Find Bugs in Static Bug Finders

Junjie Wang[1,2,3,*], Yuchao Huang[1,3,*], Song Wang[4], Qing Wang[1,2,3,#]
[1]Laboratory for Internet Software Technologies, [2]State Key Laboratory of Computer Sciences,
Institute of Software Chinese Academy of Sciences, Beijing, China;
[3]University of Chinese Academy of Sciences, Beijing, China; [*]Co-first author; [#]Corresponding author
[4]Electrical Engineering and Computer Scienc, York University, Canada;
{junjie,yuchao2019,wq}@iscas.ac.cn,wangsong@eecs.yorku.ca

ABSTRACT

Static bug finders (also known as static code analyzers, e.g., FindBugs, SonarQube) have been widely-adopted by developers to find bugs in real-world software projects. They leverage predefined heuristic static analysis rules to scan source code or binary code of a software project, and report violations to these rules as warnings to be verified. However, the advantages of static bug finders are overshadowed by such issues as uncovered obvious bugs, false positives, etc. To improve these tools, many techniques have been proposed to filter out false positives reported or design new static analysis rules. Nevertheless, the under-performance of bug finders can also be caused by the incorrectness of current rules contained in the static bug finders, which is not explored yet. In this work, we propose a differential testing approach to detect bugs in the rules of four widely-used static bug finders, i.e., SonarQube, PMD, SpotBugs, and ErrorProne, and conduct a qualitative study about the bugs found. The experiment on 2,728 open source projects reveals 46 bugs in the static bug finders, among which 30 are fixed or confirmed and the left are awaiting confirmation. We also summarize 13 bug patterns in the static analysis rules based on their context and root causes, which can serve as the checklist for designing and implementing other rules and/or in other tools. This study indicates that the commonly-used static bug finders are not as reliable as they might have been envisaged. It not only demonstrates the effectiveness of our approach, but also highlights the need to continue improving the reliability of the static bug finders.

ACM Reference Format:
Junjie Wang[1,2,3,*], Yuchao Huang[1,3,*], Song Wang[4], Qing Wang[1,2,3,#]. 2021. Find Bugs in Static Bug Finders. In *Proceedings of The 30th International Conference on Program Comprehension (ICPC 2022).* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3377811.3380380

1 INTRODUCTION

The increasing complexity of modern software systems has complicated both the development of new software features and the maintenance of source code. Techniques that can detect and reduce bugs are very beneficial to help developers improve software quality. To achieve this goal, static bug finders (also known as static code analyzers) that analyze code characteristics without program execution such as Sonarqube [10] and Findbugs [22], have been widely used to find bugs in software [8, 14, 39, 50, 53, 56]. These tools mainly leverage predefined heuristic analysis rules to scan source code or binary code of a software project, and report violations to these rules as warnings to be verified. Most of static bug finders can infer a wide variety of bugs, security vulnerabilities, and bad programming practices [16, 56, 64].

Previous studies have shown that static bug finders can help in detecting software defects faster and cheaper than human inspection or software testing [6, 26, 70]. They have been widespread adopted by professional software developers, and regularly integrated in contemporary open source projects and commercial software organizations [28, 47, 69]. For example, Errorprone and Infer has been automatically applied to code changes to support manual code review at Google and Facebook, respectively [3]. However, the advantages of static bug finders are overshadowed by such issues as uncovered obvious bugs [18, 62], false positives [26, 32], etc. To improve these tools and enhance their usability, different lines of studies have been proposed.

Several researchers proposed to utilize prioritization strategies to make it easier for developers to spot the more actionable warnings [20, 21, 29, 34, 64]. Other researchers employed feedback-based rule design for mitigating false positives [45, 54], e.g., Errorprone used unactionable warnings labeled by developers [54], and FeeFin refined the rules by the development practice of open source projects [45]. Another line of studies focused on designing project-specific rules that mined from specific projects to improve the detection accuracy [9, 11, 19, 25, 33, 36], e.g., PR-Miner [33] and NAR-Miner [9] mined programming rules and detected violations with frequent itemset mining algorithms. In addition, there are studies proposed to combine diverse static bug finders to improve the detection coverage [4, 46]. All above mentioned practices focused on improving current static bug finders by adjusting the warning results based on existing static analysis rules or designing new rules. However, the under-performance of bug finders can also be caused by the incorrectness of current rules contained in these static bug finders, which is not explored yet.

In this work, to assess the correctness of static analysis rules of static bug finders, we propose a differential testing approach to detect bugs in the rules of four widely-used static bug finders, i.e., SonarQube, PMD, SpotBugs, and ErrorProne, and conduct a qualitative study about the bugs found in these tools.

The assumption of our approach is that static analysis rules from different bug finders that target at the same type of bugs should have consistent bug detection results (i.e., warnings) on the same inputted software projects. Specifically, our work starts from

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ICPC 2022, May 21–22, 2022, Pittsburgh, PA, USA
© 2021 Association for Computing Machinery.
ACM ISBN 978-1-4503-7121-6/20/05…$15.00
https://doi.org/10.1145/3377811.3380380

retrieving *paired rules* that target at the same type of bugs among different static bug finders. We then run the static bug finders on a large set of experimental projects and check whether there are inconsistencies between the reported warnings of these paired rules. To retrieve paired rules across static bug finders for differential testing, we design a heuristic-based rule mapping method. which combines the similarity in rules' description and the overlap in warning information reported by the tools.

To evaluate our approach, for the four examined bug finders, we treat SonarQube and PMD as a pair (which scan source code of software projects to detect bugs), while SpotBugs and ErrorProne as a another pair (which scan binary code of software projects to detect bugs). Our heuristic-based rule mapping method retrieves 74 pairs of rules for SonarQube and PMD, while 30 pairs of rules are retrieved for SpotBugs and ErrorProne. We use 2,728 open source projects from an existing publicly available dataset as the experimental subjects for detecting the inconsistencies in the warnings. Results show that 7,633 inconsistencies between the paired rules from different static bug finders have been revealed.

We then apply descriptive coding, a qualitative analysis method, on the detected inconsistencies to identify the buggy rules in the static bug finders and categorize these bugs to derive bug patterns. 46 bugs in the static analysis rules across the four static bug finders are found, among which 8 are bugs in the implementations of rules that cause them to generate false positives, 38 are bugs that cause them to miss detect true bugs (i.e., false negatives). We further summarize 13 bug patterns in the static analysis rules based on the bugs' context and root causes. For example, bug pattern *fail in multiple calling operations* denotes the static analysis rules would fail to warn the suspicious code when involving the multiple calling operations as *json.exception().printStackTrace()* (work for single calling operation as *exp.printStackTrace()*). These bug patterns can serve as the checklists for developers when designing and implementing other static analysis rules and/or in other static bug finders. We also localize these bugs and summarize three types of typical faults in the implementation of these static bug finders.

To evaluate the usefulness of this study, we report these found bugs to the development team, and 30 is fixed or confirmed and the left are awaiting confirmation. This study indicates that the commonly-used static bug finders are not as reliable as they might have been envisaged. It highlights the need to continue improving the reliability of the static bug finders, and suggests the feasibility of utilizing differential testing on these static bug finders.

This paper makes the following contributions:

- We conduct the first differential testing on four widely-used static bug finders, which is the first work on testing the correctness of static analysis rules in static bug finders to the best of our knowledge.
- Our study finds 46 bugs about the implementation or design of static analysis rules, among which 30 are fixed/confirmed[1].
- We propose a heuristic-based static analysis rule mapping method to retrieve paired rules that target at the same types of bugs across different static bug finders.
- We summarize 13 bug patterns in the static analysis rules based on their context and root causes, which can serve as the checklist for designing and implementing other rules.

[1]Details are listed in https://github.com/wuchiuwong/Diff-Testing-01.

2 METHODOLOGY

2.1 Examined Static Bug Finders

In this study, we explore the correctness of static analysis rules for four popular open source static bug finders as listed below.

1) *SonarQube* is one of the most widely adopted static bug finders that leverages pre-defined static analysis rules to help find bugs in the context of continuous integration. It supports more than 20 programming languages and has been adopted by more than 85,000 organizations or software projects. SonarQube provides developers with its own analysis rules and also incorporates rules from other popular bug finders, e.g., CheckStyle, PMD, and FindBugs. In this study, we only experiment with SonarQube's own rules, i.e., 545 Java related rules in its rule repository[2].

2) *PMD* is a source code analyzer maintained by open community. It finds common programming flaws like unused variables, empty catch blocks, and unnecessary object creation, etc. It supports Java, JavaScript, PLSQL, Apache Velocity, XML, etc. We include all its 304 Java related rules for experiment[3].

3) *SpotBugs* is the spiritual successor of the pioneering FindBugs tool [23], carrying on from the point where it left off with support of its community. It is a bug finder which uses static analysis to look for bugs in Java code. It was originally developed by the University of Maryland, and has been downloaded more than a million times. We use all the 449 Java related rules for experiment[4].

4) *ErrorProne* is a static bug finder for Java that catches common programming mistakes at compile-time. It is developed by Google and is integrated into their static analysis ecosystem [53]. We experiment with all its 333 Java related rules[5].

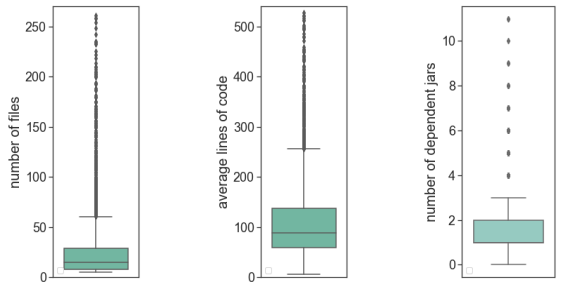
The first two static bug finders work on the source code of software projects, while the last two tools require the compiled binary code of software projects. Since this difference might lead to the variations in the marked line of the suspicious code, we group the first two tools (i.e., SonarQube and PMD) as a pair and the last two tools (SpotBugs and ErrorProne) as the second pair to conduct the differential testing. Note that, this study focuses on the static bug detection of Java projects with Java related static analysis rules, which is one of the most commonly-used programming languages.

2.2 Experimental Projects

In order to fully explore the static analysis rules, we need a large set of projects whose source code is available and compilable (for collecting binary code). We also expect these projects having flaws to cover as many static analysis rules as possible, so that the static bug finders can be triggered and the inconsistencies in their warning results can be potentially revealed.

To satisfy all above requirements, we choose to use the 50K-C projects repository [40], which contains 50,000 Java projects

[2]https://rules.sonarsource.com/java
[3]https://pmd.github.io/latest/pmd_rules_java.html
[4]https://spotbugs.readthedocs.io/en/latest/bugDescriptions.html
[5]https://errorprone.info/bugpatterns



(a) # files **(b) # LOC per file** **(c) # dependent jars**

Figure 1: Details of experimental projects

crawled from GitHub. Each project is attached with the dependencies required to compile it, and the scripts with which the projects can be compiled. All of them are active open source projects, so that there should be dozens of flaws which can trigger the static analysis rules.

We randomly download 3,000 projects from the repository, then employ their provided building framework *SourcererJBF* to compile the source code[40]. 272 projects could not be successfully compiled because of such errors as incompatible character set. We use the remaining 2,728 projects with 1565 KLOC (Kilometer Lines Of Code) in total, for the following experiment. Figure 1 presents the details of these projects with number of files in each project, average lines of source code per file in each project, and number of dependent jars for compiling the project. These projects are from various domains, e.g., player, compiler, database, map, game, etc[6]. Note that, the maximum value of the first two series of data are respectively 1877 and 3370, and we cut off the figures to facilitate visualization. There are an average of 32 files in an experimental project, and each file has an average of 115 lines of source code.

2.3 Differential Testing of Static Analysis Rules

The primary idea of this study is to detect bugs in static bug finders through differential testing, i.e., by providing the same input to different implementations of the same functionality and observing the inconsistencies between the implementations. To achieve this goal, we treat each rule implemented in the static bug finder as a functionality of the bug finder, and treat the paired rules from different bug finders which target at detecting the same types of bugs as different implementations of the same functionality.

As demonstrated in Figure 2, this study first retrieves the paired rules between two static bug finders (Section 2.3.1), i.e., two rules target at the same types of bugs, then detects the inconsistencies between the bug finders when respectively running the paired rules with the same inputted software projects (Section 2.3.2). Based on the inconsistencies, it identifies the buggy rules in the static bug finders and categorizes them (Section 2.3.3), meanwhile it also localizes the found bugs in the static bug finders (Section 2.3.4).

2.3.1 Retrieving Paired Rules. As showed in Section 2.1, each examined static bug finder has hundreds of rules, and there would be tens of thousands candidate pairs, e.g., 165,680 (545×304) pairs for matching each rule of Snoarqube to each rule of PMD. Manually mapping such large number of rules from different bug finders could be time- and effort-consuming. Besides, rules from different tools are described differently, e.g., rule *throwable.printStackTrace should not be called* in Sonarqube is described with 80 words with two code examples, while its paired rule *AvoidPrintStackTrace* in PMD has only 7 terms with one code example, which further increases the difficulty of rule mapping.

To cope with the above circumstance, we propose a heuristic-based rule mapping method which combines the similarity in rules' description and the overlap in warning information reported by the tools. In detail, we first choose the potential rule pairs with the similarity of rules based on their textual descriptions and accompanied code examples. For each candidate rule pair, we then check the detailed warning information reported by the rules and filter out the less possible rule pairs, in which two rules with larger degree of overlap have higher possibility to be a pair. This is conducted with four *mapping-rules* which will be described below.

1) Choosing Potential Rule Pairs Based on Description Similarity

Generally speaking, each static analysis rule in these static bug finders is described with three fields: *title*, *detailed description*, and *code examples*, in which title and description demonstrate what types of bugs the rule targets at and how the analysis works, while code examples present the positive and negative examples to show when the rule is triggered. We concatenate the title and detailed description fields, and treat it as the textual content of the rule. To model the similarity between the descriptions of two rules from different aspects, we use three types of similarity metrics, i.e., *term similarity* to measure the text similarity of the rule's textual content, *semantic similarity* to measure the semantic similarity of the rule's textual content, and *code similarity* to measure the similarity of code examples. Details are as follows.

Term similarity ($Term_{sim}$). The term similarity is measured with the Term Frequency and Inverse Document Frequency (TF-IDF). It is one of the most popular features for representing textual documents in information retrieval. The main idea of TF-IDF is that if a term appears many times in one rule and a few times in the other rules, the term has a good capability to differentiate the rules, and thus the term has high TF-IDF value. Specifically, given a term t and a rule r, $TF(t, r)$ is the number of times that term t occurs in rule r, while $IDF(t)$ is obtained by dividing the total number of rules by the number of rules containing term t. TF-IDF is computed as: $TF - IDF(t, r) = TF(t, r) \times IDF(t)$.

With the above formula, the textual content of a rule r can be represented as a TF-IDF vector, i.e., $r = (w_1, w_2, ..., w_n)$, where w_i denotes the TF-IDF value of the i^{th} term in rule r. Then term similarity is calculated as the cosine similarity between the vectors of two rules.

Semantic similarity ($Semt_{sim}$). The above mentioned TF-IDF similarity focuses on the similarity of rules considering the term matching. We also employ word embedding feature, which concerns more on the relationship of terms considering the context they appear, to better model the semantic similarity of two rules. Word embedding is a popular feature learning technique in natural

[6]Details of the domain distribution are in our website.

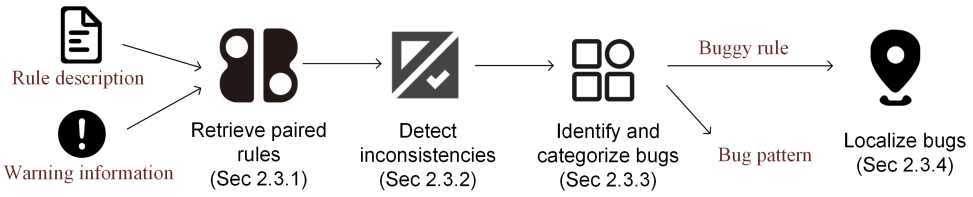


Figure 2: Overview of differential testing of static analysis rules

language processing where individual words are no longer treated as unique symbols, but represented as d-dimensional vector of real numbers that capture their contextual semantic meanings [7, 43].

We use the publicly available software[7] to obtain the word embedding of a rule. With the trained word embedding model, each word can be transformed into a d-dimensional vector where d is set to 100 as suggested in previous studies [63, 65, 67]. Meanwhile a rule can be transformed into a matrix in which each row represents a term in the rule. We then transform the rule matrix into a vector by averaging all the word vectors the rule contains as previous work did [63, 65, 67]. Specifically, given a rule matrix that has n rows in total, we denote the i^{th} row of the matrix as r_i and the transformed rule vector v_d is generated as follows:

$$v_d = \frac{\sum_i r_i}{n} \tag{1}$$

With the above formula, each rule can be represented as a word embedding vector, and semantic similarity is calculated as the cosine similarity between the vectors of two rules.

For training the word embedding model, we collect 100,000 java related questions and answers from StackOverflow. We then combine these text with the rule description of the four examined tools, and utilize them for model training. The reason why we use these data is that previous studies have revealed that to train an effective word embedding model, a domain-specific dataset with large size is preferred. The size of our training dataset is 101 Megabytes.

Code similarity ($Code_{sim}$). The code examples of each rule contain the name of class and method targeted by the rule, which are indispensable sources of information for determining whether two rules share the same functionality. We first extract the class name and method name of each rule, separate them into terms with camel-back notation following existing study [1]. We then compare the two set of terms of $rule_x$ and $rule_y$ to derive the code similarity with the following equation.

$$Code_{sim} = \frac{terms\ in\ rule_x \cap terms\ in\ rule_y}{terms\ in\ rule_x \cup terms\ in\ rule_y} \tag{2}$$

The final description similarity between two rules is calculated as follows.

$$Description_{sim} = (Term_{sim} + Semt_{sim}) \times \frac{Code_{sim} + 1}{2} \tag{3}$$

We simply add the term similarity and semantic similarity together because existing researches suggested both of them are important [63, 67]. The code similarity, which is smoothed considering it might be 0, can be seen as a filter by which if two rules share large portion of class and method names, they are more likely to be paired, otherwise they are less likely to be paired even they are similar in textual content.

[7]https://code.google.com/archive/p/word2vec/

Based on the description similarity, we design *mapping-rule a* to choose the potential static analysis rule pairs.

- *(Mapping-rule a) Retrieving pairs of mutual top-N similarity.* If $rule_b$ is within $rule_a$'s *top-N* most similar rules and vice versa, we treat $rule_a$ and $rule_b$ as a candidate pair for further investigation, where $rule_a$ and $rule_b$ are respectively from a pair of static bug finders.

2) Filtering Out Less Possible Rule Pairs Based on Warning Information

We then run each static bug finder on the experimental projects collected in Section 2.2. For each reported warning, we record the triggered static analysis rule, the warned line(s) of code, the method and file where the warning occurs.

Based on the reported warning information, we then design *mapping-rules (b,c,d)* to jointly filter out the less possible rule pairs from the potential rule pairs generated with *mappping-rule (a)*. These three mapping-rules are designed considering the following two assumptions: 1) the paired rules should have large degree of overlap in their warnings; 2) considering one of the paired rules might have bugs, we allow their reported warnings can be partially overlapped.

- *(Mapping-rule b) Pruning with one-to-one pair.* If the percentage of overlaps of the warned lines reported by $rule_a$ and $rule_b$ exceeds 80% of warned lines from each of the rules, we assume the candidate pair has extremely high possibility being the paired rules. Note that, to simplify our approach, we focus on one-to-one mapping among two sets of rules from different bug finders, thus we remove other candidate pairs related with $rule_a$ and $rule_b$. Another note is that, we choose the pair with the largest overlap ratio when multiple rule pairs satisfy the threshold.
- *(Mapping-rule c) Pruning with difference in warning trigger times.* If the difference of warning trigger times for $rule_a$ and $rule_b$ exceeds 20 times, we assume these two rules can hardly related with the same functionality and remove the candidate pair.
- *(Mapping-rule d) Pruning with difference in warning file.* If the overlap of warned files by $rule_a$ and $rule_b$ is lower than 2%, we assume these two rules can hardly related with the same functionality and remove the candidate pair.

We find a large portion of rules from SpotBugs and ErrorProne do not have code examples and exert a lower similarity; thus for this tool pair, we set N as 5 in *mapping-rule a*, while set it as 3 for another tool pair (i.e., SonarQube and PMD). Other parameters in the mapping rules is determined empirically, which aims at automatically retrieving a reasonable number of candidate pairs for

manual inspection. We will mention in the threats to validity that there do exist one-to-many mappings, however for facilitating the proposed differential testing, we only focus on one-to-one mapping between rules from different tools.

3) Retrieving Final Rule Pairs Manually

For all the remaining candidate paired rules after the above four mapping-rules, we conduct a manual check to finally determine the paired rules. In detail, the first three authors independently check the candidate pairs, and determine whether the two rules have the same functionality based on the rules' description and code examples. The results of their independent mapping have a Cohens kappa of 0.87, which is a substantial level of agreement [41]. They then discuss the disagreement online until the final consensus is reached.

2.3.2 **Detecting Inconsistencies**. Based on the retrieved paired rules, we check the inconsistencies in the warnings generated by the paired rules when running the static bug finders, i.e., whether the warnings reported by paired rules mark the same place in the source code. Since different rules would highlight the warnings at different granularities, e.g., a specific line of code or the whole method, we employ different criteria for the inconsistency detection for different paired rules.

Criterion 1, when both of the paired rules warn *a specific line of code*, we check whether the warned file and warned line coincide with each other between the paired rules, and treat the case in which the warned lines are different as the inconsistency.

Criterion 2, when one or two of the paired rules warn(s) the *whole method*, e.g., rule *ReturnEmptyArrayRatherThanNull* of PMD marks the entire method while its paired rule (*Empty arrays and collections should be returned instead of null*) in SonarQube only marks the *return* line (as the example shown below), we check whether the warned methods coincide with each other between the paired rules, and treat the case in which the warned methods are different as the inconsistency.

```
private float [] getReactionFlyShip(Ship ship) { //warn by PMD
    float box[] = new float [8];
    box[0] = ship.getAngle() / (2f * 3.14159f);
    ...
    return null;    //warn by SonarQube
}
```

2.3.3 **Identifying and Categorizing Bugs**. We apply a qualitative analysis method called descriptive coding [55][51] on the detected inconsistencies to identify the buggy rules in the static bug finders and categorize these bugs to derive bug patterns. Figure 3 demonstrates an example of how we analyze the inconsistencies and summarize bug patterns.

The detected inconsistencies are delivered to the first three authors respectively, and each of them manually checks them and identifies the bugs. Specifically, they first determine which of the paired rules is buggy, and whether the inconsistency involves a false negative bug (i.e., suspicious code is not warned) or a false positive bug (i.e., normal code is wrongly warned). We also find some inconsistencies are caused by the the imperfectness in the rule definition (see Section 3.2.2), therefore the authors also check the description of these static analysis rules to determine whether the bug is caused by the inaccurate *implementation*, or mainly because of the imperfectness in rule *definition*, to provide a more comprehensive view of these buggy rules. The authors then examine the context and root causes of the bug, and summarize bug patterns of these buggy rules, as shown in Table 4. The disagreement of the above analysis is discussed until common consensus is reached.

The categorization of the bug patterns might subject to the author's personal judgement. We mitigate this limitation by recruiting two independent workers who are not authors of the paper. These two workers, with more than five years background in software development, independently evaluate if the derived bug pattern is meaningful and correct. We provide them with the bug pattern name, the involved buggy rule, random-chosen one consistent warning and two inconsistent warnings for each buggy rule. The two workers independently determined if each of the buggy rule belongs to the bug pattern, and whether the bug pattern is meaningful. Both workers agree with the categorization and most of the bug patterns. The only disagreement is the name of *P6. Fail in unnecessary brackets* whose original name is *Fail in AST change by unnecessary brackets*. This high degree of agreement implies the quality of the derived bug patterns.

2.3.4 **Localizing Bugs**. For the detected buggy static analysis rules, we further examine the source code of the static bug finders to localize the bugs. For PMD, some rules are implemented with XPath technique [2], while the others are implemented in JAVA, both of which are navigated in the XML configuration files. We use the class names to localize the rules' implementation, and examine the faults in the source code of static bug finders. For other three static bug finders, we search the related JAVA file with corresponding rule name or rule id to localize the bugs.

3 EXPERIMENTAL RESULTS

3.1 Results of Rule Mapping

We first present the results of how many rules of each bug finder are triggered and the warning trigger times after running the bug finders on our experimental projects, with results in Table 1. We can see that 74% to 89% rules are triggered for SonarQube, PMD and SpotBugs, while for ErrorProne, 45% of its rules are triggered. We analyze the un-triggered rules of ErrorProne and find that most of them are related to the framework or package developed by Google itself, e.g., AutoValue, Dagger, etc., which are seldom used by general software projects, thus were not triggered. On average, 73% rules are triggered and each rule is triggered a median of 163 times, which suggests the generalizability of our experiments.

Following the rule mapping method described in Section 2.3.1, we retrieve the paired rules and list the results in Table 2, in which we present the number of remaining rule pairs after applying *mapping-rule (a-d)*. 74 rule pairs from SonarQube and PMD, and 30 rule pairs from SpotBugs and ErrorProne are finally determined as having identical functionality[8]. The hitting rate of our mapping method is 51.5% (74/145 in Table 2) for paired tools SonarQube and PMD, 45.4% (30/66 in Table 2) for paired tools SpotBugs and ErrorProne. This indicates we can find one mapped rule pairs by examining an estimate of two candidate pairs, which is quite cost-effective considering the tremendous number of total candidate pairs.

[8]These paired rules are listed in https://github.com/wuchiuwong/Diff-Testing-01.

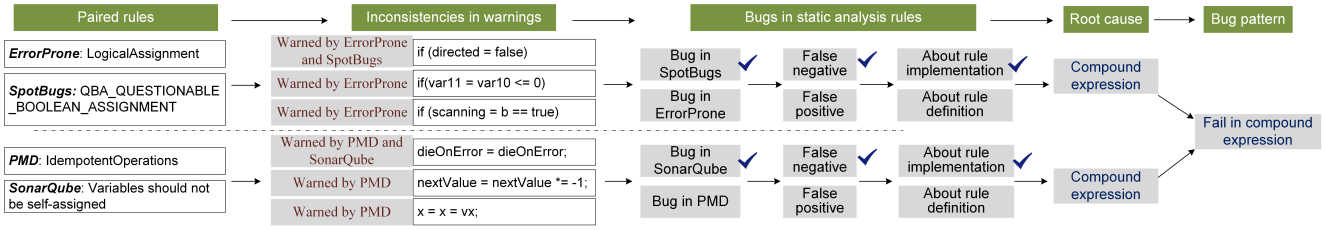


Figure 3: An example of how we analyze the inconsistencies and summarize bug patterns

Table 1: Statistics of warning information on experimental projects

Tool	% triggered rules	Warning trigger times per triggered rule				
		min	1-quarter	median	3-quarter	max
SonarQube	80%	1	38	378	2,320	1,036,425
PMD	89%	2	183	1,091	6,967	812,325
SpotBugs	74%	1	7	45	216	9,049
ErrorProne	45%	1	7	36	137	77,896
Overall	73%	1	22	163	1,363	1,036,425

It is almost impossible to measure the recall of our retrieved rule pairs, considering the large number of candidate pairs. We construct a small-scale ground truth set to roughly evaluate the recall. In detail, we choose the rule pairs which 1) the overlap of warned files by two rules is 100%; and 2) the warning trigger times are larger than 10 to reduce the noise. Based on the filtered rule pairs, we conduct the manual check as previous section and obtain the paired rules. Since these rule pairs are determined solely based on the warning information, they can be treated as orthogonal with the rule pairs retrieved by the proposed mapping method which are firstly determined with description similarity. Based on the constructed ground truth rule pairs, 56% (42/74) rule pairs for SonarQube and PMD are recalled, and 73% (22/30) rule pairs for SpotBugs and ErrorProne are recalled. This further indicates the effectiveness of our proposed mapping method which can help find sufficient number of rule pairs with little human effort.

3.2 Detected Bugs in Static Bug Finders

Overall, 7,633 inconsistencies (i.e., sum of *occurrence times* in Table 4, 5 and 6) are revealed from the two pairs of static bug finders on our experimental projects.

Following the qualitative analysis method presented in Section 2.3.3, we examine these inconsistencies and identify 46 bugs in the rules of the four static bug finders, i.e., 10 in Sonarqube, 25 in PMD, 6 in SpotBugs, and 5 in ErrorProne, as shown in Table 3. Among the bugs, 38 are false negative bugs (i.e., the suspicious code is not warned by the tool), while 8 are false positive bugs (i.e., the clean code is warned by the tool). In addition, for false negative bugs, 29 bugs are caused by rule implementation, while 9 bugs are caused by rule definition; for false positive bugs, all of them are because of the rule implementation. We discuss the three types of bugs in the following three subsections respectively.

3.2.1 **False Negative Bugs (about Rule Implementation).** For the 29 false negative bugs about rule implementation, seven bug patterns are summarized following the procedure described in Section 2.3.3. Table 4 demonstrates a summarized view of these bug patterns, with the illustrative example, the involved static analysis rules and the occurrence times of the bug (i.e., the number of inconsistencies for triggering the bug). There are 3 remaining bugs, each of which belongs to a specific type, thus we put them in *Others* category, and leave them for future exploration. We then present the detailed analysis of an example pattern to facilitate understanding.

P2) Fail in compound expression. The involved rules in this pattern cannot work with compound expressions (i.e., two or more operands in an expression). For example, rule *QBA_QUESTIONABLE_BOOLEAN_ASSIGNMENT* from Spotbugs checks *a literal boolean value (true or false) assigned to a boolean variable inside an if or while expression. Most probably this was supposed to be a boolean comparison using ==, not an assignment using =*. Code example (a) shows the consistent case where both SpotBugs and ErrorProne can detect the suspicious code with the paired rules, while code example (b) shows the inconsistent case where only ErrorProne marks the suspicious code, i.e., the corresponding rule in SpotBugs has a bug. The compound expression in code example (b) triggers the bug of this rule in SpotBugs.

(Code example a.) Warning reported by both SpotBugs and ErrorProne.

```
// birker-fsm/fsm-master/src/fsm/EdgeFsm.java
public void setDirected(boolean directed) {
    if (directed = false) throw new IllegalArgumentException(""Fsm are
        always directed!"");  //warn by SpotBugs and ErrorProne
}
```

(Code example b.) Warning reported by ErrorProne only.

```
//lunchza-VisualHDD/VisualHDD-master/Visual HDD/src/visual/gui/
    ProgramWindow.java
public void setScanStatus(boolean b) {
    if (scanning = b == true){   //mark only by ErrorProne
        scanning = true ;}
    else if(scanning = b == false && canceled == true){   //warn only by
        Errorprone
        scanning = false ;
        ...
}
```

3.2.2 **False Negative Bugs (about Rule Definition).** For the 9 false negative bugs about rule definition, three bug patterns (as shown in Table 5) are summarized following the procedure described in Section 2.3.3. The bugs in the above subsection are caused

Table 2: Results of rule mapping

Tool	Total candidate pairs	Pairs after applying rule *a*	Pairs after applying rule *b*	Pairs after applying rule *c*	Pairs after applying rule *d*	Final pairs
Sonarqube & PMD	165,680	424	367	252	145	74
ErrorProne & SpotBugs	149,517	432	387	264	66	30

Table 3: Detected bugs on four static bug finders

Tool	False negative (about rule implementation)	False negative (about rule definition)	False Positive	Overall
SonarQube	6	3	1	10
PMD	15	4	6	25
SpotBugs	4	1	1	6
ErrorProne	4	1	0	5
Overall	29	9	8	46

Table 4: False negative bugs (about rule implementation) in the examined static bug finders

Bug pattern (# involved rules)	Description	Example	Involved rules (number of inconsistencies) / C indicates fixed/confirmed bug
P1. **Fail in special data type (5)**	Rules fail to warn the suspicious code involving special data types	Rule *Object should not be created only to getClass* fails with Array, while works with ArrayList;	*PMD*: SingularField (327) *SonarQube*: String function use should be optimized for single characters (8) / **C** *SonarQube*: Objects should not be created only to getClass (5) / **C** *PMD*: RedundantFieldInitializer (3) / **C** *SpotBugs*: ICAST_BAD_SHIFT_AMOUNT (3)
P2. **Fail in compound expression (5)**	Rules fail to warn the suspicious code involving compound expression	Rule *QBA_QUESTIONABLE _BOOLEAN _ASSIGNMENT* fails when compound expression is involved, e.g., *if (scanning = b == false);*	*ErrorProne*: ToStringReturnsNull (28) / **C** *SpotBugs*: QBA_QUESTIONABLE_BOOLEAN _ ASSIGNMENT (26) / **C** *SpotBugs*: SA_LOCAL_SELF_ASSIGNMENT (26) / **C** *SonarQube*: Variables should not be self-assigned (5) / **C** *SonarQube*: Fields should not be initialized to default values (3) / **C**
P3. **Fail in implicit operation (4)**	Rules fail to warn the suspicious involving the implicit operation of the defined suspicious operation	Rule *IntLongMath (Expression of type int may overflow before assigning to a long)* fails when involving comparison operation (i.e., implicit assignment operation), e.g., *if (score >= level × level × 1000);*	*ErrorProne*: IntLongMath (135) / **C** *SonarQube*: Static fields should not be updated in constructors (72) / **C** *SpotBugs*: DMI_INVOKING_TOSTRING_ON_ ARRAY (53) / **C** *ErrorProne*: ArrayEquals (11)
P4. **Fail in multiple calling operations (4)**	Rules fail to warn the suspicious code involving multiple calling operations	Rule *AvoidPrintStackTrace* fails when involving multiple calling operations, e.g., *json.exception().printStackTrace();*	*PMD*: UseCollectionIsEmpty (399) / **C** *PMD*: AvoidPrintStackTrace (20) / **C** *PMD*: ClassCastExceptionWithToArray (5) *PMD*: DontCallThreadRun (5) / **C**
P5. **Fail in separated expressions (3)**	Rules fail to warn the suspicious code involved in separated expressions	Rule *UseProperClassLoader* fails when involving separated expressions, e.g., *Foo foo = new Foo();* *ClassLoader classLoader = foo.getClassLoader();*	*PMD*: UseProperClassLoader (68) / **C** *ErrorProne*: ToStringReturnsNull (28) / **C** *PMD*: InstantiationToGetClass (19)
P6. **Fail in unnecessary brackets (3)**	Rules fail to warn the suspicious code involving unnecessary brackets which changes AST of the code	Rule *SimplifyConditional* fails with *if(rhs != null && (rhs instanceof CodeLocation))*, yet works when deleting the unnecessary brackets on 2rd condition;	*PMD*: ReturnEmptyArrayRatherThanNull (770) *PMD*: SimplifyConditional (109) *SonarQube*: Switch statements should not contain non-case labels (39) / **C**
P7. **Fail in variables (2)**	Rules fail to warn the suspicious code involving variables, while works with constant	Rule *FinalFieldCouldBeStatic* fails when assigning to an expression, e.g., *private final double HPI = Math.PI ∗ 0.5*, yet works when assigning to a constant, e.g., *protected final int margin = 3*;	*PMD*: FinalFieldCouldBeStatic (209) *PMD*: AvoidDecimalLiteralsInBigDecimalConstructor (20) / **C**
Others (3)	**Others**	N/A	*PMD*: AvoidThrowingNullPointerException (772) *PMD*: SimplifyBooleanExpressions (623) *PMD*: StringToString (215) / **C**

by the inaccurate *implementation* of rules, while the bugs in this subsection are mainly because of the imperfectness in rule *definition*. For example, we find the definition of some rules miss specific data types. We separate them to remind the tool developers about the flaw in the design of these static analysis rules. We then present the detailed analysis of an example pattern to facilitate understanding.

Table 5: False negative bugs (about rule definition) in the examined static bug finders.

Bug pattern (# involved rules)	Description	Example	Involved rules (number of inconsistencies) / C indicates fixed/confirmed bug
P8. **Miss comparable method (4)**	Rules miss comparable method of the defined suspicious method	Rule *UseLocaleWithCaseConversions* fails with *String.format()*, while works with *String.toLowerCase()/toUpperCase();*	*PMD*: UseLocaleWithCaseConversions (464) *ErrorProne*: BoxedPrimitiveEquality (14) / C *SonarQube*: Java.lang.Error should not be extended (7) / C *SonarQube*: Execution of the Garbage Collector should be triggered only by the JVM (7) / C
P9. **Miss comparable data type or operation (3)**	Rules miss the comparable data type or operation of the defined suspicious ones	Rule *ICAST_INTEGER_MULTIPLY_CAST _TO_LONG* fails with *shift* operation, while works with *multiply* operation;	*SonarQube*: Redundant modifiers should not be used (1271) / C *PMD*: AvoidArrayLoops (362) / C *SpotBugs*: ICAST_INTEGER_MULTIPLY_CAST_TO_LONG (39)
P10. **Miss subclass or superclass (2)**	Rules miss the subclass or superclass of the defined suspicious class	Rule *AvoidCatchingThrowable* fails in the subclass of *Throwable*, e.g., catch (Error e);	*PMD*: ReturnEmptyArrayRatherThanNull (677) *PMD*: AvoidCatchingThrowable (41)

P8) Miss comparable method. The rules in this pattern miss certain comparable method. Take the rule *UseLocaleWithCaseConversions* from PMD as an example. We present its rule description, as well as the description of its paired rule *Locale should be used in String operations* of SonarQube as follows.

- *PMD (UseLocaleWithCaseConversions)*: When doing String:: *toLowerCase()/toUpperCase()* conversions, use an explicit locale argument to specify the case transformation rules.
- *SonarQube (Locale should be used in String operations)*: Failure to specify a locale when calling the methods *toLowerCase(), toUpperCase() or format()* on String objects means the system default encoding will be used, possibly creating problems with international characters or number representations.

We can see that in the rule definition of PMD, only two string related methods are mentioned, while the third method *format()* is included in the rule definition of SonarQube. The results from the inconsistent detection by running these two static bug finders on experimental projects confirm that *String.format()* cannot be warned by PMD, which suggest the design of the rule *UseLocaleWithCaseConversions* from PMD is incomplete and could be buggy.

3.2.3 **False Positive Bugs**. For the 8 false positive bugs, three bug patterns (as shown in Table 6) are summarized following the procedure described in Section 2.3.3. We then present the detailed analysis of an example pattern to facilitate understanding.

P11) Poor handling of method with same name. Bugs related with the rules in this pattern occur because they warn the correct method which shares the same method name (yet different method signatures) with the defined suspicious method. Take the rule *Thread.notify()* from PMD as an example. This rule states *its usually safer to call notifyAll() rather than notify() because the later one awakens an arbitrary thread monitoring the object when more than one thread is monitoring.* The following code examples first present the consistent case where both PMD and SonarQube can detect the suspicious code in example (a), followed by the inconsistent case where PMD wrongly highlights the normal code in example (b), i.e., bug in PMD. We can see that although the method name is *notify*, it is not *Object.notify()* as defined in the rule, since these two methods have different method signatures. PMD does not filter this special yet misleading case, which suggests a potential bug in the rule of PMD.

(Code example a.) Warning reported by both SonarQube and PMD.

```
// belaban–JGroups/JGroups–master/tests/other/org/jgroups/ tests /TestToaOrder.
      java
public void memberFinished(Address addr) {
    synchronized (members) {
        members.remove(addr);
        if (members.isEmpty()) {
            members.notify(); // warn by PMD and SonarQube
        }
    }
}
```

(Code example b.) Warning reported by PMD only.

```
// mypsycho–SwingAppFramework/SwingAppFramework–master/src/main/java/
      org/mypsycho/beans/Injection.java
private void injectSimple(Object bean, InjectionContext context){
    ...
    else if (toSet && !child.definition.isEmpty()) {
        getInjector().notify(getCanonicalName(), "'" + child.definition + "
            ' has been converted as null", null); // warn only by PMD
    }
}
```

3.3 Typical Faults in Static Bug Finders Causing Buggy Rules

Besides detecting the bugs in static analysis rules of the bug finders, we further examine the source code of these static bug finders and localize the faults for these buggy rules listed in Tables 4, 5 and 6. Based on our analysis of the bug localization results, we summarize the following three types of typical faults.

1) Inflexible design of rule implementation

We notice that the implementation of 60% (15/25) detected buggy rules in PMD involves the XPath technology [2], which searches for specific expression on the Abstract Syntax Tree (AST) of the analyzed program. This implementation is less flexible and very sensitive to noisy data.

Take the rule *SimplifyConditional* from PMD as an example, we have presented the analysis in Section 3.2.1.P6 where we show this rule fails when AST changes by adding the unnecessary brackets. When implementing this rule, as shown below, it would search the conditional statement (i.e., *ConditionalAndExpression*) and check whether it is the EqualityExpression (i.e., *rhs != null*) and InstanceOf-Expression (i.e., *rhs instanceof CodeLocation*) respectively. When the two expressions are no longer in parallel after adding the brackets, the rule would fail to work.

```
<![CDATA[ //Expression
```

Table 6: False positive bugs in the examined static bug finders

Bug pattern (# involved rules)	Description	Example	Involved rules (number of inconsistencies) / C indicates fixed/confirmed bug
P11. **Poor handling of method with same name (3)**	Rules wrongly warn the clean method which has the same name yet different method signatures with the defined suspicious method	*UseNotifyAllInsteadOfNotify* should warn *Object.notify()*, but wrongly warns *notify(para)* method in other class, e.g., getEventThread().notify(input);	*PMD*: SuspiciousEqualsMethodName (24) *PMD*: UseNotifyAllInsteadOfNotify (12) / C *SonarQube*: Object.finalize() should remain protected when overriding (6) / C
P12. **Setting over-sized scope (3)**	Rules wrongly warn the clean code which is beyond the scope of the defined suspicious case	*AvoidThrowingNullPointerException* wrongly warns the expression with *NullPointerException* yet without *Throwing*, e.g., Exception e = new NullPointerException("msg");	*PMD*: AvoidThrowingNullPointerException (18) / C *SpotBugs*: SF_SWITCH_NO_DEFAULT (11) *PMD*: AvoidCallingFinalize (6) / C
P13. **Neglecting corner case (2)**	Rules wrongly warn the clean code which is the corner case of the defined suspicious case	*MissingBreakInSwitch* wrongly warns the *switch* expression without *break* in the last *case* statement;	*PMD*: MissingBreakInSwitch (841) / C *PMD*: AvoidReassigningLoopVariables (188) / C

```
[ConditionalOrExpression ....
or ConditionalAndExpression
    [EqualityExpression[@Image='!=']// NullLiteral and
    InstanceOfExpression[PrimaryExpression[count(PrimarySuffix[
        @ArrayDereference='true']) =0]// Name[not(contains(@Image,'.'))]/
        @Image = ancestor::
    ConditionalAndExpression/EqualityExpression/PrimaryExpression/
        PrimaryPrefix/Name/@Image] and (count(InstanceOfExpression) + 1 =
        count(*))
]]]]>
```

2) Uncovering potential influenced statements

The four static bug finders employ similar strategies to implement the rules. In detail, given a specific rule, these bug finders first categorize all the statements in the code under analysis according to their functionalities, e.g, variable definition, variable assignment, conditional judgment, etc. Then they further visit the pre-defined potential problematic statements, and analyze them to determine whether there is a match, i.e., a warning is given.

Take the rule *IntLongMath* from ErrorProne (mentioned in Section 3.2.1.P3 as an example, the rule first locates the problematic statements, i.e., return, initialization, assignment (i.e., *matchAssignment()* as shown in the code below). It then determines whether the result is *Long* type, and filter out the operations which can not result in overflow. However, when locating the problematic statements, it does not consider the *compare statement* which can also trigger the bug.

```
@Override
public Description matchAssignment(AssignmentTree tree, VisitorState state)
    {
  return check(ASTHelpers.getType(tree), tree.getExpression());
}
Description check(Type targetType, ExpressionTree init) {
  if (targetType.getKind() != TypeKind.LONG) {
    return NO_MATCH;
  }
   ...
}
```

3) Missing considering special cases

Many faults are caused because of their neglecting in special cases. For the rule *Objects should not be created only to getClass* of SonarQube, it fails in the *Array* type as mentioned in Section 3.2.1.P1. The implementation code shown below demonstrates that this rule is designed for *java.lang.Object*, which does not include *Array*. More special cases should be included to ensure the robustness of these rules.

```
protected List<MethodMatcher> getMethodInvocationMatchers(){
  return Collections.singletonList(MethodMatcher.create().typeDefinition(
      TypeCriteria.subtypeOf("java.lang.Object")).name("getClass").
      withoutParameter());
}
```

3.4 Usefulness Evaluation

To further demonstrate the usefulness of this study, for each detected bug, we create a bug report by describing the issue, the example code, and the analyzed reason, then report it to the development team through an issue report. Among the 46 detected bugs, 30 have been fixed or confirmed by the developers, and the left are awaiting confirmation. The fixed/confirmed bugs are marked in Table 4 to Table 6, and all the reported issues and detailed status are listed on our website[9].

We further examine the fixed/confirmed bugs, and results show that 19 are fixed, 7 are marked as "wontfix", while 4 are awaiting fix. We summarize the reasons for the 7 "wontfix" bugs as follows: 1 of them is because of the related detection rules having been deprecated, 2 are because of the developers intentionally implementing the rule to ignore such cases, while the remaining 4 are because of the reported false positives/false negatives can be avoided by configuring the static bug finders. Since this paper only applied the default configuration of these static bug finders, when more configurations are included, the 4 "wontfix" bugs can be potentially avoided. For the 4 bugs awaiting fix, the reasons are mainly due to the smaller influence scope or the lower priority.

The fixed and confirmed bugs further demonstrate the usefulness of this study in helping the developers improving these widely-used bug finders.

4 DISCUSSIONS AND THREATS TO VALIDITY

4.1 Discussions

Checklist for static analysis rule design and implementation. Table 4, 5, and 6 summarizes the bug patterns in the implementation and design of static analysis rules. We also notice that most of these bug patterns involve rules target at different types of warnings and from different bug finders, which implies the generalizability of these bug patterns.

The bug patterns are actually the special cases which is ignored by the static analysis rules and cause them fail to warn the corresponding suspicious code or wrongly warn the clean code. We

[9] https://github.com/wuchiuwong/Diff-Testing-01

believe these bug patterns can serve as the checklist when one designs or implements new rules and/or in other static bug finders. Take bug pattern *P2. Fail in compound expression* as an example, we find some static analysis rules fail to warn the suspicious code involving compound expression. Equipped with such a bug pattern, one should pay careful attention to this special case when designing/implementing a new rule, and include related test cases to cover this special case, both of which can help improve the quality of newly implemented static analysis rules.

Customizing static bug finders to avoid execution of duplicate rules (i.e., paired rules). Many software organizations tend to configure multiple bug finders for detecting bugs with an assumption that different bug finders emphasize on detecting different types of bugs in the source code [37, 53]. For example, Github projects as Springfox[10] and Roboguice[11] employ both SpotBugs and PMD for code inspection [69]. Another example is SonarQube which incorporates static analysis rules from other bug finders, i.e., SpotBugs, PMD, cobertura, and CheckStyle (note that, we exclude these external rules in our experiment).

Our rule mapping results reveal a non-negligible portion of duplicate rules among the examined static bug finders, e.g., at least 24% (74/304) rules from PMD are the duplicates of the rules in SonarQube. However, most of the bug finders are used in default configuration [62], which results in the duplicate rules repeatedly running and brings in heavy overhead. A more feasible alternative would be customizing these bug finders to make the duplicate rules only execute once, and the mapped rule pairs retrieved in this study further provide the feasibility for the customization. The differential testing results of this study also provide the detailed guidelines to do so, e.g., choose the correct rule if one of them is buggy.

4.2 Threats to Validity

The first threat of this study is the selection of static bug finders, which may or may not be representative for a larger population. We experiment with four popular open source static bug finders, which are widely used in previous researches and industrial practice [5, 8, 64, 69], which we believe to be representative for the current state-of-the-art. Despite of this, there are other commonly-used static bug finders, e.g., Infer, CheckStyle, Coverity. The reason why do not utilize Infer or CheckStype is because they either have few static analysis rules (e.g., 25 rules in Infer) or focus more on the coding standards (i.e., CheckStyle), both of which limit us in finding plenty of mapping rules and detecting more inconsistencies. Besides, Coverity is a closed sourced bug finder with which we could not conduct the bug localization.

Another threat to validity is our methodology for mapping static analysis rules which could, in principle, miss some paired rules. To overcome the challenges that tens of thousands of candidate pairs needed to be examined, we design a heuristic-based rule mapping method. This could miss some true paired rules, yet make this mapping task can be done with reasonable human effort. We employ several empirical parameters in filtering the candidate rule pairs which might also influence the recall of paired rules. Nevertheless, based on the retrieved mapping rules, we have detected 46 bugs in the tools. Besides, we present the hitting rate and recall of rule mapping in Section 3.1 to show the reliability of the method.

[10]https://github.com/springfox/springfox
[11]https://github.com/roboguice/roboguice

5 RELATED WORK

The use of static bug finders for software defect detection is a common practice for developers during software development and has been studied by many researchers [14, 16, 26, 44, 50, 52, 71]. There were studies investigating the adoption of static bug finders within continuous integration pipelines [47, 61, 69]. Other studies focused on how these warnings are actually acted, and fixed [24, 38, 59]. Several researchers proposed to utilize prioritization strategies to make it easier for the developers to spot the more actionable warnings [20, 21, 29, 34, 64]. There were several researches employing feedback-based rule design for mitigating false positives [45, 54]. Another line of researches focused on designing project-specific rules that mined from specific projects to improve the detection accuracy [9, 11, 19, 25, 33, 36]. In addition, there are studies proposed to combine diverse static bug finders to detect defects and vulnerabilities, so as to improve the detection coverage [4, 46]. All above mentioned researches focused on improving current static bug finders by adjusting the warning results or designing new rules, while this study investigate the correctness of bug finders which can improve the detection accuracy fundamentally.

Differential testing is originally introduced by McKeeman which attempts to detect bugs by checking inconsistent behaviors across different comparable software or different software versions [42]. Randomized differential testing is a widely-used black-box differential testing technique in which the inputs are randomly generated [12, 17, 31, 35, 48, 49, 57, 58, 60, 66, 68]. Inconsistency detection has been used in other domains such as cross-platform [15], web browsers [13], document readers [30], and program variables [27]. Different from the above mentioned application scenarios, we apply differential testing to detect bugs in widely-used static bug finders which can improve the performance of bug detection.

6 CONCLUSION

Static bug finders are widely used by professional software developers, and regularly integrated in contemporary open source projects and commercial software organizations. They have been shown to be helpful in detecting software defects faster and cheaper than human inspection or software testing. To further improve the reliability of static bug finders, this paper proposes a differential testing approach to detect bugs in the static analysis rules of four widely-used static bug finders. Our study finds 46 bugs about the implementation or design of static analysis rules, among which 30 are fixed/confirmed and the left are awaiting confirmation. We also summarize 13 bug patterns in the static analysis rules based on their context and root causes, which can serve as the checklist for designing and implementing other rules.

ACKNOWLEDGMENTS

This work is supported by the National Key Research and Development Program of China under grant No.2018YFB1403400, the National Natural Science Foundation of China under grant No.62072442 and 62002348, the Youth Innovation Promotion Association Chinese Academy of Sciences.

REFERENCES

[1] 2020. https://en.wikipedia.org/wiki/camelcase.

[2] 2020. http://www.ing.iac.es/~docs/external/java/pmd/xpathruletutorial.html.

[3] Edward Aftandilian, Raluca Sauciuc, Siddharth Priya, and Sundaresan Krishnan. 2012. Building Useful Program Analysis Tools Using an Extensible Java Compiler. In *12th IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM 2012, Riva del Garda, Italy, September 23-24, 2012*. 14–23.

[4] Areej Algaith, Paulo Jorge Costa Nunes, José Fonseca, Ilir Gashi, and Marco Vieira. 2018. Finding SQL Injection and Cross Site Scripting Vulnerabilities with Diverse Static Analysis Tools. In *14th European Dependable Computing Conference, EDCC 2018, Iași, Romania, September 10-14, 2018*. IEEE Computer Society, 57–64.

[5] Nathaniel Ayewah, David Hovemeyer, J. David Morgenthaler, John Penix, and William Pugh. 2008. Using Static Analysis to Find Bugs. *IEEE Software* 25, 5 (2008), 22–29.

[6] Moritz Beller, Georgios Gousios, Annibale Panichella, Sebastian Proksch, Sven Amann, and Andy Zaidman. 2019. Developer Testing in the IDE: Patterns, Beliefs, and Behavior. *IEEE Trans. Software Eng.* 45, 3 (2019), 261–284. https://doi.org/10.1109/TSE.2017.2776152

[7] Yoshua Bengio, Réjean Ducharme, Pascal Vincent, and Christian Janvin. 2003. A Neural Probabilistic Language Model. *The Journal of Machine Learning Research* 3 (2003), 1137–1155.

[8] Al Bessey, Ken Block, Benjamin Chelf, Andy Chou, Bryan Fulton, Seth Hallem, Charles-Henri Gros, Asya Kamsky, Scott McPeak, and Dawson R. Engler. 2010. A few billion lines of code later: using static analysis to find bugs in the real world. 53, 2 (2010), 66–75.

[9] Pan Bian, Bin Liang, Wenchang Shi, Jianjun Huang, and Yan Cai. 2018. NAR-miner: discovering negative association rules from code for bug detection. In *Proceedings of the 2018 ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2018, Lake Buena Vista, FL, USA, November 04-09, 2018*. ACM, 411–422.

[10] G Campbell and Patroklos P Papapetrou. 2013. *SonarQube in action*. Manning Publications Co.

[11] Boyuan Chen and Zhen Ming (Jack) Jiang. 2017. Characterizing and detecting anti-patterns in the logging code. In *Proceedings of the 39th International Conference on Software Engineering, ICSE 2017, Buenos Aires, Argentina, May 20-28, 2017*. 71–81.

[12] Yuting Chen, Ting Su, and Zhendong Su. 2019. Deep differential testing of JVM implementations. In *Proceedings of the 41st International Conference on Software Engineering, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019*. 1257–1268.

[13] Shauvik Roy Choudhary. 2011. Detecting cross-browser issues in web applications. In *Proceedings of the 33rd International Conference on Software Engineering, ICSE 2011, Waikiki, Honolulu , HI, USA, May 21-28, 2011*. 1146–1148.

[14] Lisa Nguyen Quang Do, James Wright, and Karim Ali. 2020. Why do software developers use static analysis tools? a user-centered study of developer needs and motivations. *IEEE Transactions on Software Engineering* (2020).

[15] Mattia Fazzini and Alessandro Orso. 2017. Automated cross-platform inconsistency detection for mobile apps. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering, ASE 2017, Urbana, IL, USA, October 30 - November 03, 2017*. 308–318.

[16] Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. 2002. Extended Static Checking for Java. In *Proceedings of the 2002 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Berlin, Germany, June 17-19, 2002*. 234–245.

[17] Jianmin Guo, Yu Jiang, Yue Zhao, Quan Chen, and Jiaguang Sun. 2018. DLFuzz: differential fuzzing testing of deep learning systems. In *Proceedings of the 2018 ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2018, Lake Buena Vista, FL, USA, November 04-09, 2018*. 739–743.

[18] Andrew Habib and Michael Pradel. 2018. How many of all bugs do we find? a study of static bug detectors. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE 2018, Montpellier, France, September 3-7, 2018*, Marianne Huchard, Christian Kästner, and Gordon Fraser (Eds.). 317–328.

[19] Quinn Hanam, Fernando Santos De Mattos Brito, and Ali Mesbah. 2016. Discovering bug patterns in JavaScript. In *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2016, Seattle, WA, USA, November 13-18, 2016*. 144–156.

[20] Sarah Heckman and Laurie Williams. 2008. On Establishing a Benchmark for Evaluating Static Analysis Alert Prioritization and Classification Techniques. In *ESEM 2008*. 41–50.

[21] Sarah Heckman and Laurie Williams. 2011. A systematic literature review of actionable alert identification techniques for automated static code analysis. *Information and Software Technology* 53, 4 (2011), 363 – 387.

[22] David Hovemeyer and William Pugh. 2004. Finding bugs is easy. *Acm sigplan notices* 39, 12 (2004), 92–106.

[23] David Hovemeyer and William Pugh. 2007. Finding more null pointer bugs, but not too many. In *Proceedings of the 7th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering, PASTE'07, San Diego, California, USA, June 13-14, 2007*, Manuvir Das and Dan Grossman (Eds.). ACM, 9–14.

[24] N. Imtiaz, B. Murphy, and L. Williams. 2019. How Do Developers Act on Static Analysis Alerts? An Empirical Study of Coverity Usage. In *2019 IEEE 30th International Symposium on Software Reliability Engineering (ISSRE)*. 323–333.

[25] Guoliang Jin, Linhai Song, Xiaoming Shi, Joel Scherpelz, and Shan Lu. 2012. Understanding and detecting real-world performance bugs. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '12, Beijing, China - June 11 - 16, 2012*. 77–88.

[26] Brittany Johnson, Yoonki Song, Emerson R. Murphy-Hill, and Robert W. Bowdidge. 2013. Why don't software developers use static analysis tools to find bugs?. In *35th International Conference on Software Engineering, ICSE '13, San Francisco, CA, USA, May 18-26, 2013*. 672–681.

[27] Sayali Kate, John-Paul Ore, Xiangyu Zhang, Sebastian Elbaum, and Zhaogui Xu. 2018. Phys: Probabilistic Physical Unit Assignment and Inconsistency Detection. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2018)*. 563–573.

[28] M. Kern, F. Erata, M. Iser, C. Sinz, F. Loiret, S. Otten, and E. Sax. 2019. Integrating Static Code Analysis Toolchains. In *2019 IEEE 43rd Annual Computer Software and Applications Conference (COMPSAC)*, Vol. 1. 523–528.

[29] Sunghun Kim and Michael D. Ernst. 2007. Which Warnings Should I Fix First?. In *FSE 2007*. 45–54.

[30] Tomasz Kuchta, Thibaud Lutellier, Edmund Wong, Lin Tan, and Cristian Cadar. 2018. On the correctness of electronic documents: studying, finding, and localizing inconsistency bugs in PDF readers and files. *Empirical Software Engineering* 23, 6 (2018), 3187–3220.

[31] Vu Le, Mehrdad Afshari, and Zhendong Su. 2014. Compiler validation via equivalence modulo inputs. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014*. 216–226.

[32] Valentina Lenarduzzi, Francesco Lomio, Heikki Huttunen, and Davide Taibi. 2020. Are SonarQube Rules Inducing Bugs?. In *SANER'20*. 501–511.

[33] Zhenmin Li and Yuanyuan Zhou. 2005. PR-Miner: automatically extracting implicit programming rules and detecting violations in large software code. In *Proceedings of the 10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2005, Lisbon, Portugal, September 5-9, 2005*. 306–315.

[34] Guangtai Liang, Ling Wu, Qian Wu, Qianxiang Wang, Tao Xie, and Hong Mei. 2010. Automatic Construction of an Effective Training Set for Prioritizing Static Analysis Warnings. In *ASE 2010*. 93–102.

[35] Christopher Lidbury, Andrei Lascu, Nathan Chong, and Alastair F. Donaldson. 2015. Many-core compiler fuzzing. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015*. 65–76.

[36] V. Benjamin Livshits and Thomas Zimmermann. 2005. DynaMine: finding common error patterns by mining software revision histories. In *Proceedings of the 10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2005, Lisbon, Portugal, September 5-9, 2005*. 296–305.

[37] Bailin Lu, Wei Dong, Liangze Yin, and Li Zhang. 2018. Evaluating and Integrating Diverse Bug Finders for Effective Program Analysis. In *Software Analysis, Testing, and Evolution - 8th International Conference, SATE 2018, Shenzhen, Guangdong, China, November 23-24, 2018, Proceedings*, Lei Bu and Yingfei Xiong (Eds.), Vol. 11293. 51–67.

[38] D. Marcilio, R. Bonifácio, E. Monteiro, E. Canedo, W. Luz, and G. Pinto. 2019. Are Static Analysis Violations Really Fixed? A Closer Look at Realistic Usage of SonarQube. In *2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC)*. 209–219.

[39] Diego Marcilio, Carlo A Furia, Rodrigo Bonifacio, and Gustavo Pinto. 2020. SpongeBugs: Automatically generating fix suggestions in response to static code analysis warnings. *Journal of Systems and Software* 168 (2020), 110671.

[40] Pedro Martins, Rohan Achar, and Cristina V. Lopes. 2018. 50K-C: a dataset of compilable, and compiled, Java projects. In *Proceedings of the 15th International Conference on Mining Software Repositories, MSR 2018, Gothenburg, Sweden, May 28-29, 2018*. 1–5. https://doi.org/10.1145/3196398.3196450

[41] Mary L McHugh. 2012. Interrater reliability: the kappa statistic. *Biochemia medica: Biochemia medica* 22, 3 (2012), 276–282.

[42] William M. McKeeman. 1998. Differential Testing for Software. *Digital Technical Journal* 10, 1 (1998), 100–107.

[43] T. Mikolov, I. Sutskever, K. Chen, G. Corrado, and J. Dean. 2013. Distributed Representations of Words and Phrases and Their Compositionality. In *NIPS'13*. 3111–3119.

[44] Nachiappan Nagappan and Thomas Ball. 2005. Static analysis tools as early indicators of pre-release defect density. In *27th International Conference on Software Engineering (ICSE 2005), 15-21 May 2005, St. Louis, Missouri, USA*. 580–586.

[45] Jaechang Nam, Song Wang, Yuan Xi, and Lin Tan. 2019. A bug finder refined by a large set of open-source projects. *Inf. Softw. Technol.* 112 (2019), 164–175.

[46] Paulo Jorge Costa Nunes, Iberia Medeiros, José Fonseca, Nuno Ferreira Neves, Miguel Correia, and Marco Vieira. 2017. On Combining Diverse Static Analysis Tools for Web Security: An Empirical Study. In *13th European Dependable Computing Conference, EDCC 2017, Geneva, Switzerland, September 4-8, 2017*. IEEE Computer Society, 121–128.

[47] Sebastiano Panichella, Venera Arnaoudova, Massimiliano Di Penta, and Giuliano Antoniol. 2015. Would static analysis tools help developers with code reviews?. In *22nd IEEE International Conference on Software Analysis, Evolution, and Reengineering, SANER 2015, Montreal, QC, Canada, March 2-6, 2015*. 161–170.

[48] Jihyeok Park, Seungmin An, Dongjun Youn, Gyeongwon Kim, and Sukyoung Ryu. 2021. JEST: N+1 -version Differential Testing of Both JavaScript Engines and Specification. In *43rd IEEE/ACM International Conference on Software Engineering, ICSE 2021, Madrid, Spain, 22-30 May 2021*. IEEE, 13–24.

[49] Hung Viet Pham, Thibaud Lutellier, Weizhen Qi, and Lin Tan. 2019. CRADLE: cross-backend validation to detect and localize bugs in deep learning libraries. In *Proceedings of the 41st International Conference on Software Engineering, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019*. 1027–1038.

[50] Louis-Philippe Querel and Peter C. Rigby. 2021. Warning-Introducing Commits vs Bug-Introducing Commits: A tool, statistical models, and a preliminary user study. In *29th IEEE/ACM International Conference on Program Comprehension, ICPC 2021, Madrid, Spain, May 20-21, 2021*. IEEE, 433–443.

[51] Akond Rahman, Chris Parnin, and Laurie Williams. 2019. The seven sins: security smells in infrastructure as code scripts. In *Proceedings of the 41st International Conference on Software Engineering, ICSE 2019*. 164–175.

[52] Foyzur Rahman, Sameer Khatri, Earl T. Barr, and Premkumar T. Devanbu. 2014. Comparing static bug finders and statistical prediction. In *36th International Conference on Software Engineering, ICSE '14, Hyderabad, India - May 31 - June 07, 2014*. 424–434.

[53] Caitlin Sadowski, Edward Aftandilian, Alex Eagle, Liam Miller-Cushon, and Ciera Jaspan. 2018. Lessons from building static analysis tools at Google. *Commun. ACM* 61, 4 (2018), 58–66.

[54] Caitlin Sadowski, Jeffrey van Gogh, Ciera Jaspan, Emma Söderberg, and Collin Winter. 2015. Tricorder: Building a Program Analysis Ecosystem. In *37th IEEE/ACM International Conference on Software Engineering, ICSE 2015, Florence, Italy, May 16-24, 2015, Volume 1*. IEEE Computer Society, 598–608.

[55] Johnny Saldaña. 2009. *The coding manual for qualitative researchers*. Sage Publications Ltd.

[56] J. Smith, B. Johnson, E. Murphy-Hill, B. Chu, and H. R. Lipford. 2019. How Developers Diagnose Potential Security Vulnerabilities with a Static Analysis Tool. *IEEE Transactions on Software Engineering* 45, 9 (2019), 877–897.

[57] Thodoris Sotiropoulos, Stefanos Chaliasos, Vaggelis Atlidakis, Dimitris Mitropoulos, and Diomidis Spinellis. 2021. Data-Oriented Differential Testing of Object-Relational Mapping Systems. In *43rd IEEE/ACM International Conference on Software Engineering, ICSE 2021, Madrid, Spain, 22-30 May 2021*. IEEE, 1535–1547.

[58] Chengnian Sun, Vu Le, and Zhendong Su. 2016. Finding and analyzing compiler warning defects. In *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016*. 203–213.

[59] Ferdian Thung, Lucia, David Lo, Lingxiao Jiang, Foyzur Rahman, and Premkumar T. Devanbu. 2015. To what extent could we detect field defects? An extended empirical study of false negatives in static bug-finding tools. *Autom. Softw. Eng.* 22, 4 (2015), 561–602.

[60] Cong Tian, Chu Chen, Zhenhua Duan, and Liang Zhao. 2019. Differential Testing of Certificate Validation in SSL/TLS Implementations: An RFC-guided Approach. *ACM Trans. Softw. Eng. Methodol.* 28, 4 (2019), 24:1–24:37.

[61] Carmine Vassallo, Fabio Palomba, Alberto Bacchelli, and Harald C. Gall. 2018. Continuous code quality: are we (really) doing that?. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE 2018, Montpellier, France, September 3-7, 2018*, Marianne Huchard, Christian Kästner, and Gordon Fraser (Eds.). ACM, 790–795.

[62] Carmine Vassallo, Sebastiano Panichella, Fabio Palomba, Sebastian Proksch, Harald C. Gall, and Andy Zaidman. 2020. How developers engage with static analysis tools in different contexts. *Empirical Software Engineering* 25, 2 (2020), 1419–1457.

[63] Junjie Wang, Mingyang Li, Song Wang, Tim Menzies, and Qing Wang. 2019. Images don't lie: Duplicate crowdtesting reports detection with screenshot information. *Inf. Softw. Technol.* 110 (2019), 139–155.

[64] Junjie Wang, Song Wang, and Qing Wang. 2018. Is there a "golden" feature set for static warning identification?: an experimental evaluation. In *Proceedings of the 12th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement, ESEM 2018, Oulu, Finland, October 11-12, 2018*. 17:1–17:10.

[65] B. Xu, D. Ye, Z. Xing, X. Xia, G. Chen, and S. Li. 2016. Predicting Semantically Linkable Knowledge in Developer Online Forums via Convolutional Neural Network. In *ASE'16*. 51–62.

[66] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. 2011. Finding and understanding bugs in C compilers. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4-8, 2011*. 283–294.

[67] X. Yang, D. Lo, X. Xia, L. Bao, and J. Sun. 2016. Combining word embedding with information retrieval to recommend similar bug reports. In *ISSRE'16*. 127–137.

[68] Yibiao Yang, Yuming Zhou, Hao Sun, Zhendong Su, Zhiqiang Zuo, Lei Xu, and Baowen Xu. 2019. Hunting for bugs in code coverage tools via randomized differential testing. In *Proceedings of the 41st International Conference on Software Engineering, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019*. 488–498.

[69] Fiorella Zampetti, Simone Scalabrino, Rocco Oliveto, Gerardo Canfora, and Massimiliano Di Penta. 2017. How open source projects use static code analysis tools in continuous integration pipelines. In *Proceedings of the 14th International Conference on Mining Software Repositories, MSR 2017, Buenos Aires, Argentina, May 20-28, 2017*. 334–344.

[70] Jinglei Zhang, Rui Xie, Wei Ye, Yuhan Zhang, and Shikun Zhang. 2020. Exploiting Code Knowledge Graph for Bug Localization via Bi-Directional Attention. In *Proceedings of the 28th International Conference on Program Comprehension (ICPC '20)*. 219–229.

[71] Jiang Zheng, Laurie A. Williams, Nachiappan Nagappan, Will Snipes, John P. Hudepohl, and Mladen A. Vouk. 2006. On the Value of Static Analysis for Fault Detection in Software. *IEEE Trans. Software Eng.* 32, 4 (2006), 240–253.