The Defect4J dataset contains "Critical", "High", "Medium", and "Low". Bugs.jar dataset contains "Blocker", "Critical", "Major", "Minor", and "Trivial". This is our extracted and preprocessed dataset, but we will group some of these USLs in different RQs according to their experiment design requirements, which will be explained in their related sections accordingly. In general, we answer RQ1 and RQ2 with quantitative analysis, and RQ3 is answered with qualitative manual analysis.

2.4. Buggy vs. Not buggy

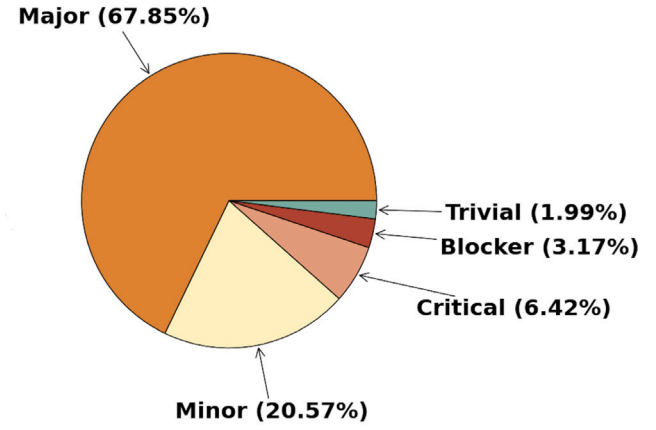
Since we focus on the method-level granularity, we consider a method as a buggy method if it is modified/removed by a bug-fixing patch. If a method, however, is introduced with a bug-fix patch, we do not label this method as buggy. We discard the static initialization blocks and constructors since those are special types of methods that are mostly used for initializing the enclosed class, mostly.

A bug-fixing patch, however, can impact multiple methods, in that case, we consider all of them as buggy. This has been a common practice in earlier studies (e.g., Chowdhury et al., 2022b; Pascarella

[1] https://www.atlassian.com/incident-management/kpis/severity-levels



(a) Defects4J



(b) Bugs.jar

Fig. 2. Buggy methods severity distributions of Defects4J and Bugs.jar datasets with their USL values.

et al., 2020; Mo et al., 2022). We discuss the relevant threats in Section 5. Finally, from the 510 bugs of the Defects4J dataset, we found 742 buggy methods and from the 1158 bugs of the Bugs.jar dataset, we found 2616 buggy methods.

Similar to earlier studies (e.g., Chowdhury et al., 2022b; Pascarella et al., 2020; Mo et al., 2022), we considered a method as non-buggy if it was not modified in the current bug fixing patch inside the buggy class. In this way, we are extracting methods that are contributing to the same functionality as the buggy methods since methods of a class have high cohesion.

We extracted 20,179 and 57,197 non-buggy methods from the Defects4J and Bugs.jar, respectively. The list of the buggy and non-buggy methods containing their project name, class name, start line, end line, and bug severity (corresponding bug severity for non-buggy methods) is available in our publicly shared GitHub repository (Mashhadi, 2023).

The severity distributions of the buggy methods for Defects4J and Bugs.jar datasets are shown in Fig. 2. We study these two datasets separately because of the difference in their labels. The severity distribution per project is shown in Figs. 3 and 4. The figures show that some projects do not have enough samples in all severity categories such as `Chart` project, containing only a few bugs in "Medium" and "Low" severity groups, or the `Collections` project, which only contains High severity bugs. Because of not having enough samples in each project/severity group, we may combine data in different RQs, which we discuss later.

2.5. Source code metrics

Intuitively, if a code component implements a complicated functionality, it is more complex and may contain source code-induced code smells resulting in bugs. Consequently, different source code metrics with module-level, class-level, and method-level granularities have been used to measure software quality and predict bugs in previous research (Giger et al., 2012; Pecorelli et al., 2019; Pascarella et al., 2020; Gil and Lalouche, 2017; Chowdhury et al., 2024, 2022a). These metrics generally focus on how large, complex, readable, and testable a code component is. The success of these code metrics in estimating maintenance efforts, such as predicting bugs, has been historically debated (Chowdhury et al., 2022a). In recent research, however, it was discovered that although code metrics are not useful in understanding software maintenance at the class/file level (Gil and Lalouche, 2017), they are very helpful in understanding maintenance in the method-level source code granularity (Landman et al., 2014; Chowdhury et al., 2022a). Then again, this observation contradicts another recent study by Pascarella et al. (2020), who observed negative results while building code metrics-based method-level bug prediction models. None of these previous studies, however, investigate if code metrics are useful to understand bug severity. *Perhaps, a bug inside a more complex method is more severe than a bug contained in a simpler method.* In this paper, we, therefore, not only investigate code metrics' effectiveness in detecting/predicting bugs at the method level but also their usefulness in understanding bug severity.

We leverage most of the common method-level source code metrics that are used in the previous research (mentioned in the Related Work section) to see their capability in predicting buggy codes and their severity. Although the list of selected metrics is not exhaustive (given the numerous metrics explored in this field and the limitations of experiment size in one article), we made sure we have most of the metrics that have been shown effective in predicting the method-level buggyness, in the past. These metrics are defined as follows:

Lines of Code (LC): Size, also known as lines of code (LC), is the most popular, easy to measure, and the most effective code metric for estimating software maintenance (Gil and Lalouche, 2017; El Emam et al., 2001; Chowdhury et al., 2022b). The use of LC as a proxy maintenance indicator is so prevalent that there are dedicated studies that completely focus on LC and its correlation with other quality metrics (e.g., Gil and Lalouche, 2017; Landman et al., 2014; Chowdhury et al., 2022b). LC has been extensively studied for bug prediction, fault localization, and for finding vulnerabilities (e.g., Chowdhury et al., 2024; Pascarella et al., 2020; Antinyan et al., 2014; Shin et al., 2010; Chowdhury et al., 2022b). In this paper, we calculate LC as the source lines of code without comments and blank lines, similar to Landman et al. (2014), Ralph and Tempero (2018), Chowdhury et al. (2022b) to prevent the code formatting and comments effects.

McCabe (MA): McCabe (McCabe, 1976; Landman et al., 2014), also known as cyclomatic complexity, is another very popular metric that indicates the number of independent paths, and thus the logical complexity of a program. Intuitively, components with high McCabe values are more bug-prone. McCabe has been studied extensively to find bugs and locate suspicious code (Antinyan et al., 2014), to understand its correlation with code quality (Pantiuchina et al., 2018), and to leverage its value for test generation methods, such as structured testing (path testing) (Watson et al., 1996). McCabe can be calculated as $1 + \#predicates$ (McCabe, 1976).

McClure (ML): McClure (McClure, 1978; Kafura and Reddy, 1987) was proposed as an improvement over McCabe. Unlike McCabe, McClure considers the number of control variables, and the number of comparisons in a predicate, which is not supported by McCabe. Intuitively, a predicate with multiple comparisons and multiple control variables would be more complex, and thus more bug-prone, than a predicate with only one comparison or only a single control variable.

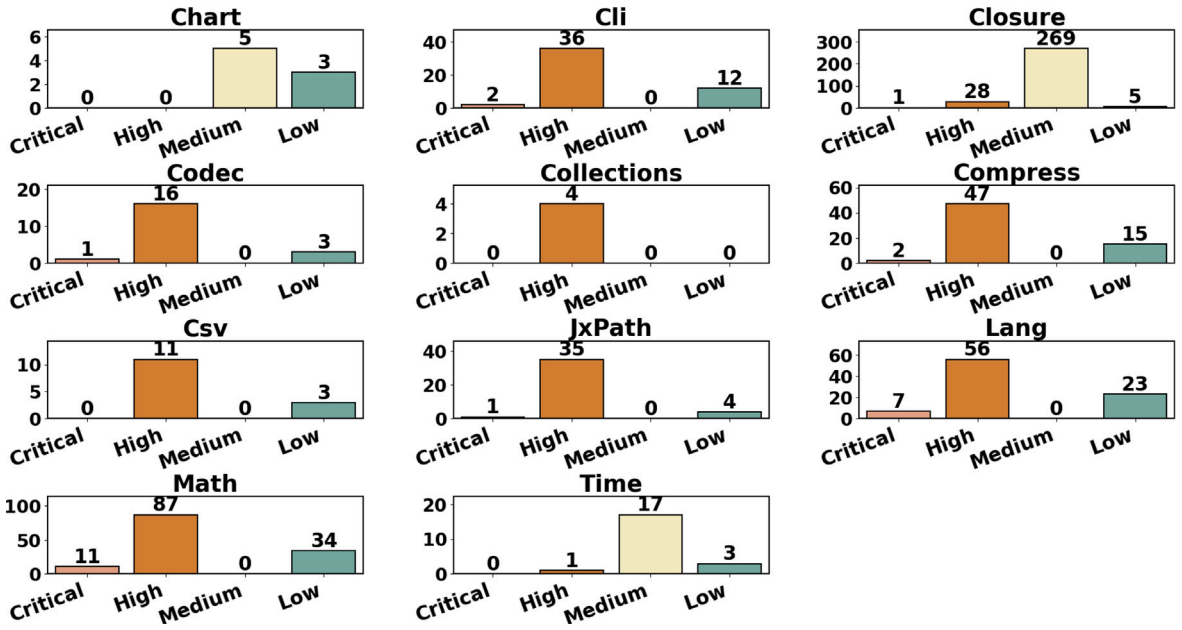


Fig. 3. Buggy methods severity distributions in Defects4J dataset with the USL values.

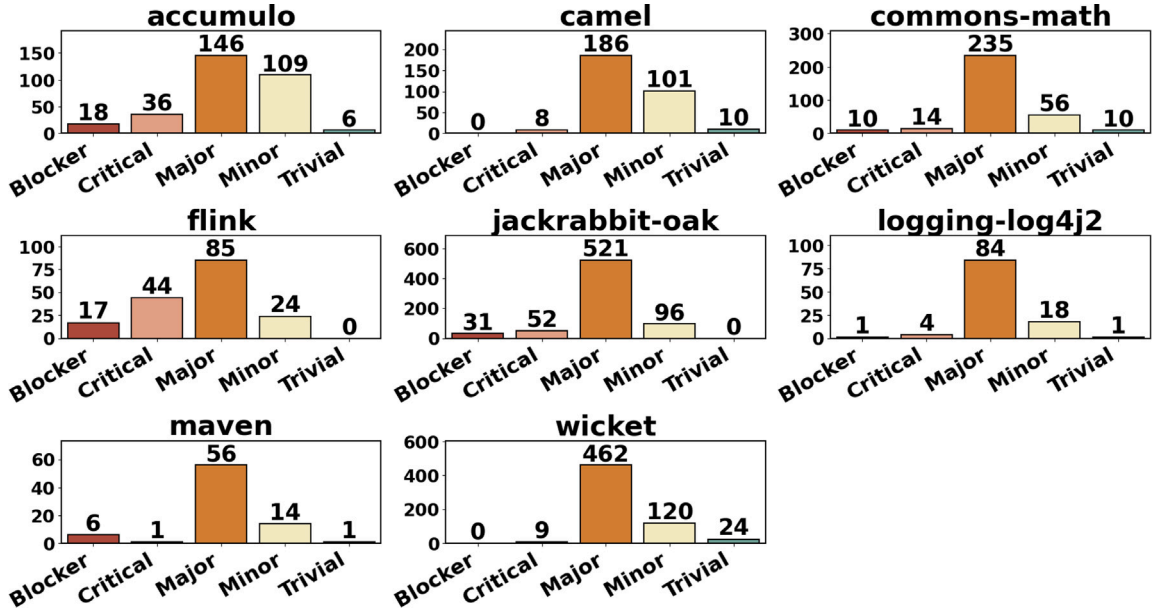


Fig. 4. Buggy methods severity distributions in Bugs.jar dataset with the USL values.

Nested Block Depth (NBD): McCabe and McClure do not consider nested depth. According to these metrics, there is no difference between two code snippets containing two identical *for* loops if they are arranged serially or nested. NBD (Kasto and Whalley, 2013; Alenezi et al., 2019; Zaw et al., 2020) has been studied alongside McCabe and McClure to alleviate this issue.

Proxy Indentation (PI): Since McCabe-like complexity measures require a language-specific parser (for finding the predicates), Hindle et al. (2008) proposed Proxy Indentation metric as a proxy for McCabe-like complexity metrics. It was shown that, for measuring complexity, indentation measurement can perform very similar to more complex measurements such as McCabe, without requiring a language-specific parser. Indentation measurement is done for each line, and then an aggregated value is calculated for the whole program component (e.g., a method). Hindle et al. showed that the standard deviation as an aggregated value outperforms the mean, median, or max. Therefore, we only use the standard deviation form, as was also done by Chowdhury et al. (2022a).

FanOut (FO): This metric calculates the total number of methods called by a given method. This provides an estimate of the coupling — i.e., dependency of a particular method on other methods. It is observed that code components that are highly coupled are less maintainable and bug-prone (Mo et al., 2016; AlOmar et al., 2019).

Readability (R): This metric combines different code features to calculate a single value for estimating code readability. We used the readability metric proposed by Buse and Weimer (2009) which generates a readability score for a given method. The readability scores range from 0 to 1 for specifying least readable code to most readable code, respectively. The authors concluded that this metric has a significant level of correlation with defects, code churn, and self-reported stability.

Table 2
List of studied metrics and their brief description.

Metric	Description
LC	Counts the number of source lines of code without comments and blank lines
MA	Measures McCabe or the number of independent paths (cyclomatic complexity)
ML	Similar to MA, except it also considers the number of control variables and comparisons in a predicate
NBD	Counts the depth of the most nested block
PI	Counts the indentation of source code lines
FO	Counts the total number of methods called by a given method
R	Measures the readability of a method in the range of 0-1 (least to most readable)
D	Measures the Halstead difficulty of a method
E	Measures the required Halstead effort of a method
MI	Measures the maintainability of a method

Halstead Metrics: The Halstead code metrics contain seven measures based on the number of operators and operands in a component (Halstead, 1977). These metrics have been used in different research such as measuring code complexity perceived by developers (Antinyan et al., 2017), calculating the complexity of software maintenance tasks (Curtis et al., 1979; Kafura and Reddy, 1987), finding their correlation with indentation measures (Hindle et al., 2008), and estimating software readability (Posnett et al., 2011). Since all the Halstead metrics are highly correlated to each other, we consider only two of them: **Difficulty (D)** and **Effort (E)** which use other Halstead metrics in their formulas. The Halstead Difficulty is calculated as shown in Eq (1):

$$D = \frac{n1}{2} * \frac{N2}{n2} \tag{1}$$

Where $n1$ is the number of distinct operators, $n2$ is the number of distinct operands, and $N2$ is the total number of operands.

The Halstead Effort is calculated as shown in Eqs (1) and (2)

$$\begin{aligned} E &= D * V \\ V &= N * \log_2(n) \\ N &= N1 + N2 \\ n &= n1 + n2 \end{aligned} \tag{2}$$

Where $N1$ is the total number of operators.

Maintainability Index (MI): The Maintainability Index has been introduced by Omran and Hagemeister (Oman and Hagemeister, 1992) where the authors defined metrics for measuring the maintainability of a software system and combine those metrics into a single value. This metric has evolved over time and was adopted by popular tools like Visual Studio (Microsoft, 2023). MI calculated as shown in Eq (3).

$$\begin{aligned} MI = 171 - 5.2 * ln(HalsteadVolume) - \\ 0.23 * (McCabe) - 16.2 * ln(LC) \end{aligned} \tag{3}$$

Table 2 provides a brief description of each of our 10 studied metrics with their abbreviation.

2.6. Static analysis tools

Static analysis tools have been introduced for finding bugs by using different techniques, such as pattern matching. **FindBugs** FindBugs (2015) is an open-source tool that analyzes Java byte code for finding bugs. **SpotBugs** SpotBugs (2023) is a successor of FindBugs which is actively under maintenance. SpotBugs has more than 400 bug patterns and their description is available on the tool's website (SpotBugs, 2023). **Infer** Infer (2023) is a static analysis tool for different programming languages such as Java, C++, Objective-C, and C, which is also actively maintained. Infer has more than 100 predefined issue types which are described on its website (Infer, 2023). **Error prone** ErrorProne (2023) is another static analysis tool from Google that is used for Java code to catch common programming faults at compile-time, and it is actively under maintenance. We have selected SpotBugs and Infer tools for our study because of their extensive usage in previous studies and in practice, and we exclude ErrorProne because its latest versions do not support older versions of Java projects that exist in our datasets. A more detailed explanation of our selected configuration for these tools is explained in Section 3.2.2.

3. Experiment design and results

In this section, we provide the motivation, experiment designs, and results for each RQ. Since each RQ has a few sub-RQs, they may have different experiment designs which are explained in their relevant sections.

3.1. Code metrics & bug severity (RQ1)

We aim to understand the capabilities of source code metrics in finding bugs and predicting their severity labels.

3.1.1. Motivation

One of the most important attributes in every issue tracking system is bug severity which is usually determined manually by the developers/QA team. This process is time-consuming and error-prone since the technical team should investigate the effects of the reported bug (e.g., number of affected users, number of crashes, probable consequence on the whole system) based on the bug description or other available data from analytical systems (e.g., crash report logging systems).

There has been significant research that uses source code metrics for defect-related issues (more in Section 6), but the severity prediction studies mostly use bug description which works in cases where there exists a well-written bug description. Therefore, in this RQ we study source code metrics' ability to distinguish among different types of bug severities.

3.1.2. Approach

To understand the effectiveness of source code metrics, we answer RQ1, which is divided into two sub-RQs.

RQ1: Are source code metrics good indicators of buggyness and bug severity?

- **RQ1-1**: Do source code metrics distinguish between buggy and not-buggy code?
- **RQ1-2**: Do source code metrics distinguish between different bug severity?

Design of RQ1: In this RQ, we first assess code metrics' capabilities in finding bugs and then evaluate them further in terms of their potential in estimating the bugs' severity. To find the answer to this RQ, we apply a statistical test to see if distributions of methods (e.g., buggy vs non-buggy or critical vs low) are statistically different according to our selected code metrics.

After applying the Shapiro–Wilk test (Shaphiro and Wilk, 1965) using 5% level of significance ($\alpha = 0.05$), we found out that none of our distributions are normal, so we use the non-parametric Wilcoxon Rank-sum test for answering RQ1 with the 5% level of significance ($\alpha = 0.05$) with the following two null-hypotheses for RQ1-1 and RQ1-2:

- RQ1-1: Source code distributions of buggy methods and non-buggy methods are not statistically different.
- RQ1-2: Source code distributions of buggy methods with different severity values are not statistically different.

Table 3
Merging different USLs to create categories for Defect4J and Bugs.jar datasets. For example, the first row means that bugs having Critical or High USLs are considered in the High category.

Dataset	Unified Severity Label (USL)	Merged Category
D4J	Critical, High	High
	Low, Medium	Low
Bugs.jar	Blocker, Critical	Critical
	Major	Major
	Minor, Trivial	Minor

To find the significance of the difference, we calculated Cliff's Delta effect size. We followed the values provided by Hess and Kromrey (2004) for interpreting the result of this value. We considered values smaller than 0.147 to be negligible, values in the range [0.147, 0.33) to be small, values in the range [0.33, 0.474) to be medium, and values greater than 0.474 to be large.

Since we are using the statistical analysis test for this RQ, it is important to make sure that each distribution has enough samples, and that distributions are diverse enough (having samples from different projects to reduce the bias problem) to make the results robust. Also, we discard the methods with $LC < 4$ in this RQ since they are mostly boilerplate code such as setters, getters, and constructors which are generated automatically.

We found that our studied datasets contain imbalanced distributions for RQ1-2 since some USLs contain only a few samples (e.g., the Critical USL in the Defects4J contains only 25 samples) or some USLs do not contain enough bugs from various projects to make that category diverse (e.g., 92% of samples in the Medium USL of the Defect4J dataset are from `Closure` project). Because of the mentioned problem, we merge some USLs (defined in Section 2.1) one level further for RQ1-2 which is shown in Table 3.

As shown in Table 3, for the Defects4J dataset, the Critical USL is merged into the High, and also the Medium and Low USLs are merged, so we ended up with two categories of High and Low severity bugs.

In the Bugs.jar dataset, the Blocker and Critical USLs are merged, and also the Trivial and Minor USLs are merged, so we concluded with three severity categories: Critical, Major, and Minor. The Blocker and Critical USLs are merged since the Critical USL does not contain enough samples, and the Minor and Trivial USLs are merged because of not enough samples in Trivial. We keep the Major USL as an independent category since it has enough samples.

3.1.3. Results

RQ1-1 Answer: Fig. 5 compares the distribution of different code metrics between buggy and non-buggy methods (Fig. 5(a) for the Defect4J, and Fig. 5(b) for the Bugs.jar datasets). These are aggregated results, as we have combined data for all the projects in a specific dataset. Results suggest that the median values of all code metrics, except for the Readability and Maintainable Index, in the buggy methods are larger than the median or Q3 (75th percentile) values in the non-buggy methods. Not surprisingly, for the Maintainability Index and Readability, the median of buggy methods is smaller than the median or Q3 (75th percentile) values in the non-buggy methods: less readable and maintainable code is more bug-prone. Evidently, the buggy methods are generally larger (LC), more complex (e.g., McCabe, McClure), and less readable than the non-buggy methods. These figures imply that buggy and non-buggy distributions are different regarding all code metrics.

After applying the Wilcoxon Rank Sum Test to find if there is a statistical difference between these two distributions, we found that all of our comparisons for code metrics between buggy and non-buggy methods are statistically significant ($P \leq 0.5$). Therefore, the null hypothesis is rejected which means that these metrics are able to distinguish between buggy code and non-buggy code. We provided

Table 4
Cliff's Delta Effect sizes for comparing code metrics between buggy and non-buggy methods. S refers to Small and M refers to Medium effect size.

Dataset	LC	PI	MA	NBD	ML	D	MI	FO	R	E
Defects4J	M	S	M	S	M	S	M	M	S	M
Bugs.jar	M	S	S	S	S	S	M	M	S	M

the Cliff's delta size values in Table 4 which shows that effect size values are small or medium in all cases. In particular, LC, FanOut, Maintainable Index and, Halstead Effort exhibit better performance than other metrics, because for them the effect size is medium in both datasets.

With aggregated analysis, however, we cannot observe if our results are true for all of the projects. The aggregated data can be highly influenced by a few large projects. Also, different external factors, such as code review policy (Wang et al., 2019), developers' commit patterns (Herzig and Zeller, 2013), and expertise (Matter et al., 2009), can impact the distribution of code metrics and bug-proneness of a particular project. Therefore, we now reproduce the results for each project individually.

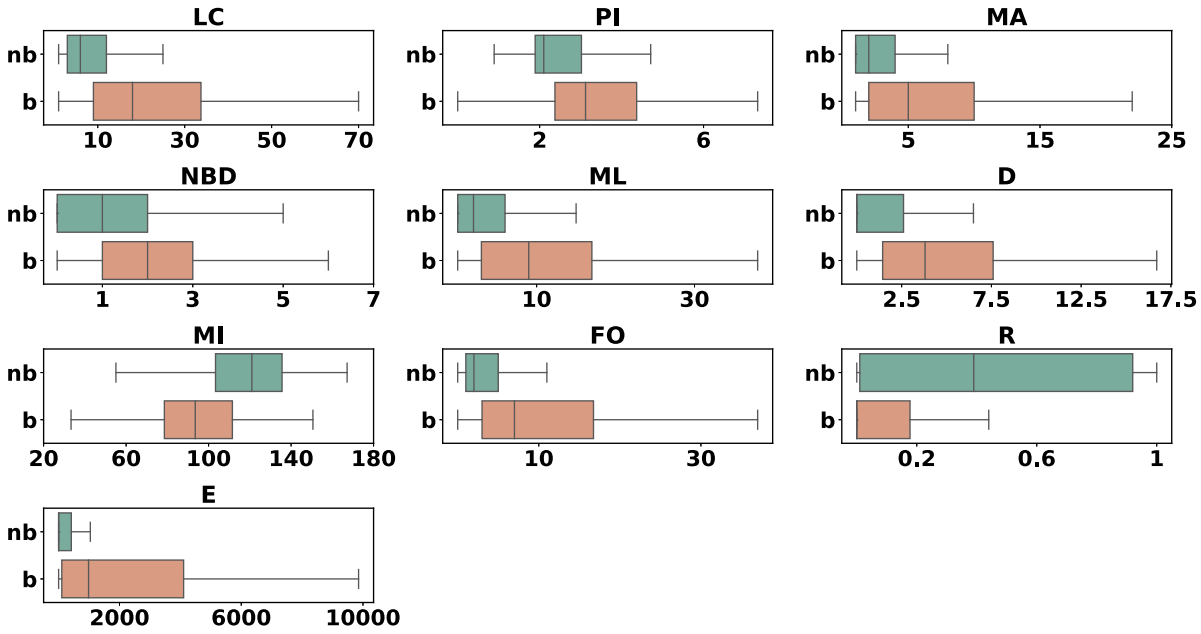
We calculated the percent of projects where the distributions of a particular metric are statistically different between buggy and non-buggy methods. Table 5 shows the results for all the code metrics. The distribution of LC, Maintainable Index, and FanOut are statistically different between buggy and non-buggy methods in all the studied projects from both datasets (100%). In the Defect4J dataset, the proxy indentation (PI) metric has the weakest performance since in 50% of the projects, the result is not significantly different. Most of the metrics, however, show good performance in distinguishing the buggy method from the non-buggy method. We also calculated Cliff's delta effect sizes of the differences, as presented and described in Table 6. The results show that Lines of Code, Maintainable Index, Fan Out, and Readability metrics do not exhibit negligible effect size for both datasets, but Proxy Indentation has the largest percentage of negligible effect size in both datasets.

Summary of RQ1.1 Results: Our selected code metrics show high performance in identifying the buggyness of methods in both datasets (with both aggregated and individual project analysis). Line of Code, Maintainable Index, Fan-out, readability, and Effort metrics have the best performance while Proxy Indentation and Nested Block Depth exhibit very poor performance.

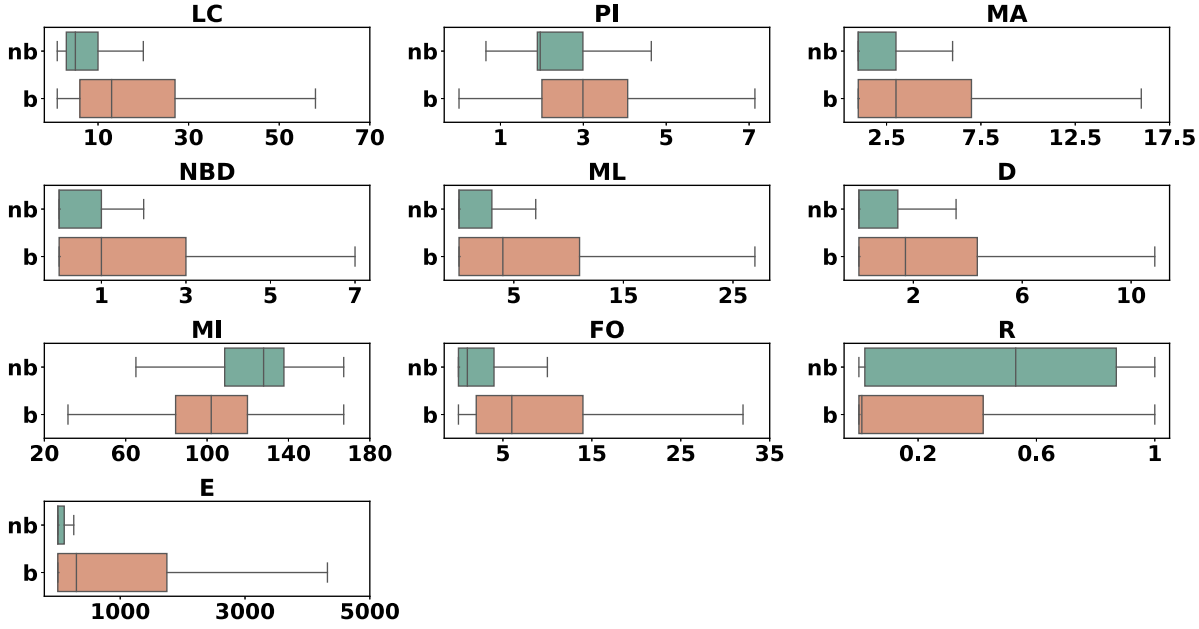
RQ1.2 Answer: After observing the somewhat known power of code metrics in distinguishing buggy and non-buggy methods, in this sub-RQ, we show their power in estimating the bug's severity. Although we wanted to perform the analysis for each project separately, there is not enough data in each severity category, except for `accumulo` and `jackrabbit-oak` projects. The `accumulo` project has 44, 100, and 87, and the `jackrabbit-oak` project has 71, 439, and 80 samples in Critical, Major, and High categories (we described in Section 2.1) respectively. The number of samples for each category in other projects was much lower, so we did not report their statistical test results separately.

Fig. 6 compares the distributions of different code metrics after grouping the bugs into different bug severity categories. Clearly, the usefulness of code metrics in distinguishing bug severity is not convincing. To provide better insights, we also show the Wilcoxon Rank-Sum test and Cliff's Delta size values for both datasets in Table 7. In the table, the first two major rows (for Defect4J, and Bugs.jar) show results for the aggregated analysis, and the last two rows show results for the two individual projects: `accumulo` and `jackrabbit-oak`.

In contrast to the previous section, in many cases, the results are not significantly different between different bug severity categories.



(a) Defects4J



(b) Bugs.jar

Fig. 5. Comparing source code metrics between buggy methods (b axis) and non-buggy methods (nb axis) using aggregated dataset.

Table 5
Wilcoxon Rank-sum test results to show the percentage of projects where the distribution of different code metrics are statistically different between buggy and non-buggy methods.

Dataset	LC	PI	MA	NBD	ML	D	MI	FO	R	E
Defects4J	100%	50%	70%	80%	80%	70%	100%	100%	90%	80%
Bugs.jar	100%	87.5%	100%	100%	100%	100%	100%	100%	100%	100%

Considering the aggregated analysis in both datasets, only Halstead difficulty, and effort show desired behavior: methods with higher severity bugs have higher difficulty and effort distribution. The Maintainability index (MI) performs very poorly: only in two cases, it exhibits statistically different distributions, but with negligible effect size. Readability shows the desired behavior for the Bugs.jar dataset when the comparison was done between critical and major, and critical and minor. The statistically different distributions among these groups suggest that a method with lower readability has a higher chance of having more severe bugs. Although source lines of code (LC) were very helpful in differentiating between buggy and non-buggy methods (Fig. 5(b), Tables 5 and 6), its performance is significantly worse when it comes to distinguishing different levels of bug severity. This is interesting because numerous previous studies (e.g., El Emam et al., 2001; Gil and Lalouche, 2017; Sjøberg et al., 2013) have shown that size (source lines of code) is the most important code metric, and none of the

Table 6
Cliff's Delta Effect sizes after comparing buggy vs non-buggy methods. N refers to Negligible, S refers to Small, M refers to Medium, and L refers to Large effect size. For example, the first row compares the LC distributions between buggy and non-buggy methods and shows the percent of differences with negligible, small, medium, and large effect sizes. Noticeably, the values for negligible (N) are zero for both datasets, implying that the difference in LC distributions between buggy and non-buggy methods is never negligible. This observation is also true for MI, FO, and R.

Metrics	Defects4J				Bugs.jar			
	N	S	M	L	N	S	M	L
LC	0	40	30	30	0	25	62.5	12.5
PI	30	50	10	10	37.5	62.5	0	0
MA	20	40	20	20	12.5	37.5	50	0
NBD	20	50	10	20	12.5	75	12.5	0
ML	10	30	40	20	0	62.5	37.5	0
D	20	40	30	10	12.5	37.5	50	0
MI	0	30	40	30	0	25	62.5	12.5
FO	0	40	40	20	0	37.5	50	12.5
R	0	50	40	10	0	87.5	12.5	0
E	10	30	30	30	0	37.5	62.5	0

other code metrics provide any new information if their correlation with size is normalized. Our result challenges that claim because we find the superior performance of other code metrics (e.g., Halstead difficulty, and Halstead effort) than size when distinguishing methods with different bug severity.

In many cases, there is no statistical difference between these code metrics when compared against bug severity. It means that there are some cases where the code is complex which makes it error-prone, but severe bugs are not necessary in the complex code. We may intuitively say that there may be bugs in a simple method containing an SQL query for reading data from a database. In such a method, the code complexity metrics suggest a simple method, but if the developers do not handle input validation, the code will be vulnerable to SQL Injection. On the other hand, we may have a very complex and big method containing nested loops with different switch cases and exception-handling statements. The code complexity metrics would suggest it is a complex method, but this method may try to handle the UI/GUI part of the system only, which is not vulnerable to any critical or high-severity bugs. We will explore the real examples from our datasets regarding our intuition in RQ3.

Summary of RQ1.2 Results: None of the selected code metrics show promising results in distinguishing bug severity. While Difficulty and Effort metrics have the best performance in the aggregated datasets, the Maintainable Index and Proxy indentation metrics have the weakest performance. Although the Line of Code metric has an excellent performance in distinguishing buggyness of the methods, it exhibits extremely poor performance in finding bug severity, in both the aggregated datasets and the individual projects.

3.2. Static analysis tools & bug severity (RQ2)

The second goal of our study is to find out if static analysis tools can detect the bugs and their severity labels.

3.2.1. Motivation

Since source code metrics do not show promising results in predicting the bug severity, we explore other approaches. To do so, we study the static analysis tools that are widely used in practice for different purposes (e.g., finding programming errors, coding standard violations, syntax violations, and security vulnerabilities) by technical teams. This approach is favorable because of its higher speed than the human code review and its capability to work offline with low required resources. Since these tools are already integrated into the deployment process of many companies, if we could use them for predicting the severity we can leverage them for prioritizing the bugs without requiring any other tools/methods. Therefore, in this RQ, first, we need to find out if these tools can detect real-world complex bugs, and then we study their performance in predicting the detected bug's severity.

3.2.2. Approach

We answer one RQ with its two sub-RQs to find the ability of the static analysis tools to detect bugs and their severity.

RQ2: What is the capability of static analysis tools in finding bugs and their severity?

- **RQ2-1**: How effective static analysis tools are in detecting buggy methods?
- **RQ2-2**: Can static analysis tools estimate detected bugs' severity?

Design of RQ2: To answer this RQ, we selected two static analysis tools named SpotBugs, SpotBugs (2023) and Infer Infer (2023). We selected SpotBugs because it not only can detect bugs but also report a Rank value which indicates the severity. We used the popular Infer tool only to detect buggy methods regardless of their severity values. Since there is no need for a statistical test in this RQ, there is no restriction on the number of samples in each group. Therefore, we use the USL values (mentioned in Section 2.1) directly without applying any merging.

SpotBugs has different configurations that affect its performance in finding bugs. It has some specific detector modules that focus on specific bug types, but we used the standard detectors, which is also the default option. Also, this tool has a configuration option named `effort` which adjusts the taken effort in finding bugs. We used the `effort=max`, which is the highest level of effort, so in this way, we have provided as much as the computation cost this tool requires to work on our datasets.

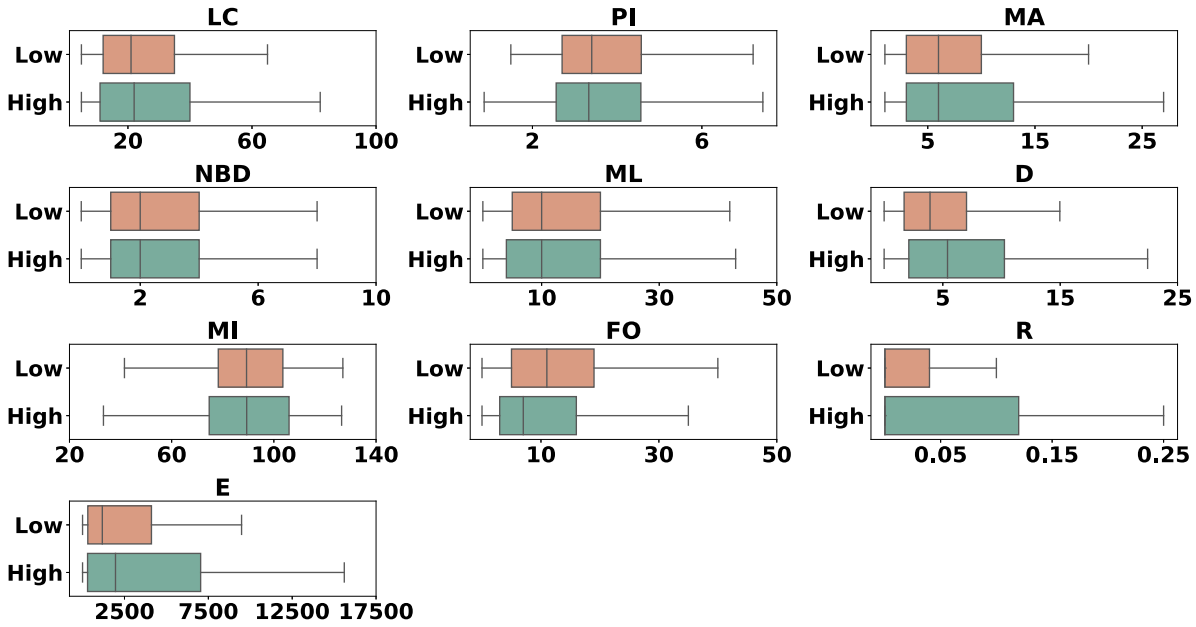
There are different ways to provide projects' source code to these tools, such as providing only the buggy class (es), the package containing the buggy class (es), or the whole project. By providing only the buggy class, the tool may not be able to find the bugs that are across different classes since it only analyzes that specific class, but when we provide the whole package or the whole project, it analyzes any file in these modules to find the bugs accurately which also needs more resources and time. We provided the whole project to be analyzed with tools to make the result more robust. However, this takes 3 and 52 h for the Defect4J and Bugs.jar dataset, respectively, on a regular computer with 16 GB RAM and 16 CPU cores.

Although we successfully applied these tools on all of the Defects4J projects, for several projects of the Bugs.jar dataset we faced build errors due to dependency issues. We were successful in building `flink`, `commons-math`, and `accumulo` projects which consist of 248 buggy methods.

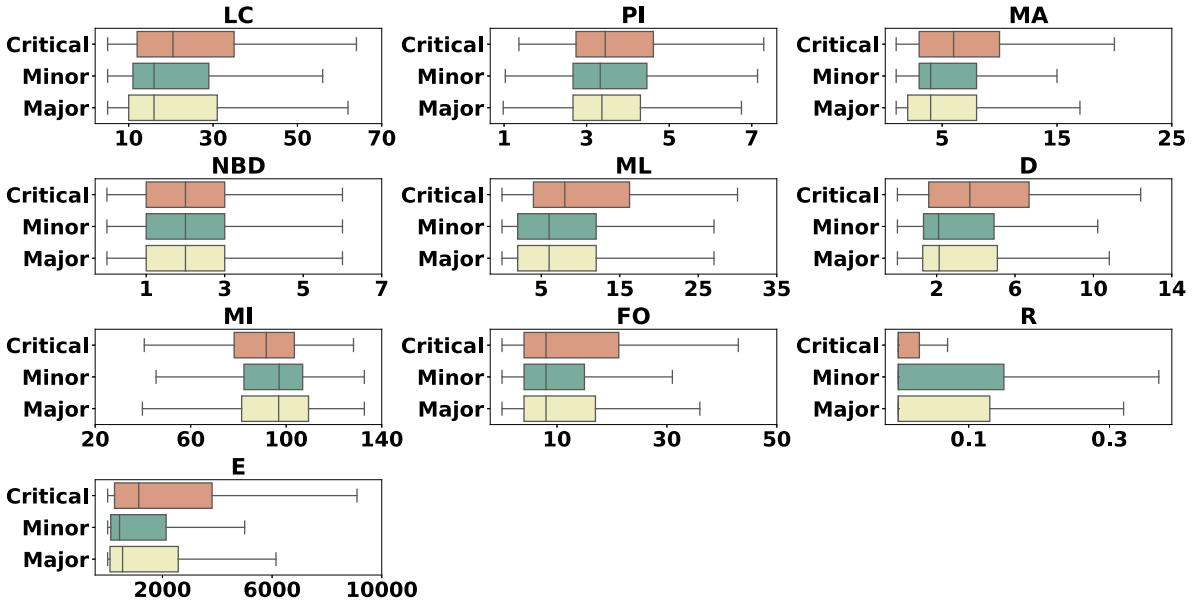
SpotBugs reports the `Confidence` value for each bug instance, which indicates the level of confidence the tool has in reporting this warning. This attribute may have different values such as 1 to match high-confidence warnings, 2 to match normal-confidence warnings, or 3 to match low-confidence warnings. We considered all of the reported warnings regardless of their confidence values.

Also, this tool reports the `Rank` property which indicates the bug rank.This tool categorizes bugs into 4 different levels; scariest, scary, troubling, and concern. The reported Rank property represents an integer value ranging from 1 to 20. Within this range, 1 to 4 corresponds to the scariest, 5 to 9 to scary, 10 to 14 to troubling, and 15 to 20 to bugs of concern (SpotBugs, 2023). The lower the number, the more important it is. This property has a similar semantic to the bug severity value, so we used this value to see the tool's capability in finding the bugs' severity.

The translation between the range of this value and our USLs is provided in Table 8. As the table shows, we consider 1 to 4 as Critical and Blocker USL, 5 to 9 as Major and High USL, 10 to 14 as Medium USL, and 15 to 20 as Minor, Low, and Trivial USL. For example, when



(a) Defects4J



(b) Bugs.jar

Fig. 6. Comparing source code metrics between different severity groups for Defects4J and Bugs.jar datasets.

the SpotBugs reports any number in the range of [1,4], and the actual severity value is either Critical or Blocker we say that it detects the severity correctly.

The Infer tool reports the `IssueType` attribute with these categories: (1) Error (2) Warning (3) Info, Advice, Like. These values can be interpreted as the bug importance, but we found that all of our bugs are mapped to the Error category, based on their severity values. Thus it is not possible to leverage the `IssueType` property to predict the bug severity, and we cannot use this tool to predict the severity. Therefore, we leverage this tool only to see if it can detect buggy methods, regardless of their severity values.

Both of these tools report the roots of detected bugs according to their defined patterns, by locating the start and end lines of the bug. Since we are considering bugs at the method-level granularity, we consider a method to be buggy whenever these tools report a bug in that method regardless of the reported start and end lines.

3.2.3. Results

RQ2-1 Answer: After applying SpotBugs and Infer tools we count the number of buggy methods that are reported as buggy (TP), the number of non-buggy methods which are not reported as buggy (TN), the number of buggy methods that are not detected by tools (FN), and the number of non-buggy methods which are reported as buggy methods (FP). If any of these tools report more than one bug for each method we only count that method once to prevent the duplication problem.

We calculated the Accuracy (ACC), Precision, Recall, and F1 values which are shown in Table 9. These values show that these tools have poor performance in detecting buggy methods of both datasets. While the accuracy value is so high, other metric values are relatively low which is because of the imbalanced dataset we have and accuracy is not the best metric for measuring performance for imbalanced datasets. The precision value range is 7%–9% which means that they may report

Table 7
Cliff's Delta Size for bugs with different severity values. A red cell indicates that the null hypothesis is rejected (i.e., the distributions are statistically significantly different according to the Wilcoxon test), whereas a blue cell means they are not statistically different. Effect sizes are N∼Negligible, and S∼Small. For example, the first cell shows that in the Defect4J dataset, the distribution of source lines of code (LC) between methods with high and low severe bugs is not statistically different (blue) and the difference has a negligible effect size.

Project	Distributions	LC	PI	MA	NBD	ML	D	MI	FO	R	E
Defects4J (all)	High-Low	N	N	N	N	N	S	N	S	N	N
Bugs.jar (all)	Critical-Major	N	N	S	N	S	S	N	N	N	S
	Critical-Minor	N	N	N	N	N	S	N	N	N	S
	Major-Minor	N	N	N	N	N	N	N	N	N	N
accumulo	Critical-Major	N	N	S	S	S	N	N	N	N	N
	Critical-Minor	S	S	S	S	S	S	S	S	N	S
	Major-Minor	N	S	N	N	N	N	N	S	N	N
jackrabbit-oak	Critical-Major	N	N	N	N	N	N	N	S	N	N
	Critical-Minor	N	S	N	S	S	S	N	N	S	N
	Major-Minor	S	S	S	S	S	S	N	S	N	S

Table 8
Translation between the SpotBugs reported rank (SRR) value and the unified severity label (USL) value.

USL	SRR
Critical, Blocker	1 to 4
Major, High	5 to 9
Medium	10 to 14
Low, Trivial, Minor	15 to 20

Table 9
Static analysis tools performance in finding buggy methods. The Infer tool requires the project under test to be compiled, which did not work for the Bugs.jar dataset due to different build systems and heavy dependencies.

Dataset	Tool	ACC	Precision	Recall	F1
D4J	SpotBugs	93%	7.1%	7.2%	7.1%
	Infer	95%	9%	1.9%	3.1%
Bugs.jar	SpotBugs	95%	7.6%	4.3%	5.4%

Table 10
SpotBugs performance in finding bug severity of Defects4J and Bugs.jar datasets.

Dataset	Accuracy	Precision	Recall	F1
D4J	0.94%	12.35%	1.25%	1.67%
Bugs.jar	0.03%	1.00%	0.04%	0.08%

many non-buggy methods as buggy methods, so the technical team may spend extra time on finding bugs in these methods, and this increases the cost of the maintenance phase without any useful result. However, the recall value (predicting buggy methods as non-buggy methods) is a more important metric in our case which has a smaller range of 2%–7%. This small value indicates that these tools miss many bugs which may lead to harmful consequences. The high accuracy with low precision and recall means that tools are making many Type I errors (false positives) and Type II errors (false negatives). The calculated F1 score shows that SpotBugs has a better performance in finding bugs in the Defect4J dataset than the Bugs.jar dataset, and the Infer tool has a worse performance than SpotBugs for the Defects4J dataset.

Our results are consistent with the previous study (Habib and Pradel, 2018) in 2018 where the authors studied the capability of SpotBugs, Infer, and ErrorProne tools in finding buggy methods of an older version of the Defects4J dataset. They concluded that 95.5% of buggy methods are not detected by these tools. Surprisingly, although these tools have been updated several times since four years ago, their performance in finding bugs is not improved much.

Since these tools match the provided code with their predefined generic bug patterns, we may intuitively say that in cases where the bugs are complex or related to the software specification, they are missed by these tools. We will perform a qualitative study of some randomly sampled bugs to find the reasons behind this poor performance, in RQ3.

Summary of RQ2.1 Results: Both SpotBugs and Infer tools have significantly low performance in finding bugs even with many existing developed patterns. The precision range for both datasets is 7%–9% and the Recall range is 2%–7%, which implies the extremely poor performance of these tools.

RQ2.2 Answer: Since only the SpotBugs tool can detect the severity of bugs, in this section we provide results only for this tool on both Defects4J and Bugs.jar datasets.

We provided the confusion matrices for the SpotBugs tool in Fig. 7. These figures show that this tool labels most of the bugs with wrong severity labels. For example, it detects two critical bugs as low-severity bugs and 20 high-severity bugs as low-severity bugs, in the Defects4J dataset. These figures show that SpotBugs did not detect the severity of any Critical bugs in the Defect4J and Bugs.jar datasets.

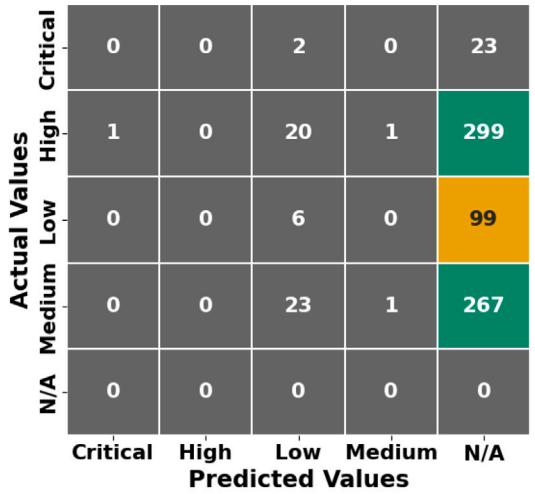
Table 10 shows different descriptive statistics of this tool's performance. Since we have multi-class labels, we used `macro average` (Scikit-Learn, 2007-2023a) for calculating these values. In this method, the final result is computed by taking the arithmetic mean of all the per-class descriptive scores.

In this table we observe small accuracy values for both datasets which show their weakness in finding the correct bug severity. Although the accuracy metric is not a good metric for imbalanced datasets, we report this value only for the sake of completeness and we do not use this metric for any comparison or conclusion. For the mentioned reason we report other metrics such as precision, recall, and F1 which are suitable for imbalanced datasets. The F1 score has been used in this paper to value the performance of these tools and accuracy is reported for completeness. The F1 score shows that although SpotBugs exhibits better performance for the Defect4J dataset, its performance is still very poor. We studied bug characteristics manually in RQ3 to find the reasons for this weak performance.

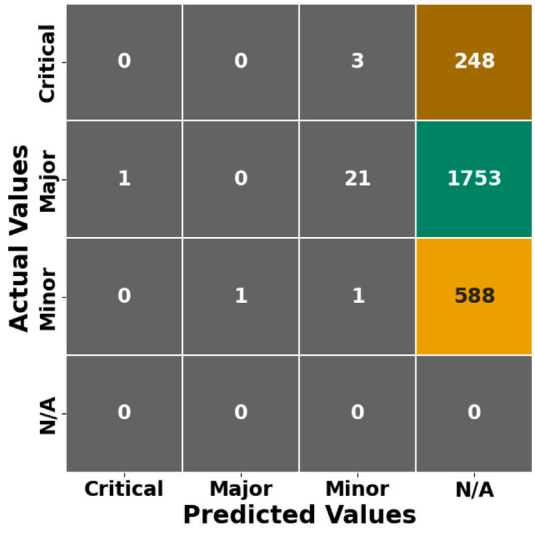
Summary of RQ2.2 Results: SpotBugs has significantly better performance in finding the bug severity of the Defects4J dataset than the Bugs.jar dataset. However, this tool wrongly assigns lower severity values to bugs in many cases. The overall performance shows that there is still much work to be done for this tool to accurately estimate bug severity.

3.3. Manual analysis of bug characteristics (RQ3)

Our final goal, in RQ3, is to dig more into the results of RQ1 and RQ2 and study the bug characteristics in different important scenarios



(a) Defects4J



(b) Bugs.jar

Fig. 7. Confusion matrices of bug severity prediction by SpotBugs for Defects4J and Bugs.jar datasets. In this figure, the 'N/A' column shows the number of bugs that were not detected by the tool. Therefore, in the first row of (a) Defects4J, there are 23 critical bugs in this dataset that were not detected.

such as (a) when there is a contradiction between code metrics and severity and (b) when the static analysis tools miss/mislabel severe bugs.

3.3.1. Motivation

To understand why source code metrics cannot indicate the bug severity, we need to study the cases where there is a contradiction between the bug severity and source code metrics. Also, it seems important to find out if there is any relationship between the bug's type and its severity. The answer to this question can help researchers to consider bug types as important features when providing new approaches for bug prediction to have better accuracy, and also developers may pay more attention to specific bug types during the manual severity assignment process.

To find out why static analysis tools miss many severe bugs or predict lower severity than the actual severity, in most cases, we need to study if this problem exists because of the nature of static analysis tools or if there is potential for improvement. The answer to this question will help researchers to improve the developed approaches to enhance the bug severity performance and also developers can get insights into how confidently they should use these tools in their projects.

Table 11
Two sample bugs from Severe and Non-Severe categories with their code metric values, normalized values, and the Sum Complexity values. The value in each row and the value of Sum Complexity are rounded to two precision.

Metrics	Severe		Non-Severe	
	Raw	Normalized	Raw	Normalized
LC	13.00	−0.35	532.00	21.27
PI	2.89	−0.25	8.61	2.75
MA	3.00	−0.37	194.00	23.50
NBD	1.00	−0.33	12.00	3.33
ML	2.00	−0.53	219.00	13.93
D	1.86	−0.39	18.84	2.20
FO	4.00	−0.35	40.00	22.14
E	220.14	−0.27	214 131.00	45.50
MI	−103.93	−0.53	23.86	4.15
R	−0.22	−3.66	0	0
Sum Complexity	143.74	−7.07	215 459.26	138.80

3.3.2. Approach

We answer one RQ with two sub-RQs to study the bug characteristics.

RQ3: What are the characteristics of bugs with different severity values?

- **RQ2-1**: How do code metrics perform in distinguishing bug severity for different types of bugs?
- **RQ2-2**: Why do static analysis tools miss or mislabel severe bugs?

Design of RQ3-1: As we show in the RQ1 results section, there is a correlation between bug existence and code complexity metrics, but there is no significant correlation between bug severity and code complexity metrics. Therefore, we manually investigated a random sample of bugs to find factors that lead to contradictions between bug severity values and code metrics. The goal of this sub-RQ is to study the characteristics of buggy methods with high severity (e.g., Blocker, Critical, Major) which have low code metrics measurement, and to study buggy methods with low severity (e.g., Low, Minor, Trivial) which have high code metrics measurement.

To facilitate the manual analysis, we split all bugs into two high-level categories of severity, which are: severe (including Blocker, Critical, Major, and High severity USLs from RQ1) and non-severe (including Medium, Low, Minor, and Trivial USLs from RQ1).

To find the two lists of the most and the least complex methods, we calculate a representative sum complexity value from different code metrics. Since our code metric values do not have the same range, we normalize them by using the Robust Scaler algorithm (Scikit-Learn, 2007-2023b), which is robust against outliers. It follows a similar algorithm to the MinMax scale, but it uses the interquartile range rather than the min–max. The formula for Robust Scaler is shown in Eq (4):

$$RobustScaler = \frac{x_i - Q_1(x)}{Q_3(x) - Q_1(x)} \tag{4}$$

Where $Q1$ is the 1st quartile, and $Q3$ is the third quartile.

To find the Sum Complexity (SC) value—which indicates the total complexity of the method—we take the sum of different code metrics (we multiply Readability and Maintainable Index by −1 to reverse their value). We provide the same weight to all the code metrics and the possible impact of this decision is discussed in the Threats to Validity section. Table 11 shows two sample bugs with the calculated code metrics, the normalized code metric, and also the Sum Complexity values.

After calculating the SC value, to find the contradiction, we sorted our samples based on the SC (normalized version) value in ascending

Table 12
Distribution of the randomly sampled bugs in the nine low-level bug labels.

Low-level Label	Number
Wrong/missing objects and parameters	29
Input/state checking	21
Completely wrong implementation	8
Exception handling	7
Null checking	5
Math/String calculation	4
I/O	3
Multi-threading	2
UI	1

and descending orders separately and kept the first 80 samples in both cases for both datasets which resulted in 360 samples (80 $*$ 2 (Severe | Non-Severe) $*$ 2 (Defects4J | Bugs.jar)).

Then, for the manual analysis, we randomly select 80 bugs out of these 360 bugs distributed evenly between Severe and Non-Severe bugs (our two categories in this RQ) and also between Defects4J and Bugs.jar (our datasets). Then, we provide the buggy code and the corresponding fixed code of these samples to the first and third authors.

We use a Card Sorting approach (Miles and Huberman, 1994) to label the bug type manually. In this approach, we asked the first and third authors to label each buggy method with one or more "bug types" of their choice, based on their own defined list of bug types (e.g., Network, UI/GUI, IndexOutofBound, Math calculation), by looking at the buggy code and the fixed code only. The total unique labels created by the two authors in this phase were nine low-level labels (mentioned in the 3.3.3 section).

Then, the first and third authors had a meeting to discuss their assigned labels and merged the nine lower-level labels into the final set of four higher-level labels. During the merging process, while authors agreed on most of the samples' labels, several assigned labels had similar meanings but were expressed in different terms. We resolved those conflicts by assigning a more generic label that covers both original labels. For example, there is a sample where the first author assigned "I/O bug", as the label, but the third author assigned "Network bug". We merged these two labels in the I/O bug label which covers the Network bug too. Following this approach, we ended up with the following four labels: Integration, Edge/Boundary, Security, and Specification.

Design of RQ3-2: Based on RQ2 results, we found that the SpotBugs tool missed or mislabeled bug severity in many cases, which may lead to undesirable consequences. For example, mislabeling the high-severity bugs with low severity values will reduce the bug's importance. Thus the development team may postpone the fixing process. Therefore, in this sub-RQ, we want to find out the reasons behind why these tools are not able to find the bugs or their severity values.

To do so, we first manually analyze 30 randomly chosen severe bugs (Blocker, Critical, Major, and High severity categories) in both datasets that are missed by the static analysis tools. In this step, we mainly want to know if the lack of bug patterns in static analysis tools prevents them from detecting these bugs, or if these bugs' nature is such that static analysis tools cannot detect them regardless of what pattern/rule they implement.

We then study a set of 30 randomly selected samples of severe bugs (Blocker, Critical, Major, and High severity categories) that are detected by SpotBugs tools, but the reported severity is lower than the actual severity values, e.g., when the actual severity value is Critical, but the tools report the Low severity value.

3.3.3. Results

RQ3-1 Answer: After applying the Card Sorting approach which is explained in we found nine low-level bug labels such as UI (User Interface) bugs, multi-threading bugs, and math calculation bugs.

Table 13
Distribution of the four higher-level bug labels where there are contradictions between severity values and Sum Complexity (SC) values.

High-level Label	Severe Bug & Low SC	Non-severe Bug & High SC
Specification	26	28
Edge/Boundary	4	9
Security	7	3
Integration	3	0

Table 12 shows bug types and their distributions. **Wrong/missing objects and parameters type** refers to code where the developer passes a wrong parameter to a method or uses the wrong object to call. **Input/state checking** bugs happen when the code does not validate the function inputs or it does not consider the specific case of an object state for having different logic. **Completely wrong implementation** bug type shows cases where the fixed code is changed completely to implement new logic compared to the buggy code. **Exception handling** bugs occur when the method does not handle exceptions when a variable/object has a special value. **Null checking** bugs happen when the code accesses the null object or does not handle special specifications when the method argument is null. **Math/String calculation** bugs occur when the mathematical calculations or string manipulations are performed wrongly. **I/O** bugs occur in a method where the code accesses any I/O such as a file or network. **Multi-threading** bugs are due to not handling threading issues such as deadlock. Finally, **UI** bugs are related to improper implementation of the user interfaces including the command line interface and graphical.

Table 13 shows the distributions of the four higher-level bug labels by grouping the contradictions — severe bugs but low Sum Complexity (SC), and non-severe bugs but high Sum Complexity (SC). The number of Security faults is higher when the bugs are labeled as severe, but the code metric values are low which indicates that security bugs may appear in simple methods but may lead to severe bugs. Furthermore, Edge/Boundary faults are mostly available in samples with high Sum Complexity (SC) values where the code is complex by having many different branches and statements where developers tend to miss edge/boundary cases, but these bugs are not severe necessarily.

In the following, we will provide definitions for each of these four bug-type categories and give some samples per category. Just a reminder that these categories have been defined based on our dataset and are the result of merging the labels from the Card Sorting phase.

Specification faults: This category, which is our largest category, exhibits bugs where developers have not implemented the full specification (e.g., the logic related to a partition in the input space has been ignored, i.e., misrepresented). This bug type is called Specification faults since they may not occur if the developers follow the specification of the software completely (Hemmati, 2015). Patches for these bugs contain changing conditional statements, math calculations, string manipulation, or a complete rewrite of the code.

For example, there is a bug with a Low severity value in `Closure-119` project where the buggy method is too complex (containing 120 lines of code having many nested `switch-case` and conditional statements), but the patch is very simple which handles a small part of input space. Part of this method and its patch are shown in Fig. 8. Long methods such as our mentioned example are usually bug-prone, and they are detected as buggy by code metrics correctly, but the bug could have a Low severity value since these long and complicated methods may handle a non-sensitive task such as handling the GUI part.

Also, Fig. 9 shows part of a buggy method from the `Lang30` project which has a Low severity value. In this sample, the patch is not as simple as in the previous example, and part of the code is rewritten, but the severity is still low.

The number of Specification faults in both severe and non-severe categories is almost equal as shown in Table 13, so further information is strongly required to detect the bug severity in this case.

```
        case Token.ASSIGN:
            if (parent.getFirstChild() == n) {
                isSet = true;
                type = getValueType(n.getNext());
            }
            break;
        case Token.GETPROP:
            return;
        case Token.FUNCTION:
            Node gramps = parent.getParent();
            if (gramps == null ||
                NodeUtil.isFunctionExpression(parent)) {
                return;
            }
            isSet = true;
            type = Name.Type.FUNCTION;
            break;
    +   case Token.CATCH:
        case Token.INC:
        case Token.DEC:
            isSet = true;
            type = Name.Type.OTHER;
            break;
        default:
```

Fig. 8. Sample bug with the Low severity value in the Closure-119 project of the Defects4J dataset.

```
   char ch = cs.charAt(i);
   for (int j = 0; j < searchLength; j++) {
      if (searchChars[j] == ch) {
-        if (i < csLast && j < searchLast &&
-           ch >= Character.MIN_HIGH_SURROGATE &&
-           ch <= Character.MAX_HIGH_SURROGATE) {
- // missing low surrogate, fine, like String.indexOf(String)
-            if (searchChars[j + 1] == cs.charAt(i + 1)) {
+        if (Character.isHighSurrogate(ch)) {
+           if (j == searchLast) {
+ // missing low surrogate, fine, like String.indexOf(String)
+              return true;
+            }
+            if (i < csLast &&
+                searchChars[j + 1] == cs.charAt(i + 1)) {
                   return true;
              }
            } else {
               // ch is in the Basic Multilingual Plane
              return true;
              }
           }
       }
   }
   return false;
}
```

Fig. 9. Sample bug with the Low severity value in the Lang-30 project of the Defects4J dataset.

Edge/Boundary cases: This kind of bug happens when the developers do not handle edge/boundary cases properly. Multiple samples in this category contain severe bugs, but with simple code according to our code metrics, containing only a few statements. For example, Fig. 10 shows the buggy method in the `Math-36` project, where the

```
public double doubleValue() {
    double result = numerator.doubleValue() /
    denominator.doubleValue();
+   if (Double.isNaN(result)) {
+   // Numerator and/or denominator must be out of range:
+   // Calculate how far to shift them to put them in range.
+       int shift = Math.max(numerator.bitLength(),
+       denominator.bitLength()) - Double.MAX_EXPONENT;
+       result = numerator.shiftRight(shift).doubleValue() /
+           denominator.shiftRight(shift).doubleValue();
+   }
    return result;
}
```

Fig. 10. Sample bug with High severity value in the Math-36 project of the Defects4J dataset.

```
@Override
public boolean isGranted(@Nonnull String path, long permissions) {
-   Iterator<PermissionEntry> it = getEntryIterator(
-   new EntryPredicate(path,
-   Permissions.respectParentPermissions(permissions)));
-   return hasPermissions(it, permissions, path);
+   EntryPredicate predicate = new EntryPredicate(
+       path, Permissions.respectParentPermissions(permissions));
+   return hasPermissions(getEntryIterator(predicate),
+   predicate, permissions, path);
}
```

Fig. 11. Sample bug with the Major severity value in the jackrabbit-oak-3324_5f863af6 project of Bugs.jar dataset.

developer did not handle the edge case (`Double.isNaN(result)`), but the patch shows that the studied method should have performed an extra calculation when the `result` variable is `NaN`. This bug may lead to a software crash, unauthorized access, or undesired behavior.

Another severe bug exists in the `accumulo` project where the buggy method renames a requested table. In this method, the code does not check if the requested table exists already or whether the new requested name is already taken by another table, so this bug probably leads to a software crash or overwriting the existing table.

We found that most of the Edge/Boundary bugs are not severe in our datasets, but they appear in complex methods. Measuring the bug consequences in the whole system would be required to find the bug impact and eventually the severity. Edge/Boundary case testing could be a possible solution to determine the bug impact based on the number of failed tests.

Security faults: Security faults can be exploited to gain unauthorized access/privileges by intruders. Since it is not easy to find if there is a security fault, solely from the source code, we found that some bugs are highly potential to be exploited by intruders by considering keywords and the method context.

For example, there is a bug with Major severity in the `jackrabbit-oak-3324_5f863af6` project which is shown in Fig. 11. This simple (based on the code metric values) method checks whether the requested path argument has access permission to be used. Handling permissions incorrectly may potentially lead to unauthorized access, and this usually has harmful consequences such as data loss or breaking the system completely.

After investigating all the samples in this category, we found that Security faults are severe (Blocker, Critical, Major, High) in most cases. However, these bugs are available in simple methods (based on our code metrics) that handle a sensitive task which may lead to harmful consequences and result in severe bugs.

Although we found most of our Security faults happen in simple methods, more specific methodologies such as security testing are

```
 protected Object functionFloor(EvalContext context) {
     assertArgCount(1);
     double v = InfoSetUtil.
     doubleValue(getArg1().computeValue(context));
+    if (Double.isNaN(v) || Double.isInfinite(v)) {
+        return new Double(v);
+    }
     return new Double(Math.floor(v));
 }
```

Fig. 12. Sample bug with High severity value in the JxPath-14 project of the Defects4J dataset.

required to find these bugs' severity values. If we can find the security fault type (e.g., Injection, Data Exposure, Broken Access Control, etc.) the bug severity can be found easily since security fault types typically have a specific ranking based on their probable consequences in a few defined standards.

Integration faults: This bug label exhibits cases where the bug is related to another component, module, or class that our buggy method is using, so it would be impossible to access the buggy method without considering the called module(s). For example, Fig. 12 shows a buggy code and its corresponding fixed code in the `JxPath-14` project. The buggy method seems to be very simple containing only three statements, so based on the code metric values this is not a complex code, but the actual severity value is high. The bug is related to the `floor()` method which is from another module (`math`). It seems that this module may not handle `NaN` and `Infinite` values or it may handle them differently than the studied project's (`JxPath`) specification. From the patch, we see that the developer needs to return a different value when the `v` parameter is `NaN` or `Infinite`, so the developer had to handle this case in the code before calling the `Math.floor()` function.

As another example, the buggy method in the `Cli-2` project uses the `NumberUtils.createNumber()` method to convert the variable with String type to the Number type, but the patch shows that the developer has rewritten the logic of converting String to Number inside the code without using the `NumberUtils` anymore. In this case, the called component may contain a bug or it may be developed with different specifications which does not satisfy the caller component's developer's requirements. Therefore, for detecting the severity value of this kind of bug accurately it is required to consider both caller and called module code together.

Since we found only three Integration faults, it is less reliable to conclude the severity of bugs in this category. However, our observations show that estimating the severity of faults in this category by only considering the caller method seems impossible. Thus an analysis in a coarser granularity (e.g., class/module/package) is suggested.

Summary of RQ3.1 Results: Most of our studied bugs were placed in the Specification categories, evenly, regardless of their severity values. Our studied bugs in the Edge/Boundary category are complex (high SC value) but mainly have a Low severity value. Security bugs have High severity, but they are mostly not complex (low SC value). Although we need more samples to confirm, our dataset indicates that bugs related to security are always severe.

RQ3-2 Answer: In this section, we provide the results of our manual investigation of severe bugs (Blocker, Critical, Major, High) that are not detected by static analysis tools and go through some examples of severe bugs that are miss-labeled by SpotBugs.

Fig. 13 shows a Critical bug in `Cli-2` project, which is not detected by SpotBugs and Infer because it cannot be fitted into any rule. The

```
 protected void burstToken(String token,
 boolean stopAtNonOption){
    int tokenLength = token.length();
    for (int i = 1; i < tokenLength; i++){
        String ch = String.valueOf(token.charAt(i));
        boolean hasOption = options.hasOption(ch);
        if (hasOption){
            tokens.add("-" + ch);
            currentOption = options.getOption(ch);
            if (currentOption.hasArg() &&
                (token.length() != (i + 1))){
                tokens.add(token.substring(i + 1));
                break;
            }
        }
        else if (stopAtNonOption){
            process(token.substring(i));
        }
        else{
-           tokens.add("-" + ch);
+           tokens.add(token);
+           break;
        }
    }
 }
```

Fig. 13. Critical bug in the Cli-2 project of the Defects4J dataset.

```
public double density(final double[] vals) throws
   DimensionMismatchException {
   final int dim = getDimension();
   if (vals.length != dim) {
       throw new DimensionMismatchException(vals.length,
       dim);
   }
-  return FastMath.pow(2 * FastMath.PI, -dim / 2) *
+  return FastMath.pow(2 * FastMath.PI, -0.5 * dim) *
          FastMath.pow(covarianceMatrixDeterminant, -0.5) *
          getExponentTerm(vals);
}
```

Fig. 14. Critical bug in the Math-11 project of the Defects4J dataset.

provided patch seems to be simple, but the bug is too complex since it is related to a specific case where the developer should use `token` variable and also the `break` statement in the last `else` branch to handle a special case required by the specification.

Another example is a Critical bug in the `Math-11` project which is shown in Fig. 14. The patch is very simple, and the buggy statement looks almost identical to the fixed statement such that it is even very hard for a human to find the issue without knowing the special values of `dim` variable that leads to the bug.

Fig. 15 shows the High severity bug in `Times-27` project where the `PeriodFormatter` class is not working well and the fix is as simple as having the `null` checking before the logic. While the fixed code is only one line and simple, it is hard for static analysis tools to detect this since we may have different logic based on the nullability of objects and there is no general error pattern because of this.

The `Compress-29` bug which has a high severity is shown in Fig. 16 where the polymorphism is applied with the method overloading approach. Although this is the standard approach to change the behavior of code in compile-time, the decision to use a particular overloaded function completely depends on the other lines of the code.

```
 if (size >= 2 && elementPairs.get(0) instanceof Separator) {
     Separator sep = (Separator) elementPairs.get(0);
+    if (sep.iAfterParser==null && sep.iAfterPrinter==null) {
         PeriodFormatter f = toFormatter(
         elementPairs.subList(2,size), notPrinter, notParser);
         sep = sep.finish(f.getPrinter(), f.getParser());
         return new PeriodFormatter(sep, sep);
+     }
 }
```

Fig. 15. High-severity bug in the Times-27 project of the Defects4J dataset.

```
+ if (entryEncoding != null) {
+   return new JarArchiveOutputStream(out, entryEncoding);
+   } else {
         return new JarArchiveOutputStream(out);
+   }

  if (CPIO.equalsIgnoreCase(archiverName)) {
+      if (entryEncoding != null) {
+        return new ArjArchiveInputStream(in, entryEncoding);
+      } else {
           return new ArjArchiveInputStream(in);
+      }
 }
```

Fig. 16. High-severity bug in the Compress-29 project of the Defects4J dataset.

```
    public double getSumSquaredErrors() {
-     return sumYY - sumXY * sumXY / sumXX;
+     return Math.max(0d, sumYY - sumXY * sumXY / sumXX);
    }
```

Fig. 17. Major bug in the Math-105 project of the Defects4J dataset.

```
    if (cs1 instanceof String && cs2 instanceof String) {
        return cs1.equals(cs2);
    }
-   return CharSequenceUtils.regionMatches(cs1, false, 0,
-           cs2, 0, Math.max(cs1.length(), cs2.length()));
+   return cs1.length() == cs2.length() &&
+           CharSequenceUtils.regionMatches(cs1, false, 0,
            cs2, 0, cs1.length());
    }
```

Fig. 18. High-severity bug in the Codec-18 project of the Defects4J dataset.

There is no issue from the static analysis tools in this case since no patterns can fit into this situation.

We found other severe bugs that cannot be generalized in the bug patterns such as a bug in the `Math-105` project shown in Fig. 17 where the `getSumSquaredErrors()` in the buggy code returns the maximum number between the error value and zero wrongly. The patch shows that it should return the calculated error value only (there is no need for the max function).

Also, `Codec-18` project (shown in Fig. 18) contains a bug where the `equals` method checks if two provided `CharSequence` objects are equal. The patch shows that the developer implemented the equality condition wrongly by not considering the maximum length of these strings which are patched in the fixed version.

Another example is the `FLINK-3011` major bug in the bugs.jar dataset (shown in Fig. 19) where the state of a running job is not handled comprehensively, so it cannot cancel a failing/restarting job. Since state handling is one of the most complex parts of every application,

```
+ else if (current == JobStatus.FAILING) {
+        if (transitionState(current, JobStatus.CANCELLING)) {
+                return;
+          }
+   }
+ else if (current == JobStatus.RESTARTING) {
+    synchronized (progressLock) {
+        if (transitionState(current, JobStatus.CANCELED)) {
+                postRunCleanup();
+                progressLock.notifyAll();
+                LOG.info("Canceled during restart.");
+                return;
+            }
+        }
+ }
```

Fig. 19. Major bug in the FLINK-3011 project of the Bugs.jar dataset.

```
    public int compareTo(Fraction object) {
-     double nOd = doubleValue();
-     double dOn = object.doubleValue();
+     long nOd = ((long) numerator) * object.denominator;
+     long dOn = ((long) denominator) * object.numerator;
      return (nOd < dOn) ? -1 : ((nOd > dOn) ? +1 : 0);
    }
```

Fig. 20. High-severity bug in the Math-91 project of the Defects4j dataset.

especially in multi-threaded applications our static analysis tools were not able to detect this bug based on their pre-defined patterns.

In all of the mentioned examples, we found that these bugs are so unique that they cannot be mapped to any of the predefined patterns. Detecting these severe bugs by only leveraging the source code (even at the coarse-level granularity such as class/package/module) is almost impossible. Therefore, more sophisticated patterns need to be implemented by static analysis tools. One may also try using dynamic approaches such as testing to detect these types of bugs.

Furthermore, we found some samples where there is a specific rule for the bug in the tools, but the bug is not detected by any of the tools. For example, the buggy method in the `Cli-9` project copies the values of a `mapping` variable (defined at the class level) to the provided argument of the method. The provided patch shows that there is a specific condition when the `mapping` variable is `null`, so the buggy method should return the provided argument instead of copying data. Although the SpotBugs tool has some predefined rules regarding the `null` handling of variables in the method it cannot report this method as buggy.

One possibility for this low effectiveness of static analysis tools is that since these tools are heavily used in production as a part of the development environment or building process, there is a high chance that all of these projects were scanned by these tools after each commit/release, so all bugs that can be detected by these tools have been fixed already.

By investigating the results of the SpotBugs tool to see why this tool mislabeled the severity of many bugs we found that 70 bugs are detected to have lower severity values than their actual severity value. After analyzing these samples, we found two scenarios. The first one would be samples that SpotBugs reported another bug in the buggy method than the actual bug. In the second case, Spotbugs reports the bug correctly, but it assigns the lower severity labels.

```
-    currEntry = new TarArchiveEntry(headerBuf);
+    try {
+         currEntry = new TarArchiveEntry(headerBuf);
+    } catch (IllegalArgumentException e) {
+         IOException ioe = new IOException(
+                   "Error detected parsing the header");
+         ioe.initCause(e);
+         throw ioe;
+    }
```

Fig. 21. High-severity bug in the Compress-12 project of the Defects4j dataset.

For example, the reported bug in the `Math-91` (Fig. 20) project has High severity, but SpotBugs reports Low severity. The actual reported bug is due to the equality problem of two `Fraction` objects because of the limited precision,[2] but the SpotBugs reports the `CO_COMPARETO_INCORRECT_FLOATING` as the issue type which is in the `BAD_PRACTICE` category with the following description:

"This method compares double or float values using a pattern like this: $val1 > val2\ ?\ 1\ :\ val1 < val2\ ?\ -1\ :\ 0$"

Although this statement exists in the buggy method, the actual reported bug is not happening because of this statement.

Another example is the bug in the `Compress-12` project where the actual reported bug is related to throwing the `IllegalArgumentException` instead of `IOException` on corrupted files.[3] This can lead to serious issues such as crash or wrong exception handling in different layers where this exception is being handled. The path for this bug is shown in Fig. 21. However, SpotBugs reported this method containing a Low-severity bug with the `I18N` issue type in the `DM_DEFAULT_ENCODING` category with the following description:

"Found a call to a method which will perform a byte to String (or String to byte) conversion, and will assume that the default platform encoding is suitable".

In the mentioned examples we found that SpotBugs reported a different bug with Low severity instead of the actual bug with High severity.

The `Math-100` project with the Critical bug crashes with `ArrayOutOfBounds` exception in the `getCovariances()` and `guessParametersErrors()` functions[4] but the SpotBugs tool reports bugs in this function with `Bx-DM_NUMBER_CTOR` pattern with the following description:

"Method invokes inefficient Number constructor; use static valueOf instead"

Although the reported bug from SpotBugs tools may apply to the buggy method, SpotBugs is finding another possible bug which is not the actual bug available in this project, so the reported severity from SpotBugs is lower than the actual severity. The partial actual fix for this bug is shown in Fig. 23.

Another High-severity bug in `Lang-58` project of Defects4J dataset exists where the reported bug is because of throwing a `NumberFormatException` when passing special values to the `createNumber` method and it cannot parse the input correctly.[5] The SpotBugs tool reports another bug for this function that points to a different possible bug that is not the same as the actual bug. SpotBugs report `SF-SF_SWITCH_FALLTHROUGH` with the following description:

"Switch statement found where one case falls through to the next case"

Although the reported issue by SpotBugs exists in the studied function, this is not the reason for the actual reported bug (the fix is shown in Fig. 24) and SpotBugs is reporting low severity since this bug pattern is defined to have a low severity generally by SpotBugs.

Another example is `accumulo-209_397f86f6` project with Major bug with the multi-byte character encoding issue[6] (part of the patch is shown in Fig. 22), and the SpotBugs reported a `DM_DEFAULT_ENCODING` as the issue type in `I18N` category with the following description:

"Found a call to a method which will perform a byte to String (or String to byte) conversion, and will assume that the default platform encoding is suitable".

It seems that SpotBugs has detected the bug correctly, but it assigns a low severity to it. Based on our best knowledge, SpotBugs assigns the severity of the reported bugs based on their implemented patterns, so it always reports Low severity for this kind of bug. However, in practice, one specific issue may lead to a severe or non-severe bug in different cases such as the number of affected users or the project timeline.

Summary of RQ3.2 Results: Based on our manual investigation, SpotBugs missed most of the studied bugs regardless of their severity value for two reasons. First, SpotBugs finds other possible bugs than the actual reported bugs. Second, there is no matching rule for the actual bugs, and SpotBugs reports another bug based on its available pre-defined bug pattern. In specific cases there are matching (almost the same) rules, however, the SpotBugs cannot detect their severity accurately. In this case, the reported severity is lower than the actual severity value since SpotBugs reports the severity value based on its general pre-defined bug patterns, but in practice, each specific bug may have different severity values based on different conditions such as the number of affected users or the project timeline.

4. Discussion on practical implications

We now summarize our findings and discuss how these findings can be useful in practice and bug prediction research.

In **RQ1**, we evaluated the effectiveness of code metrics in detecting bugs and their severity at the method-level source code granularity. Our results suggest that code metrics are good indicators of bug-prone methods. Our results are interesting because the true effectiveness of code metrics has been historically debated (Chowdhury et al., 2022a). While some studies claimed code metrics are useful to understand software maintenance (Johnson et al., 2019; Landman et al., 2014; Chowdhury et al., 2022a), such as finding or predicting bug-prone source code, other studies found code metrics useless (Shepperd, 1988; Gil and Lalouche, 2017). In recent studies, it was found that code metrics are useful at the method-level granularity, although they are not useful at the file/class-level granularity (Landman et al., 2014; Chowdhury et al., 2022a). This observation, however, does not match with the findings of Pascarella et al. (2020); the authors observed that code metrics were not useful in predicting bugs at the method level.

We also observed that although all the code metrics are helpful for bug detection, only a few of them help understand bug severity. Notably, when bug severity is considered, some code metrics — such as readability, Halstead effort, and difficulty — outperform the famous lines of source code (LC). This a very unique observation, as according to multiple research, LC is the metric that rules them all — LC always outperforms other code metrics in understanding maintenance

[2] https://issues.apache.org/jira/browse/MATH-252
[3] https://issues.apache.org/jira/browse/COMPRESS-178
[4] https://issues.apache.org/jira/browse/MATH-200
[5] https://issues.apache.org/jira/browse/LANG-300
[6] https://github.com/bugs-dot-jar/accumulo/tree/bugs-dot-jar_ACCUMULO-209_397f86f6

effort (Chowdhury et al., 2022a; Gil and Lalouche, 2017; Chowdhury et al., 2022b; El Emam et al., 2001; Sjøberg et al., 2013).

We conclude that practitioners should be aware of code metrics-induced code smells because we observed a clear difference in code metrics distribution between bug-prone and bug-free source code methods. Researchers should focus on producing more code metrics that are more geared toward bug severity estimation. We see potential future research where code metrics can be used in combination with large language models (LLMs) such as CodeBERT (Feng et al., 2020; Mashhadi et al., 2023) or GPT (OpenAI, 2023). Through word embeddings, LLMs can potentially find and suggest important tokens that often produce severe bugs.

In RQ2, we observed that existing static bug detection tools perform extremely poorly in detecting real-world bugs. These tools suffer from both false positives and false negatives, confirming earlier findings (Habib and Pradel, 2018; Thung et al., 2012). In a few cases, when these tools successfully captured real-world bugs, their estimation of bug severity was inaccurate—a unique finding of our study. We suggest that practitioners should be careful in relying on the existing static bug detection tools. In RQ3, we provided a qualitative analysis explaining why these tools fail in detecting bugs and estimating their severity. For example, we found that a less complex source code method that contains no bug patterns can contain severe bugs for containing security-related APIs.

We believe that our observations will be instrumental for future research on producing accurate bug severity prediction models. For example, based on our observation, we suggest future studies to combine static analysis and code metrics results to get the benefits of both. This is useful since the metrics and the tools seem to focus on different aspects of severity. By comparing the calculated SC (mentioned in) and the severity predicted by the SpotBugs tool we found some samples where using both of these results will be more useful. For example, in the `Math-87` project, the buggy method `computeShiftIncrement()` has High severity value. The SpotBugs tool reports a Low severity value, but the SC value shows that this method is a complex method regarding the code metrics we used. In this case, the code metric values exhibit the severity better. Conversely, there is a buggy method named `parse()` in the `Closure-68` project with a Medium severity value. The SC value for this buggy method is very large which shows the complexity of the method, so we may assume this is a severe bug, but the SpotBugs reports a Low severity value. In the mentioned example, the SpotBugs prediction is closer to the actual severity value than the code metric values.

We want to emphasize that although many of our findings are negative, it does not mean we do not need this nature of research. It is in fact as important to publish negative results for the community so that future researchers know what does not work and they do not spend more time on the same tools and techniques. We also have to point out that none of the findings are "obvious" before being empirically and systematically examined. Therefore, we think that identifying what did not work by itself is an important contribution to the research community.

5. Threats to validity

Similar to any empirical study there are some threats that may impact our results, so in this section, we provide possible threats and our actions to mitigate these threats.

5.1. External validity

Our selected projects may not be consistent with the closed-source projects or may not cover various software domains or other programming languages. We did our best to select two large and popular datasets containing real-world bugs of a diverse set of open-source projects that have been used in many studies (Martinez et al., 2017; Shamshiri et al., 2015; Pearson et al., 2017; Saha et al., 2017). The next threat would be the selection of static analysis tools, and for mitigating this problem we used the most popular tools which are used in practice and state-of-the-art research. We exclude Google Error Prone from the list of our tools since we found that it is not possible to handle older Java versions (which exist in our dataset) with the latest versions of this tool. Also, the selected code metrics may not reveal the potential of all available code metrics in finding bug severity and this may impact our result generalizability, but to mitigate this issue we tried to leverage as many as available method-level code metrics that are used in previous research for defect-related problems (e.g., defect prediction).

```
-    babcs.set(bs);
-    matcher.reset(babcs);
-    return matcher.matches();
+    try {
+      matcher.reset(
+        new String(bs.getBackingArray(), encoding));
+      return matcher.matches();
+    } catch (UnsupportedEncodingException e) {
+      e.printStackTrace();
+    }
```

Fig. 22. High-severity bug in the accumulo-209-397f86f6 project of the Bugs.jar dataset.

```
  int m = problem.getMeasurements().length;
- int p = problem.getAllParameters().length;
+ int p = problem.getUnboundParameters().length;
  if (m <= p) {
    throw new EstimationException("no degrees of freedom ({0}
    measurements, {1} parameters)",
            new Object[] { new Integer(m), new Integer(p)});
}
-  double[] errors = new double[
-                    problem.getAllParameters().length];
+  double[] errors = new double[
+                    problem.getUnboundParameters().length];
```

Fig. 23. Critical bug in the Math-100 project of the Defects4j dataset.

```
  if (dec == null && exp == null
-     && isDigits(numeric.substring(1))
-     && (numeric.charAt(0) == '-' ||
-     Character.isDigit(numeric.charAt(0)))) {
+     && (numeric.charAt(0) == '-' &&
+     isDigits(numeric.substring(1)) || isDigits(numeric))) {
        try {
          return createLong(numeric);
        }
```

Fig. 24. High-severity bug in the Lang-58 project of the Defects4j dataset.

5.2. Internal validity

Another threat would be our strategy in selecting the buggy methods since we selected all methods that are changed/removed during the bug fixing patch as a buggy method which is a common approach in the state-of-the-art approaches (referenced in 2.1). The reason behind this is that bug fixing usually requires changing multiple methods and specifying one single method to be the source of a bug requires other information such as a bug report description. To mitigate this threat we consider each of those methods as an independent sample, so code metrics and static analysis tools can consider them independently.

The next threat is the method of identifying reported bugs by static analyzers' tools. In this study, we consider the tools' report at the method-level granularity without considering the exact reported lines by these tools. We did this because there is no information regarding the exact bug location (line) in our datasets, and we want to keep the same method-level granularity for RQ1 and RQ2. To mitigate this threat whenever any of our tools report a bug inside the method, we count it as a correct detection to not underestimate the tools' performance.

5.3. Construct validity

For collecting ground truths about bug severity, we relied on practitioner-assigned labels, which can be inaccurate. Tian et al. (2016) observed that a duplicate bug report can have multiple severity labels. Practitioner-assigned severity labels, however, are perhaps the only way to understand bug severity and are widely adopted in bug severity research (e.g., Tian et al., 2012, 2013; Ramay et al., 2019; Lamkanfi et al., 2010). In our study, however, this threat was mitigated as we have merged similar severity for most of our analysis. That is, although different practitioners may assign different severity labels to the same bug report, the labels are still expected to be similar (e.g., critical and major, instead of critical and minor). When calculating the sum complexity for answering RQ3-1, we have provided the same weight to all the different code metrics. These can be inaccurate because some code metrics can be more important than others.

5.4. Conclusion validity

Our results may not generalize to other tools and datasets. Our results of static analysis tools may be affected by using projects that have already applied the studied static analyzer tools warnings during the software development life cycle. Since it is not easy to understand if it happened, our results should be considered as an assessment of those bugs that are not already fixed by tools. Our selection of code metrics may not be sufficient to expose the bug severity, but to mitigate this issue we considered the 10 popular metrics which have established performance in finding bugs in our study.

6. Related work

There has been much research in the bug-prediction field by focusing on different aspects using diverse granularity levels and various techniques (Hall et al., 2011; Hosseini et al., 2017; Wahono, 2015; Li et al., 2020). Previous studies can be categorized from different aspects such as bug reproducibility (Sahoo et al., 2010), working on the characteristics of source code (Khomh et al., 2012; Palomba et al., 2018; Bacchelli et al., 2010; Bavota et al., 2012; Pai and Dugan, 2007; Spadini et al., 2018; Rahman and Williams, 2019), or bug description (Sun et al., 2019; Wang et al., 2010; Khatiwada et al., 2018). From the source code granularity aspect, there has been research in file level (Zimmermann et al., 2007; Giger et al., 2011), package level (Zimmermann et al., 2007), class level (Shivaji et al., 2009), and method level (Giger et al., 2012). For bug severity prediction tasks, almost all of the existing works have leveraged bug reports using natural language techniques or different classic and deep neural network models. In the rest of this section, we briefly discuss the related work on generic bug prediction, bug severity prediction, and static analysis tools.

6.1. Bug prediction

Bug prediction studies have used different code metrics such as LOC (Pascarella et al., 2020; Antinyan et al., 2014; Shin et al., 2010; Chowdhury et al., 2022b), McCabe (Antinyan et al., 2014), Halstead metrics (Antinyan et al., 2017), C&K metrics (Subramanyam and Krishnan, 2003; Jureczko and Spinellis, 2010). Code metrics have been applied on different granularity levels such as package/class level (Okutan and Yıldız, 2014; Koru and Liu, 2005; Zimmermann et al., 2007), method level (Pascarella et al., 2020; Giger et al., 2012; Ferenc et al., 2020b; Chowdhury et al., 2022b; Grund et al., 2021; Mo et al., 2022), and line level (Wattanakriengkrai et al., 2020). One major problem with the high granularity levels (package/class) is that they are practically less helpful for the developers (Giger et al., 2012; Pascarella et al., 2020) since it requires significant effort to locate bugs at the package/class components. Unfortunately, line-level granularity can suffer from too many false positives, because multiple lines can be similar just by chance (Steidl et al., 2014; Servant and Jones, 2017). Consequently, method level granularity has been the new focus of the community (Pascarella et al., 2020; Giger et al., 2012; Ferenc et al., 2020b; Chowdhury et al., 2022b; Grund et al., 2021; Mo et al., 2022), especially for developing bug prediction models, and several studies show positive and encouraging results (Giger et al., 2012; Ferenc et al., 2020b; Mo et al., 2022).

6.2. Bug severity prediction

Due to time and budget constraints, practitioners often try to solve the severe bugs first. This, however, requires manually labeling the severity of the reported bugs which is time-consuming. Consequently, previous research has focused on predicting the severity of bug reports using different machine learning and natural language processing techniques.

Tian et al. (2012) leveraged bug reports and their severity labels in the past to recommend fine-grained severity labels for newly reported bugs by measuring the similarity between bug reports and using nearest neighbors classification. The authors improved the f-measure of the state-of-the-art approach significantly. In a separate work (Tian et al., 2013), the authors found a positive impact on prediction accuracy with multi-factor analysis — they have produced features from different factors such as text, related reports, and temporal information.

Ramay et al. (2019) proposed an approach by using a deep learning model, natural language techniques, and emotion analysis by using bug reports for predicting bug severity. They mentioned that this method improved the f-measure of state-of-the-art approaches by 7.90% on average. Lamkanfi et al. (2010) proposed bug severity prediction using text mining algorithms. Their analysis with Mozilla, Eclipse, and GNOME software projects suggests that with enough training data accurate bug report severity models can be built.

Tan et al. (2020) leveraged question-and-answer pairs from Stack Overflow and combined them with related bug reports to make an enhanced version of the bug reports. They predict the severity of bug reports using different machine learning models which led to improvements by approximately 23% of the average f-measure. In a recent study, Agrawal and Goyal (2021) investigated the usefulness of word embedding in predicting the severity of bug reports, and observed mixed results. Interested readers are encouraged to read the survey paper by Gomes et al. (2019) to understand the research gaps and opportunities in bug report severity prediction.

The objective of our study is significantly different than those previous studies: our focus was not to understand the severity of bug reports but to understand and predict source code methods with different bug severity. The authors in Zhou and Leung (2006) employed statistical (logistic regression) and machine learning (naive Bayes, random forest, and NNge) methods to investigate the fault-proneness prediction usefulness of OO design metrics with regard to ungraded, high, and low severity faults. Although relevant, this study is much more limited than ours from multiple aspects. They only investigate one C++ project, with just over 2000 faulty and non-faulty methods and around 500 bugs, and seven OO metrics. Whereas ours looks into over 77,000 methods including over 3000 faulty methods from 19 Java projects. Therefore, even the metric analysis part of our experiment is much larger scale, let aside the static analysis contribution.

6.3. Static analysis tools

The application of static analysis tools is common in software projects, as suggested by the findings of Lavazza et al. (2020). The authors found that approximately 50% of the issues reported by the SpotBugs tool have disappeared in the next revision. Various research has been done to evaluate the true effectiveness of various static analysis tools for bug detection tasks. Some researchers leveraged these tools as an oracle for their own provided techniques. For example, Tomassi (2018) used SpotBugs and ErrorProne to detect bugs in the BugSwarm dataset, and they found that these tools were not effective in finding the bugs since their results showed only one successful bug detection by SpotBugs. Ayewah et al. (2007) leveraged FindBugs to find bugs in Google's internal code base, and they found that integrating this tool into Google's Mondrian code review system would help developers see the potential bug existence in the code. Habib and Pradel (2018) studied SpotBugs, Infer, and Google Error Prone tools to find their capabilities in detecting Java real bugs. They concluded that these tools are mostly complementary to each other, and they miss the large majority of the bugs. Similar results — less effectiveness of bug detection tools — were also observed by Thung et al. (2012). Dura et al. (2021) introduced the JavaDL which is a Datalog-based declarative specification language for bug pattern detection in Java code, and compared it against the SpotBugs and ErrorProne tools. The authors found that JavaDL has comparable performance to these tools. Habib and Pradel (2019) proposed a method to consider bug detection as a classification problem by using the neural networks and Google Error Prone as an oracle.

To the best of our knowledge, none of the previous studies have investigated the relationship between various popular code metrics and bug severity in method-level granularity, nor the effectiveness of static analysis tools for bug severity detection.

7. Conclusion and future work

In this paper, we studied 10 source code metrics and two static analysis tools to understand and distinguish buggy code and to find their capability in estimating the bugs' severity levels. Our quantitative and qualitative studies on the Defects4J and Bugs.jar datasets — containing 3358 buggy methods from 19 Java open-source projects — showed that code metrics are good indicators of buggyness, but neither code metrics nor static analysis tools are significant estimators of bug severity. We found that there is no relationship between the code complexity and bug severity, and static analysis tools miss many bugs due to the lack of specialized patterns. Manual inspection of severe bugs reveals that some bug types (e.g., integration, and security bugs) are often more severe than others (e.g., edge/boundary, and specification bugs).

Potential future directions of this research are studying the power of dynamic analysis and testing in estimating bug severity. In addition, one can use the findings of this study regarding static analysis tools' limitations and try to enrich their rule sets to better identify severe bugs. It is also promising to combine static analysis and code metrics for better severity prediction. Lastly, the advances in large language models can be leveraged to estimate the severity of a bug based on the pre-trained models on very large datasets of code bases and further fine-tune them on smaller datasets containing severity keywords in the fixes.

CRediT authorship contribution statement

Ehsan Mashhadi: Conceptualization, Data curation, Formal analysis, Investigation, Methodology, Software, Validation, Visualization, Writing – original draft, Writing – review & editing. **Shaiful Chowdhury:** Formal analysis, Methodology, Software, Validation, Visualization, Writing – original draft, Writing – review & editing. **Somayeh Modaberi:** Data curation, Validation. **Hadi Hemmati:** Conceptualization, Funding acquisition, Investigation, Methodology, Resources, Supervision, Validation, Writing – review & editing. **Gias Uddin:** Funding acquisition, Project administration, Resources.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

Our experimental artifacts, including the source code and data, are available in our publicly shared GitHub repository (Mashhadi, 2023).

Acknowledgments

This work was partially supported by the NSERC Discovery Grant (RGPIN/04552-2020) and the NSERC and Alberta Innovates Alliance Grant (ALLRP/568643-2021).

References

Agrawal, R., Goyal, R., 2021. Developing bug severity prediction models using word2vec. Int. J. Cogn. Comput. Eng. 2, 104–115.

Alenezi, M., Akour, M., Al Sghaier, H., 2019. The impact of co-evolution of code production and test suites through software releases in open source software systems. Int. J. Innov. Technol. Explor. Eng. (IJITEE) 9 (1), 2737–2739.

AlOmar, E.A., Mkaouer, M.W., Ouni, A., Kessentini, M., 2019. On the impact of refactoring on the relationship between quality attributes and design metrics. In: 2019 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement. ESEM, IEEE, pp. 1–11.

Aniche, M.F., Oliva, G.A., Gerosa, M.A., 2013. What do the asserts in a unit test tell us about code quality? a study on open source and industrial projects. In: 2013 17th European Conference on Software Maintenance and Reengineering. IEEE, pp. 111–120.

Antinyan, V., Staron, M., Meding, W., Österström, P., Wikstrom, E., Wranker, J., Henriksson, A., Hansson, J., 2014. Identifying risky areas of software code in agile/lean software development: An industrial experience report. In: 2014 Software Evolution Week-IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering. (CSMR-WCRE), IEEE, pp. 154–163.

Antinyan, V., Staron, M., Sandberg, A., 2017. Evaluating code complexity triggers, use of complexity measures and the influence of code complexity on maintenance time. Empir. Softw. Eng. 22 (6), 3057–3087.

ATLASSIAN, 2023. Jira software. https://www.atlassian.com/software/jira.

Ayewah, N., Pugh, W., Morgenthaler, J.D., Penix, J., Zhou, Y., 2007. Using findbugs on production software. In: Companion to the 22nd ACM SIGPLAN Conference on Object-Oriented Programming Systems and Applications Companion. pp. 805–806.

Bacchelli, A., Bird, C., 2013. Expectations, outcomes, and challenges of modern code review. In: 2013 35th International Conference on Software Engineering. ICSE, IEEE, pp. 712–721.

Bacchelli, A., D'Ambros, M., Lanza, M., 2010. Are popular classes more defect prone? In: International Conference on Fundamental Approaches to Software Engineering. Springer, pp. 59–73.

Bavota, G., De Carluccio, B., De Lucia, A., Di Penta, M., Oliveto, R., Strollo, O., 2012. When does a refactoring induce bugs? an empirical study. In: 2012 IEEE 12th International Working Conference on Source Code Analysis and Manipulation. IEEE, pp. 104–113.

Bennett, K.H., Rajlich, V.T., 2000. Software maintenance and evolution: A roadmap. In: Proceedings of the Conference on the Future of Software Engineering. pp. 73–87.

Bhat, T., Nagappan, N., 2006. Evaluating the efficacy of test-driven development: industrial case studies. In: Proceedings of the 2006 ACM/IEEE International Symposium on Empirical Software Engineering. pp. 356–363.

Börstler, J., Paech, B., 2016. The role of method chains and comments in software readability and comprehension—An experiment. IEEE Trans. Softw. Eng. 42 (9), 886–898.

Buse, R.P., Weimer, W.R., 2009. Learning a metric for code readability. IEEE Trans. Softw. Eng. 36 (4), 546–558.

Celerity, 2022. The true cost of a software bug: Part one. Online; https://www.celerity.com/insights/the-true-cost-of-a-software-bug. (Last Accessed 1 September 2022).

Chowdhury, S., Holmes, R., Zaidman, A., Kazman, R., 2022a. Revisiting the debate: Are code metrics useful for measuring maintenance effort? Empir. Softw. Eng. 27 (6), 158.

Chowdhury, S., Uddin, G., Hemmati, H., Holmes, R., 2024. Method-level bug prediction: Problems and promises. ACM Trans. Softw. Eng. Methodol. http://dx.doi.org/10.1145/3640331, in press.

Chowdhury, S., Uddin, G., Holmes, R., 2022b. An empirical study on maintainable method size in java. In: 19^{Th} International Conference on Mining Software Repositories.

Curtis, B., Sheppard, S.B., Milliman, P., Borst, M., Love, T., 1979. Measuring the psychological complexity of software maintenance tasks with the halstead and McCabe metrics. IEEE Trans. Softw. Eng. (2), 96–104.

Dura, A., Reichenbach, C., Söderberg, E., 2021. JavaDL: automatically incrementalizing java bug pattern detection. Proc. ACM Program. Lang. 5 (OOPSLA), 1–31.

El Emam, K., Benlarbi, S., Goel, N., Rai, S.N., 2001. The confounding effect of class size on the validity of object-oriented metrics. IEEE Trans. Softw. Eng. 27 (7), 630–650.

ErrorProne, 2023. Errorprone. Online; URL https://errorprone.info/. (Accessed 20 April 2022).

Feng, Z., Guo, D., Tang, D., Duan, N., Feng, X., Gong, M., Shou, L., Qin, B., Liu, T., Jiang, D., et al., 2020. Codebert: A pre-trained model for programming and natural languages. arXiv preprint arXiv:2002.08155.

Ferenc, R., Bán, D., Grósz, T., Gyimóthy, T., 2020a. Deep learning in static, metric-based bug prediction. Array 6, 100021.

Ferenc, R., Gyimesi, P., Gyimesi, G., Tóth, Z., Gyimóthy, T., 2020b. An automatically created novel bug dataset and its validation in bug prediction. J. Syst. Softw. 169, 110691.

FindBugs, 2015. Findbugs. Online; URL http://findbugs.sourceforge.net/. (Accessed 20 April 2022).

Giger, E., D'Ambros, M., Pinzger, M., Gall, H.C., 2012. Method-level bug prediction. In: Proceedings of the 2012 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement. IEEE, pp. 171–180.

Giger, E., Pinzger, M., Gall, H.C., 2011. Comparing fine-grained source code changes and code churn for bug prediction. In: Proceedings of the 8th Working Conference on Mining Software Repositories. pp. 83–92.

Gil, Y., Lalouche, G., 2017. On the correlation between size and metric validity. Empir. Softw. Eng. 22 (5), 2585–2611.

Gomes, L.A.F., da Silva Torres, R., Côrtes, M.L., 2019. Bug report severity level prediction in open source software: A survey and research opportunities. Inf. Softw. Technol. 115, 58–78.

Grund, F., Chowdhury, S., Bradley, N.C., Hall, B., Holmes, R., 2021. CodeShovel: Constructing method-level source code histories. In: 2021 IEEE/ACM 43rd International Conference on Software Engineering. ICSE, IEEE, pp. 1510–1522.

Habib, A., Pradel, M., 2018. How many of all bugs do we find? a study of static bug detectors. In: 2018 33rd IEEE/ACM International Conference on Automated Software Engineering. ASE, IEEE, pp. 317–328.

Habib, A., Pradel, M., 2019. Neural bug finding: A study of opportunities and challenges. arXiv preprint arXiv:1906.00307.

Hall, T., Beecham, S., Bowes, D., Gray, D., Counsell, S., 2011. A systematic literature review on fault prediction performance in software engineering. IEEE Trans. Softw. Eng. 38 (6), 1276–1304.

Halstead, M.H., 1977. Elements of Software Science (Operating and Programming Systems Series). Elsevier Science Inc..

Hata, H., Mizuno, O., Kikuno, T., 2012. Bug prediction based on fine-grained module histories. In: 2012 34th International Conference on Software Engineering. ICSE, IEEE, pp. 200–210.

Hemmati, H., 2015. How effective are code coverage criteria? In: 2015 IEEE International Conference on Software Quality, Reliability and Security. IEEE, pp. 151–156.

Herzig, K., Zeller, A., 2013. The impact of tangled code changes. In: 2013 10th Working Conference on Mining Software Repositories. pp. 121–130.

Hess, M.R., Kromrey, J.D., 2004. Robust confidence intervals for effect sizes: A comparative study of cohen'sd and cliff's delta under non-normality and heterogeneous variances. In: Annual Meeting of the American Educational Research Association, vol. 1, Citeseer.

Hindle, A., Godfrey, M.W., Holt, R.C., 2008. Reading beside the lines: Indentation as a proxy for complexity metric. In: 2008 16th IEEE International Conference on Program Comprehension. IEEE, pp. 133–142.

Hosseini, S., Turhan, B., Gunarathna, D., 2017. A systematic literature review and meta-analysis on cross project defect prediction. IEEE Trans. Softw. Eng. 45 (2), 111–147.

Infer, 2023. Infer. Online; URL https://fbinfer.com/. (Accessed 20 April 2022).

Johnson, J., Lubo, S., Yedla, N., Aponte, J., Sharif, B., 2019. An empirical study assessing source code readability in comprehension. In: 2019 IEEE International Conference on Software Maintenance and Evolution. pp. 513–523.

Jureczko, M., Spinellis, D., 2010. Using object-oriented design metrics to predict software defects. Models Methods Syst. Dependability. Oficyna Wydawnicza Politechniki Wrocławskiej 69–81.

Just, R., Jalali, D., Ernst, M.D., 2014. 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. pp. 437–440.

Kafura, D., Reddy, G.R., 1987. The use of software complexity metrics in software maintenance. IEEE Trans. Softw. Eng. (3), 335–343.

Kanwal, J., Maqbool, O., 2012. Bug prioritization to facilitate bug report triage. J. Comput. Sci. Tech. 27 (2), 397–412.

Kasto, N., Whalley, J., 2013. Measuring the difficulty of code comprehension tasks using software metrics. In: Proceedings of the Fifteenth Australasian Computing Education Conference, vol. 136, pp. 59–65.

Khatiwada, S., Tushev, M., Mahmoud, A., 2018. Just enough semantics: An information theoretic approach for IR-based software bug localization. Inf. Softw. Technol. 93, 45–57.

Khomh, F., Penta, M.D., Guéhéneuc, Y.-G., Antoniol, G., 2012. An exploratory study of the impact of antipatterns on class change-and fault-proneness. Empir. Softw. Eng. 17 (3), 243–275.

Kondo, M., German, D.M., Mizuno, O., Choi, E.-H., 2020. The impact of context metrics on just-in-time defect prediction. Empir. Softw. Eng. 25 (1), 890–939.

Kononenko, O., Baysal, O., Godfrey, M.W., 2016. Code review quality: How developers see it. In: Proceedings of the 38th International Conference on Software Engineering. pp. 1028–1038.

Koru, A., Liu, H., 2005. Building effective defect-prediction models in practice. IEEE Softw. 22 (6), 23–29.

Lamkanfi, A., Demeyer, S., Giger, E., Goethals, B., 2010. Predicting the severity of a reported bug. In: 2010 7th IEEE Working Conference on Mining Software Repositories. (MSR 2010), IEEE, pp. 1–10.

Landman, D., Serebrenik, A., Vinju, J., 2014. Empirical analysis of the relationship between CC and SLOC in a large corpus of java methods. In: 2014 IEEE International Conference on Software Maintenance and Evolution. IEEE, pp. 221–230.

Lavazza, L., Tosi, D., Morasca, S., 2020. An empirical study on the persistence of spotbugs issues in open-source software evolution. In: International Conference on the Quality of Information and Communications Technology. pp. 144–151.

Le Goues, C., Nguyen, T., Forrest, S., Weimer, W., 2011. Genprog: A generic method for automatic software repair. Ieee Trans. Softw. Eng. 38 (1), 54–72.

Li, N., Shepperd, M., Guo, Y., 2020. A systematic review of unsupervised learning techniques for software defect prediction. Inf. Softw. Technol. 122, 106287.

Long, F., Rinard, M., 2016. Automatic patch generation by learning correct code. In: Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. pp. 298–312.

Mäntylä, M.V., Lassenius, C., 2008. What types of defects are really discovered in code reviews? IEEE Trans. Softw. Eng. 35 (3), 430–448.

Martin, R.C., 2007. Professionalism and test-driven development. Ieee Softw. 24 (3), 32–36.

Martinez, M., Durieux, T., Sommerard, R., Xuan, J., Monperrus, M., 2017. Automatic repair of real bugs in java: A large-scale experiment on the defects4j dataset. Empir. Softw. Eng. 22 (4), 1936–1964.

Mashhadi, E., 2023. Bug severity empirical study. https://github.com/EhsanMashhadi/BugSeverityEmpiricalStudy.

Mashhadi, E., Ahmadvand, H., Hemmati, H., 2023. Method-level bug severity prediction using source code metrics and LLMs. In: 2023 IEEE 34th International Symposium on Software Reliability Engineering. ISSRE, pp. 635–646. http://dx.doi.org/10.1109/ISSRE59848.2023.00055.

Mashhadi, E., Hemmati, H., 2021. Applying codebert for automated program repair of java simple bugs. In: 2021 IEEE/ACM 18th International Conference on Mining Software Repositories. MSR, IEEE, pp. 505–509.

Matter, D., Kuhn, A., Nierstrasz, O., 2009. Assigning bug reports using a vocabulary-based expertise model of developers. In: 2009 6th IEEE International Working Conference on Mining Software Repositories. pp. 131–140.

McCabe, T.J., 1976. A complexity measure. IEEE Trans. Softw. Eng. (4), 308–320.

McClure, C.L., 1978. A model for program complexity analysis. In: Proceedings of the 3rd International Conference on Software Engineering. pp. 149–157.

Microsoft, 2023. Visual Studio3. Online; URL https://docs.microsoft.com/en-us/visualstudio/code-quality/code-metrics-maintainability-index-range-and-meaning?view=vs-2022&viewFallbackFrom=vs-2022%3A. (Accessed 20 April 2022).

Miles, M.B., Huberman, A.M., 1994. Qualitative data analysis: An expanded sourcebook. Sage.

Mo, R., Cai, Y., Kazman, R., Xiao, L., Feng, Q., 2016. Decoupling level: A new metric for architectural maintenance complexity. In: 2016 IEEE/ACM 38th International Conference on Software Engineering. ICSE, IEEE, pp. 499–510.

Mo, R., Wei, S., Feng, Q., Li, Z., 2022. An exploratory study of bug prediction at the method level. Inf. Softw. Technol. 144, 106794.

Nawrocki, J., Wojciechowski, A., 2001. Experimental evaluation of pair programming. Eur. Softw. Control Metrics (Escom) 99–101.

Neysiani, B.S., Babamir, S.M., Aritsugi, M., 2020. Efficient feature extraction model for validation performance improvement of duplicate bug report detection in software bug triage systems. Inf. Softw. Technol. 126, 106344.

Okutan, A., Yıldız, O.T., 2014. Software defect prediction using Bayesian networks. Empir. Softw. Eng. : Int. J. 19 (1), 154–181.

Oman, P., Hagemeister, J., 1992. Metrics for assessing a software system's maintainability. In: Proceedings Conference on Software Maintenance 1992. IEEE Computer Society, pp. 337–338.

OpenAI, 2023. GPT-4 technical report. arXiv:2303.08774.

Pai, G.J., Dugan, J.B., 2007. Empirical analysis of software fault content and fault proneness using Bayesian methods. IEEE Trans. Softw. Eng. 33 (10), 675–686.

Palomba, F., Bavota, G., Penta, M.D., Fasano, F., Oliveto, R., Lucia, A.D., 2018. On the diffuseness and the impact on maintainability of code smells: A large scale empirical investigation. Empir. Softw. Eng. 23 (3), 1188–1221.

Pantiuchina, J., Lanza, M., Bavota, G., 2018. Improving code: The (mis) perception of quality metrics. In: 2018 IEEE International Conference on Software Maintenance and Evolution. ICSME, IEEE, pp. 80–91.

Pascarella, L., Palomba, F., Bacchelli, A., 2020. On the performance of method-level bug prediction: A negative result. J. Syst. Softw. 161, 110493.

Pearson, S., Campos, J., Just, R., Fraser, G., Abreu, R., Ernst, M.D., Pang, D., Keller, B., 2017. Evaluating and improving fault localization. In: 2017 IEEE/ACM 39th International Conference on Software Engineering. ICSE, IEEE, pp. 609–620.

Pecorelli, F., Palomba, F., Di Nucci, D., De Lucia, A., 2019. Comparing heuristic and machine learning approaches for metric-based code smell detection. In: 2019 IEEE/ACM 27th International Conference on Program Comprehension. ICPC, IEEE, pp. 93–104.

Polo, M., Piattini, M., Ruiz, F., 2001. Using code metrics to predict maintenance of legacy programs: A case study. In: Proceedings IEEE International Conference on Software Maintenance. ICSM 2001. IEEE, pp. 202–208.

Posnett, D., Hindle, A., Devanbu, P., 2011. A simpler model of software readability. In: Proceedings of the 8th Working Conference on Mining Software Repositories. pp. 73–82.

Rafique, Y., Mišić, V.B., 2012. The effects of test-driven development on external quality and productivity: A meta-analysis. IEEE Trans. Softw. Eng. 39 (6), 835–856.

Rahman, A., Williams, L., 2019. Source code properties of defective infrastructure as code scripts. Inf. Softw. Technol. 112, 148–163.

Ralph, P., Tempero, E., 2018. Construct validity in software engineering research and software metrics. In: Proceedings of the 22nd International Conference on Evaluation and Assessment in Software Engineering 2018. pp. 13–23.

Ramay, W.Y., Umer, Q., Yin, X.C., Zhu, C., Illahi, I., 2019. Deep neural network-based severity prediction of bug reports. IEEE Access 7, 46846–46857.

Saha, R.K., Khurshid, S., Perry, D.E., 2014. An empirical study of long lived bugs. In: 2014 Software Evolution Week-IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering. (CSMR-WCRE), IEEE, pp. 144–153.

Saha, R.K., Lyu, Y., Lam, W., Yoshida, H., Prasad, M.R., 2018. Bugs. jar: A large-scale, diverse dataset of real-world java bugs. In: Proceedings of the 15th International Conference on Mining Software Repositories. pp. 10–13.

Saha, R.K., Lyu, Y., Yoshida, H., Prasad, M.R., 2017. Elixir: Effective object-oriented program repair. In: 2017 32nd IEEE/ACM International Conference on Automated Software Engineering. ASE, IEEE, pp. 648–659.

Sahoo, S.K., Criswell, J., Adve, V., 2010. An empirical study of reported bugs in server software with implications for automated bug diagnosis. In: Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering, vol. 1, pp. 485–494.

Scikit-Learn, 2007-2023a. Macroaverage. Online; URL https://scikit-learn.org/stable/modules/generated/sklearn.metrics.f1_score.html. (Accessed 20 April 2022).

Scikit-Learn, 2007-2023b. Robustscaler. Online; URL https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.RobustScaler.html. (Accessed 20 April 2022).

Servant, F., Jones, J.A., 2017. Fuzzy fine-grained code-history analysis. In: Proceedings of the International Conference on Software Engineering. ICSE, pp. 746–757.

Shamshiri, S., Just, R., Rojas, J.M., Fraser, G., McMinn, P., Arcuri, A., 2015. Do automatically generated unit tests find real faults? an empirical study of effectiveness and challenges (t). In: 2015 30th IEEE/ACM International Conference on Automated Software Engineering. ASE, IEEE, pp. 201–211.

Shaphiro, S., Wilk, M., 1965. An analysis of variance test for normality. Biometrika 52 (3), 591–611.

Shepperd, M., 1988. A critique of cyclomatic complexity as a software metric. Softw. Eng. J. 3 (2), 30–36.

Shihab, E., Hassan, A.E., Adams, B., Jiang, Z.M., 2012. An industrial study on the risk of software changes. In: Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering. pp. 1–11.

Shin, Y., Meneely, A., Williams, L., Osborne, J.A., 2010. Evaluating complexity, code churn, and developer activity metrics as indicators of software vulnerabilities. IEEE Trans. Softw. Eng. 37 (6), 772–787.

Shivaji, S., Whitehead, E.J., Akella, R., Kim, S., 2009. Reducing features to improve bug prediction. In: 2009 IEEE/ACM International Conference on Automated Software Engineering. IEEE, pp. 600–604.

Sjøberg, D.I.K., Yamashita, A., Anda, B.C.D., Mockus, A., Dybå, T., 2013. Quantifying the effect of code smells on maintenance effort. IEEE Trans. Softw. Eng. 39 (8), 1144–1156.

Spadini, D., Palomba, F., Zaidman, A., Bruntink, M., Bacchelli, A., 2018. On the relation of test smells to software code quality. In: 2018 IEEE International Conference on Software Maintenance and Evolution. ICSME, IEEE, pp. 1–12.

SpotBugs, 2023. Spotbugs. Online; URL https://spotbugs.github.io/. (Accessed 20 April 2022).

Steidl, D., Hummel, B., Juergens, E., 2014. Incremental origin analysis of source code files. In: Proceedings Working Conference on Mining Software Repositories. MSR, pp. 42–51.

Subramanyam, R., Krishnan, M.S., 2003. Empirical analysis of ck metrics for object-oriented design complexity: Implications for software defects. IEEE Trans. Softw. Eng. 29 (4), 297–310.

Sun, W., Marakas, G., Aguirre-Urreta, M., 2015. The effectiveness of pair programming: Software professionals' perceptions. IEEE Softw. 33 (4), 72–79.

Sun, X., Zhou, W., Li, B., Ni, Z., Lu, J., 2019. Bug localization for version issues with defect patterns. IEEE Access 7, 18811–18820.

Tan, Y., Xu, S., Wang, Z., Zhang, T., Xu, Z., Luo, X., 2020. Bug severity prediction using question-and-answer pairs from stack overflow. J. Syst. Softw. 165, 110567.

Thung, F., Lucia, Lo, D., Jiang, L., Rahman, F., Devanbu, P.T., 2012. To what extent could we detect field defects? an empirical study of false negatives in static bug finding tools. In: Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering. pp. 50–59.

Tian, Y., Ali, N., Lo, D., Hassan, A.E., 2016. On the unreliability of bug severity data. Empir. Softw. Eng. 21, 2298–2323.

Tian, Y., Lo, D., Sun, C., 2012. Information retrieval based nearest neighbor classification for fine-grained bug severity prediction. In: 2012 19th Working Conference on Reverse Engineering. IEEE, pp. 215–224.

Tian, Y., Lo, D., Sun, C., 2013. Drone: Predicting priority of reported bugs by multi-factor analysis. In: 2013 IEEE International Conference on Software Maintenance. pp. 200–209.

Tomassi, D.A., 2018. Bugs in the wild: examining the effectiveness of static analyzers at finding real-world bugs. In: Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. pp. 980–982.

Tosun, A., Bener, A., Turhan, B., Menzies, T., 2010. Practical considerations in deploying statistical methods for defect prediction: A case study within the turkish telecommunications industry. Inf. Softw. Technol. 52 (11), 1242–1257.

Tufano, M., Palomba, F., Bavota, G., Oliveto, R., Di Penta, M., De Lucia, A., Poshyvanyk, D., 2015. When and why your code starts to smell bad. In: 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering, vol. 1, IEEE, pp. 403–414.

Uddin, J., Ghazali, R., Deris, M.M., Naseem, R., Shah, H., 2017. A survey on bug prioritization. Artif. Intell. Rev. 47 (2), 145–180.

Vucevic, D., Yaddow, W., 2012. Testing the Data Warehouse Practicum: Assuring Data Content, Data Structures and Quality. Trafford Publishing.

Wahono, R.S., 2015. A systematic literature review of software defect prediction. J. Softw. Eng. 1 (1), 1–16.

Wang, D., Lin, M., Zhang, H., Hu, H., 2010. Detect related bugs from source code using bug information. In: 2010 IEEE 34th Annual Computer Software and Applications Conference. IEEE, pp. 228–237.

Wang, Q., Xia, X., Lo, D., Li, S., 2019. Why is my code change abandoned? Inf. Softw. Technol. 110, 108–120.

Watson, A.H., Wallace, D.R., McCabe, T.J., 1996. Structured Testing: A Testing Methodology Using the Cyclomatic Complexity Metric, vol. 500, (235), US Department of Commerce, Technology Administration, National Institute of ....

Wattanakriengkrai, S., Thongtanunam, P., Tantithamthavorn, C., Hata, H., Matsumoto, K., 2020. Predicting defective lines using a model-agnostic technique. arXiv preprint arXiv:2009.03612.

Williams, L., Kessler, R.R., Cunningham, W., Jeffries, R., 2000. Strengthening the case for pair programming. IEEE Softw. 17 (4), 19–25.

Wong, W.E., Gao, R., Li, Y., Abreu, R., Wotawa, F., 2016. A survey on software fault localization. IEEE Trans. Softw. Eng. 42 (8), 707–740.

Zaw, K.K., Hnin, H.W., Kyaw, K.Y., Funabiki, N., 2020. Software quality metrics calculations for java programming learning assistant system. In: 2020 IEEE Conference on Computer Applications. ICCA, IEEE, pp. 1–6.

Zhou, Y., Leung, H., 2006. Empirical analysis of object-oriented design metrics for predicting high and low severity faults. IEEE Trans. Softw. Eng. 32 (10), 771–789.

Zhou, Y., Xu, B., Leung, H., 2010. On the ability of complexity metrics to predict fault-prone classes in object-oriented systems. J. Syst. Softw. 83 (4), 660–674.

Zimmermann, T., Premraj, R., Zeller, A., 2007. Predicting defects for eclipse. In: Third International Workshop on Predictor Models in Software Engineering (PROMISE'07: ICSE Workshops 2007). IEEE, 9–9.

Ehsan Mashhadi is an experienced software developer with a Master's degree in Computer Science from the University of Calgary. His research interests include Automated Software Engineering and AI, specifically in areas like bug detection, program repair, and empirical software engineering.

Shaiful Chowdhury is an Assistant Professor in the Department of Computer Science, at the University of Manitoba, Winnipeg, Canada. He has worked as a postdoctoral fellow at the University of Calgary, and the University of British Columbia, Canada. He received his Ph.D from the University of Alberta, Canada which won the outstanding PhD thesis award. He received his MSc and BSc degrees in Computer Science from the University of Saskatchewan, Canada, and the University of Chittagong, Bangladesh, respectively. Shaiful's research interests include software maintenance, software energy modeling and efficiency, and mining software repositories. Among many other awards, Shaiful won an ACM SIGSOFT DISTINGUISHED paper award

(at ICSE 2021), and the Early Achievement Award in PhD (Computing Science) at the University of Alberta. He also received the mining challenge paper award at MSR 2015.

Dr. Hadi Hemmati is an associate professor at the electrical engineering and computer science department, at York University. Previously he was an associate professor at the electrical and software engineering department at the University of Calgary, AB, Canada. In the past, he was also an assistant professor at the University of Manitoba, and a postdoctoral fellow at the University of Waterloo, and Queen's university. He received his Ph.D. from the University of Oslo, Norway. His main research interests are automated software engineering (with a focus on software testing, debugging, and repair), and trustworthy AI (with a focus on robustness and explainability). His research has a strong focus on pragmatic software/ML solutions for large-scale systems and empirically investigating them in practice. He has been a PI on multiple industry research projects in different domains such as IT, aviation, insurance, urban development, fintech, and beyond.

Dr. Gias Uddin has been a tenure-track assistant professor at the department of Electrical and Software Engineering of University of Calgary since July 2020. Gias completed his PhD from McGill University in 2018 and his master's from Queen's University in 2008. Prior to joining the University of Calgary, Gias was a senior data scientist at the Bank of Canada, the central Bank of Government of Canada. Between Sept 2021 - June 2022, he also managed the first-ever data science team at the Office of the Superintendent of Financial Institutions (OSFI). Gias has been deeply involved with the Canadian software industry and public sector since 2008 in various increasingly senior roles related to software engineering and data science. At IBM, he was a core member of the Natural Language Processing team of IBM Watson Analytics. At the University of Calgary, his research lies at the intersection of Software Engineering (SE) and AI/ML to understand how these two domains can benefit from each other (AI = Artificial Intelligence). More specifically, he Studies the challenges practitioners face in software engineering (SE) and Machine Learning (ML) domains. To address the challenges, he designs human-centric interactive tools by combining human expertise with machine intelligence. His research works were published at the topmost journals and conferences in SE research (IEEE TSE, ICSE, ASE, etc.). His research was covered by international news (e.g., BBC news) and is currently funded by several external grants (IBM, NSERC, Alberta Innovates, etc.). Website: https://giasuddin.ca/.