To What Extent Could We Detect Field Defects?

An Empirical Study of False Negatives in Static Bug Finding Tools

Ferdian Thung[1], Lucia[1], David Lo[1], Lingxiao Jiang[1],
Foyzur Rahman[2], and Premkumar T. Devanbu[2],
[1]Singapore Management University, Singapore
[2]University of California, Davis, USA
{ferdianthung,lucia.2009,davidlo,lxjiang}@smu.edu.sg,
{mfrahman,ptdevanbu}@ucdavis.edu

ABSTRACT

Software defects can cause much loss. Static bug-finding tools are believed to help detect and remove defects. These tools are designed to find programming errors; but, do they in fact help prevent actual defects that occur in the field and reported by users? If these tools had been used, would they have detected these field defects, and generated warnings that would direct programmers to fix them? To answer these questions, we perform an empirical study that investigates the effectiveness of state-of-the-art static bug finding tools on hundreds of reported and fixed defects extracted from three open source programs: Lucene, Rhino, and AspectJ. Our study addresses the question: *To what extent could field defects be found and detected by state-of-the-art static bug-finding tools?* Different from past studies that are concerned with the numbers of false positives produced by such tools, we address an orthogonal issue on the numbers of false negatives. We find that although many field defects could be detected by static bug finding tools, a substantial proportion of defects could not be flagged. We also analyze the types of tool warnings that are more effective in finding field defects and characterize the types of missed defects.

Categories and Subject Descriptors

D.2.5 [**Testing and Debugging**]: Debugging aids/Testing tools

General Terms

Experimentation, Measurement, Reliability

Keywords

Static bug-finding tools, field defects, false negatives

1. INTRODUCTION

Bugs are prevalent in many software systems. The National Institute of Standards and Technology (NIST) has estimated that bugs cost the US economy billions of dollars annually [47]. Bugs are not merely economically harmful; they can also harm life & property when mission critical systems malfunction. Clearly, techniques that can detect and reduce bugs would be very beneficial. To achieve this goal, many static analysis tools have been proposed to find bugs. Static bug finding tools, such as FindBugs [28], JLint [5], and PMD [14], have been shown to be helpful in detecting many bugs, even in mature software [8]. It is thus reasonable to believe that such tools are a useful adjunct to other bug finding techniques such as testing and inspection.

Although static bug finding tools are effective in some settings, it is unclear whether the warnings that they generate are really useful. Two issues are particularly important to be addressed: First, many warnings need to correspond to actual defects that would be experienced and reported by users. Second, many actual defects should be captured by the generated warnings. For the first issue, there have been a number of studies showing that the numbers of false warnings (or false positives) are too many, and some have proposed techniques to prioritize warnings [23–25, 43]. While the first issue has received much attention, the second issue has received less. Many papers on bug detection tools just report the number of defects that they can detect. It is unclear how many defects are missed by these bug detection tools. While the first issue is concerned with false positives, the second focuses on false negatives. We argue that *both* issues deserve equal attention as both have impact on the quality of software systems. If false positives are not satisfactorily addressed, this would make bug finding tools unusable. If false negatives are not satisfactorily addressed, the impact of these tools on software quality would be minimal. On mission critical systems, false negatives may even deserve more attention. Thus, there is a need to investigate the false negative rates of such tools on actual field defects.

Our study tries to fill this research gap by answering the following research question, and we use the term "bug" and "defect" interchangeably both of which refer to errors or flaws in a software:

> *To what extent could state-of-the-art static bug finding tools detect field defects?*

To investigate this research question, we make use of abundant data available in bug-tracking systems and software repositories. Bug-tracking systems, such as Bugzilla or JIRA, record descriptions of bugs that are actually experienced and reported by users. Software repositories contain information on what code elements get changed, removed, or added at different periods of time. Such information can be linked together to track bugs and when and how they get fixed. JIRA has the capability to link a bug report with the changed code that fixes the bug. Also, many techniques have been employed to link bug reports in Bugzilla to their corresponding SVN/CVS code changes [19,50]. These data sources provide us descriptions of actual field defects and their treatments. Based on the descriptions, we are able to infer root causes of defects (i.e., the faulty lines of code) from the bug treatments. To ensure accurate identification of faulty lines of code, we perform several iterations of manual inspections to identify lines of code that are responsible for the defects. Then, we are able to compare the identified root causes with the lines of code flagged by static bug finding tools, and to analyze the proportion of defects that are missed or captured by the tools.

In this work, we perform an exploratory study with three state-of-the-art static bug finding tools, FindBugs, PMD, and Jlint, on three reasonably large open source Java programs, Lucene, Rhino, and AspectJ. Lucene, Rhino, and AspectJ have 58, 35, and 9 committers respectively[1]. We use bugs reported in JIRA for Lucene version 2.9, and the iBugs dataset provided by Dallmeier and Zimmermann [19] for Rhino and AspectJ. Our manual analysis identifies 200 real-life defects that we can unambiguously locate faulty lines of code. We find that many of these defects could be detected by FindBugs, PMD, and JLint, but a number of them remain undetected.

The main contributions of this work are:

1. We examine the number of real-life defects missed by various static bug finding tools, and evaluate the tools' performance in terms of their false negative rates.
2. We investigate the warning families in various tools that are effective in detecting actual defects.
3. We characterize actual defects that could not be flagged by the static bug finding tools.

The paper is structured as follows. In Section 2, we present introductory information on various static bug finding tools. In Section 3, we present our experimental methodology. In Section 4, we present our empirical findings and discuss interesting issues. In Section 5, we describe related work. We conclude with future work in Section 6.

2. BUG FINDING TOOLS

In this section, we first provide a short survey of different bug finding tools that could be grouped into: static, dynamic, and machine learning based. We then present the 3 static bug finding tools that we evaluate in this study, namely FindBugs, AspectJ, and JLint.

2.1 Categorization of Bug Finding Tools

Many bug finding tools are based on static analysis techniques [38], such as type systems [36], constraint-based analysis [51], model checking [10, 15, 26], abstract interpretation [16, 17], or a combination of various techniques [9, 22, 29, 48]. They often produce various false positives, and in theory they should be free of false negatives for the kinds of defects they are designed to detect. However, due to implementation limitations and the fact that a large program often contains defect types that are beyond the designed capabilities of the tools, such tools may still suffer from false negatives with respect to all kinds of defects.

In this study, we analyze several static bug finding tools that make use of warning patterns for bug detection. These tools are lightweight and can scale to large programs. On the downside, these tools do not consider the specifications of a system, and may miss defects due to specification violations.

Other bug finding tools also use dynamic analysis techniques, such as dynamic slicing [49], dynamic instrumentation [37], directed random testing [12, 21, 44], and invariant detection [11, 20]. Such tools often explore particular parts of a program and produce no or few false positives. However, they seldom cover all parts of a program; they are thus expected to have false negatives.

There are also studies on bug prediction with data mining and machine learning techniques, which may have both false positives and negatives. For example, Sliwerski *et al.* [45] analyze code change patterns that may cause defects. Ostrand *et al.* [39] use a regression model to predict defects. Nagappan *et al.* [35] apply principal component analysis on the code complexity metrics of commercial software to predict failure-prone components. Kim *et al.* [32] predict potential faults from bug reports and fix histories.

2.2 FindBugs

FindBugs was first developed by Hovemeyer and Pugh [28]. It statically analyzes Java bytecode against various families of warnings characterizing common bugs in many systems. Code matching a set of warning patterns are flagged to the user, along with the specific locations of the code.

FindBugs comes with a lot of built-in warnings. These include: null pointer dereference, method not checking for null argument, close() invoked on a value that is always null, test for floating point equality, and many more. There are hundreds of warnings; these fall under a set of warning families including: correctness, bad practice, malicious code vulnerability, multi-threaded correctness, style, internationalization, performance, risky coding practice, etc.

2.3 JLint

JLint, developed by Artho *et al.*, is a tool to find defects, inconsistent code, and problems with synchronization in multi-threading applications [5]. Similar to FindBugs, JLint also analyzes Java bytecode against a set of warning patterns. It constructs and checks a lock-graph, and does data-flow analysis. Code fragments matching the warning patterns are flagged and outputted to the user along with their locations.

JLint provides many warnings such as potential deadlocks, unsynchronized method implementing 'Runnable' interface, method finalize() not calling super.finalize(), null reference, etc. These warnings are group under three families: synchronization, inheritance, and data flow.

2.4 PMD

PMD, developed by Copeland *i.e.*et al., is a tool that finds defects, dead code, duplicate code, sub-optimal code, and overcomplicated expressions [14]. Different from FindBugs and JLint, PMD analyzes Java source code rather than Java bytecode. PMD also comes with a set of warning patterns and finds locations in code matching these patterns.

PMD provides many warning patterns, such as jumbled incrementer, return from finally block, class cast exception

[1]These are counted based on information available on their respective websites and GitHub on June 28, 2012.

with toArray, misplaced null check, etc. These warning patterns fall into families, such as design, strict exceptions, clone, unused code, String and StringBuffer, security code, etc., which are referred to as *rule sets*.

3. METHODOLOGY

We make use of bug-tracking, version control, and state-of-the-art bug finding tools. First, we extract bugs and the faulty lines of code that are responsible for them. Next, we run bug finding tools for the various program releases before the bugs get fixed. Finally, we compare warnings given by bug finding tools and the real bugs.

3.1 Extraction of Faulty Lines of Code

We analyze two common configurations of bug tracking systems and code repositories to get historically faulty lines of code. One configuration is the combination of CVS as the source control repository, and Bugzilla as the bug tracking system. Another configuration is the combination of Git as the source control repository, and JIRA as the bug tracking system. We describe how these two configurations could be analyzed to extract root causes of fixed defects.

Data Extraction: CVS with Bugzilla. For the first configuration, Dallmeier and Zimmermann have proposed an approach to automatically analyze CVS and Bugzilla to link information [19]. Their approach is able to extract bug reports linked to corresponding CVS entries that fix the corresponding bugs. They are also able to download the code before and after the bug fix. The code to perform this has been publicly released for several software systems. As our focus is on defects, we remove bug reports that are marked as *enhancements* (i.e., they are new feature requests), and the CVS commits that correspond to them.

Data Extraction: Git cum JIRA. Git and JIRA have features that make it preferable over CVS and Bugzilla. JIRA bug tracking systems explicitly links bug reports to the revisions that fix the corresponding defects. From these fixes, we use Git diff to find the location of the buggy code and we download the revision prior to the fix by appending the "∧" symbol to the hashcode of the corresponding fix revision number. Again we remove bug reports and Git commits that are marked as *enhancements*.

Identification of Faulty Lines. The above process gives us a set of real-life defects along with the set of changes that fix them. To find the corresponding root causes, we perform a manual process based on the treatments of the defects. Kawrykow and Robillard have proposed an approach to remove non-essential changes [30] and convert "dirty" treatments to "clean" treatments. However, they still do not recover the root causes of defects.

Our process for locating root causes could not be easily automated by a simple *diff* operation between two versions of the systems (after and prior to the fix), due to the following reasons. *First*, not all changes fix the bug [30]; some, such as addition of new lines, removal of new lines, changes in indentations, etc., are only cosmetic changes that make the code aesthetically better. Figure 1 shows such an example. *Second*, even if all changes are essential, it is not straightforward to identify the defective lines from the fixes. Some fixes introduce additional code, and we need to find the corresponding faulty lines that are fixed by the additional code. We show several examples highlighting the process of extracting root causes from their treatments for a simple and a slightly more complicated case in Figures 2 & 3.

Figure 2 describes a bug fixing activity where one line of code is changed by modifying the operator == to !=. It is easy for us to identify the faulty line which is the line that gets changed. A more complicated case is shown in Figure 3. There are four sets of faulty lines that could be inferred from the *diff*: one is line 259 (marked with *), where an unnecessary method call needs to be removed; a similar fault is at line 262; the third set of faulty lines are at lines 838-340, and they are condition checks that should be removed; the fourth one is at line 887 and it misses a pre-condition check. For this case, the *diff* is much larger than the faulty lines that are manually identified.

Figure 3 illustrates the difficulties in automating the identification of faulty lines. To ensure the fidelity of identified root causes, we perform several iterations of manual inspections. For some ambiguous cases, several of the authors discussed and came to resolutions. Some cases that are still deemed ambiguous (i.e., it is unclear or difficult to manually locate the faulty lines) are removed from this empirical study. We show such an example in Figure 4. This example shows a bug fixing activity where an if block is inserted into the code and it is unclear which lines are really faulty.

At the end of the above process, we get the sets of faulty lines `Faulty` for all defects in our collection. Notation-wise, we refer to these sets of lines using an index notation `Faulty[]`. We refer to the set of lines in `Faulty` that correspond to the i^{th} defect as `Faulty[i]`.

3.2 Extraction of Warnings

To evaluate the static bug finding tools, we run the tools on the program versions before the defects get fixed. An ideal bug finding tool would recover the faulty lines of the program—possibly with other lines corresponding to other defects lying within the software system. Some of these warnings are false positives, while others are true positives.

For each defect, we extract the version of the program in the repository prior to the bug fix. We have such information already as an intermediate result for the root cause extraction process. We then run the bug finding tools on these program versions. Because 13.42% of these versions could not be compiled, we remove them from our analysis, which is a threat to validity of our study (see section 4.3).

As described in Section 2, each of the bug finding tools takes in a set of rules or the types of defects and flags them if found. By default, we enable all rules/types of defects available in the tools, except that we exclude two rule sets from PMD: one related to Android development, and the other whose XML configuration file could not be read by PMD (i.e., Coupling).

Each run would produce a set of warnings. Each warning flags a set of lines of code. Notation-wise, we refer to the sets of lines of code for all runs as `Warning`, and the set of lines of code for the i^{th} warning as `Warning[i]`. Also, we refer to the sets of lines of code for all runs of a particular tool T as $\texttt{Warning}_T$, and similarly we have $\texttt{Warning}_T\texttt{[i]}$.

3.3 Extraction of Missed Defects

With the faulty lines `Faulty` obtained through manual analysis, and the lines flagged by the static bug finding tools `Warning`, we can look for false negatives, *i.e.*, actual reported and fixed defects that are missed by the tools.

AspectJ-file name= org.aspectj/modules/org.aspectj.ajdt.core/src/org/aspectj/ajdt/internal/compiler/ast/ThisJoinPointVisitor.java	
Buggy version	**Fixed version**
Line 70: }	*//insert a line between line 70 and 71 in the buggy version*
Line 71: }	Line 72: //System.err.println("done: " + method);
Line 78: //System.err.println("isRef: " + expr + ", " + binding);	*Line is deleted*
Line 87: else if (isRef(ref, thisJoinPointStaticPartDec))	Line 89: } else if (isRef(ref, thisJoinPointStaticPartDec)) {

Figure 1: Example of Simple Cosmetic Changes

Lucene 2.9 - file name=src/java/org/apache/lucene/search/Scorer.java	
Buggy version	**Fixed version**
*Line 90: return doc == NO_MORE_DOCS	Line 90: return doc != NO_MORE_DOCS;

Figure 2: Identification of Root Causes (Faulty Lines) from Treatments [Simple Case]

AspectJ - file name= org.aspectj/modules/weaver/src/org/aspectj/weaver/bcel/LazyMethodGen.java	
Buggy version	**Fixed version**
*Line 259: lng.setStart(null);	Line is deleted
*Line 262: lng.setEnd(null);	Line is deleted
*Line 838: if (i instanceof LocalVariableInstruction) {	Line 836: if (localVariableStarts.get(lvt) == null) {
*Line 839: int index = ((LocalVariableInstruction)i).getIndex();	Line 837: localVariableStarts.put(lvt, jh);
*Line 840: if (lvt.getSlot() == index) {	Line 838: }
Line 841: if (localVariableStarts.get(lvt) == null) {	Line 839: localVariableEnds.put(lvt, jh);
Line 842: localVariableStarts.put(lvt, jh);	
Line 843: }	
Line 844: localVariableEnds.put(lvt, jh);	
Line 845: }	
Line 846: }	
Line 882: keys.addAll(localVariableStarts.keySet());	Line 870: keys.addAll(localVariableStarts.keySet());
Line 883: Collections.sort(keys,new Comparator() {	Line 871-874: *//these lines are commented codes*
Line 884: public int compare(Object a,Object b) {	Line 875: Collections.sort(keys, new Comparator() {
Line 885: LocalVariableTag taga = (LocalVariableTag)a;	Line 876: public int compare(Object a, Object b) {
Line 886: LocalVariableTag tagb = (LocalVariableTag)b;	Line 877: LocalVariableTag taga = (LocalVariableTag) a;
*Line 887: return taga.getName().compareTo(tagb.getName());	Line 878: LocalVariableTag tagb = (LocalVariableTag) b;
Line 888: }});	Line 879: if (taga.getName().startsWith("arg")) {
Line 889: for (Iterator iter = keys.iterator(); iter.hasNext();) {	Line 880: if (tagb.getName().startsWith("arg")) {
	Line 881: return -taga.getName().compareTo(tagb.getName());
	Line 882: } else {
	Line 883: return 1; // Whatever tagb is, it must come out before 'arg'
	Line 884: }
	Line 885: } else if (tagb.getName().startsWith("arg")) {
	Line 886: return -1; // Whatever taga is, it must come out before 'arg'
	Line 887: } else {
	Line 888: return -taga.getName().compareTo(tagb.getName());
	Line 889: }
	Line 890: }
	Line 891: });
	Line 892-894: *//these lines are commented codes*
	Line 895: for (Iterator iter = keys.iterator(); iter.hasNext();) {

Figure 3: Identification of Root Causes (Faulty Lines) from Treatments [Complex Case]

AspectJ - org.aspectj/modules/weaver/src/org/aspectj/weaver/patterns/SignaturePattern.java	
Buggy version	**Fixed version**
Line 87: }	*// Insert these 2 lines between line 87and 88 in the buggy version*
Line 88: if (!modifiers.matches(sig.getModifiers())) return false;	Line 88: if (kind == Member.ADVICE) return true;
Line 89:	Line 89: <new line>
Line 90: if (kind == Member.STATIC_INITIALIZATION) {	

Figure 4: Identification of Root Causes (Faulty Lines) from Treatments [Ambiguous Case]

To get these missed defects, for every i^{th} warning, we take the intersection of the sets `Faulty[i]` and `Warning[i]` from all bug finding tools, and the intersection between `Faulty[i]` with each $\texttt{Warning}_T$`[i]`. If an intersection is an empty set, we say that the corresponding bug finding tool *misses* the i^{th} defect. If the intersection covers a true subset of the lines in `Faulty[i]`, we say that the bug finding tool *partially captures* the i^{th} defect. Otherwise, if the intersection covers all lines in `Faulty[i]`, we say that the bug finding tool *fully captures* the i^{th} defect. We differentiate the partial and full cases as developers might be able to recover the other faulty lines, given that some of the faulty lines have been flagged.

3.4 Overall Approach

Our overall approach is illustrated by the pseudocode in Figure 5. Our approach takes in a bug repository (e.g., Bugzilla or JIRA), a code repository (e.g., CVS or Git), and a bug finding tool (e.g., FindBugs, PMD, or JLint). For each bug report in the repository, it performs three steps mentioned in previous sub-sections: Faulty lines extraction, warning identification, and missed defect detection.

The first step corresponds to lines 3-8. We find the bug fix commit corresponding to the bug report. We identify the version prior to the bug fix commit. We perform a *diff* to find the differences between these two versions. Faulty lines are then extracted by a manual analysis. The second step corresponds to lines 9-11. Here, we simply run the bug finding tools and collect lines of code flagged by the various warnings. Finally, step three is performed by lines 12-19. Here, we detect cases where the bug finding tool *misses*, *partially captures*, or *fully captures* a defect. The final statistics is output at line 20.

```
Procedure IdentifyMissedDefects
Inputs:
      BugRepo : Bug Repository
      CodeRepo : Code Repository
      BFTool : Bug Finding Tool
Output:
      Statistics of Defects that are Missed and
      Captured (Fully or Partially)
Method:
1:  Let Stats = {}
2:  For each bug report br in BugRepo
3:  // Step 1: Extract faulty lines of code
4:     Let fixC = br's corresponding fix commit in CodeRepo
5:     Let bugC = Revision before fixC in CodeRepo
6:     Let diff = The difference between fixC and bugC
7:     Extract faulty lines from diff
8:     Let Faulty_br = Faulty lines in bugC
9:  // Step 2: Get warnings
10:    Run BFTool on bugC
11:    Let Warning_br = Flagged lines in bugC by BFTool
12: // Step 3: Detect missed defects
13:    Let Common = Faulty_br ∩ Warning_br
14:    If Common = {}
15:       Add ⟨br, miss⟩ to Stats
16:    Else If Common = Faulty_br
17:       Add ⟨br, full⟩ to Stats
18:    Else
19:       Add ⟨br, partial⟩ to Stats
20: Output Stats
```

Figure 5: Identification of Missed Defects.

4. EMPIRICAL EVALUATION

In this section we present our research questions, datasets, empirical findings, and threats to validity.

4.1 Research Question & Dataset

We would like to answer the following research questions:

RQ1: How many real-life reported and fixed defects from Lucene, Rhino, and AspectJ are missed by state-of-the-art static bug finding tools?

RQ2: What types of warnings reported by the tools are most effective in detecting actual defects?

RQ3: What are some characteristics of the defects missed by the tools?

We evaluate three static bug finding tools, namely FindBugs, JLint, and PMD, on three open source projects: Lucene, Rhino, and AspectJ. Lucene is a general purpose text search engine library [1]. Rhino is an implementation of JavaScript written in Java [2]. AspectJ is an aspect-oriented extension of Java [3]. The average sizes of Lucene, Rhino, and AspectJ are around 265,822, 75,544, and 448,176 lines of code (LOC) respectively. We crawl JIRA for defects tagged for Lucene version 2.9. For Rhino and AspectJ, we analyze the iBugs repository prepared by Dallmeier and Zimmermann [19]. We show the numbers of unambiguous defects that we are able to manually locate root causes from the three datasets in Table 1 together with the total numbers of defects available in the datasets and the average faulty lines per defect.

Table 1: Number of Defects for Various Datasets

Dataset	# of Unambiguous Defects	# of Defects	Avg # of Faulty Lines Per Defect
Lucene	28	57	3.54
Rhino	20	32	9.1
AspectJ	152	350	4.07

4.2 Experimental Results

We answer the three research questions as follows.

4.2.1 RQ1: Number of Missed Defects

We show the number of missed defects by each and all of the three tools, for Lucene, Rhino, and AspectJ in Table 2. We do not summarize across software projects as the numbers of warnings for different projects differ greatly and any kind of summarization across projects may not be meaningful. We first show the results for all defects, and then zoom into subsets of the defects that span a small number of lines of code, and those that are severe. Our goal is to evaluate the effectiveness of state-of-the-art static bug finding tools in terms of their false negative rates. We would also like to evaluate whether false negative rates may be reduced if we use all three tools together.

4.2.1.1 All Defects.

Lucene. For Lucene, as shown in Table 2, we find that with all three tools, 35.7% of all defects could be fully identified (i.e., all faulty lines `Faulty[i]` of a defect i are flagged). An addition of 14.3% of the defects could also be partially identified (i.e., some but not all faulty lines in `Faulty[i]` are flagged). Still, 50% of the defects could not be flagged by the tools. Thus, the three tools are effective but are not very successful in capturing Lucene defects. Among the tools, PMD captures the most numbers of bugs, followed by FindBugs, and finally JLint.

Rhino. For Rhino, as shown in Table 2, we find that with all three tools, 95% of all defects could be fully identified. Only 5% of the defects could not be captured by the tools. Thus, the tools are very effective in capturing Rhino defects. Among the tools, FindBugs captures the most numbers of defects, followed by PMD, and finally JLint.

AspectJ. For AspectJ, as shown in Table 2, we find that with all three tools, 70.4% of all defects could be fully captured. Also, another 27.6% of the defects could be partially captured. Only 1.9% of the defects could not be captured by any of the tools. Thus, the tools are very effective in cap-

Table 2: Percentages of Defects that are Missed, Partially Captured, and Fully Captured. The numbers in the parentheses indicate the numbers of actual faulty lines captured, if any.

Tools vs.	Lucene			Rhino			AspectJ		
Programs	Miss	Partial	Full	Miss	Partial	Full	Miss	Partial	Full
FindBugs	71.4%	14.3%(5)	14.3%(8)	10.0%	0%	90.0%(179)	33.6%	19.7%(91)	46.7%(137)
JLint	82.1%	10.7%(3)	7.1%(4)	80.0%	10.0%(2)	10.0%(3)	97.4%	1.3%(2)	1.3%(2)
PMD	50.0%	14.3%(11)	35.7%(13)	15.0%	5.0%(18)	80.0%(157)	3.9%	27.6%(251)	68.4%(196)
All	50.0%	14.3%(11)	35.7%(13)	5.0%	0%	95.0%(181)	1.9%	27.6%(203)	70.4%(262)

turing AspectJ defects. PMD captures the most numbers of defects, followed by FindBugs, and finally JLint.

Also, to compare the bug finding tools, we show the average number of lines *in one defect program version* (over all defects) that are flagged by the tools for our subject programs in Table 3. We note that PMD flags the most numbers of lines of code, and likely produces more false positives than others, while JLint flags the least numbers of lines of code. With respect to the average sizes of programs (see Column "Avg # LOC" in Table 4), the tools may flag 0.2% to 56.9% of the whole program as buggy.

Table 3: Average Numbers of Lines Flagged Per Defect Version by Various Static Bug Finding Tools.

Tools vs. Programs	Lucene	Rhino	AspectJ
FindBugs	33,208.82	40,511.55	83,320.09
JLint	515.54	330.6	720.44
PMD	124,281.5	42,999.3	198,065.51
All	126,367.75	52,123.05	220,263

Note that there may be many other issues in a version besides the defect in our dataset in a program version, and a tool may generate many warnings for the version. These partially explain the high average numbers of lines flagged in Table 3. To further understand these numbers, we present more statistics about the warnings generated by the tools for each of the three programs in Table 4. Column "Avg # Warning" provides the average number of reported warnings. Column "Avg # Flagged Line Per Warning" is the average number of lines flagged by a warning. Column "Avg # Unfixed" is the average number of lines of code that are flagged but are not buggy lines fixed in a version. Column "Avg # LOC" is the number of lines of code in a particular software system (across all buggy versions). We notice that the average number of warnings generated by PMD is high. FindBugs reports less warnings but many warnings span many consecutive lines of code. Many flagged lines do not correspond to the buggy lines fixed in a version. These could either be false positives or bugs found in the future. We do not check the exact number of false positives though, as they require much manual labor (i.e., checking thousands of warnings in each of the hundreds of program versions), and they are the subject of other studies (e.g., [25]). On the other hand, the high numbers of flagged lines raise concerns with the effectiveness of warnings generated by the tools: Are the warnings effectively correlated with actual defects?

To answer such a question, we create random "bug" finding tools that would randomly flag some lines of code as buggy according to the distributions of the number of the lines flagged by each warning generated by each of the three bug finding tools, and compare the bug capturing effectiveness of such random tools with the actual tools. For each version of the subject programs and each of the actual tools, we run a random tool 10,000 times; each time the random tool would randomly generate the same number of warnings by following the distribution of the numbers of lines flagged by each warning; then, we count the numbers of missed, partially captured, and fully captured defects by the random tool; finally, we compare the effectiveness of the random tools with the actual tools by calculating the p-values as in Table 5. A value x in each cell in the table means that our random tool would have $x \times 100\%$ chance to get at least as good results as the actual tool for either partially or fully capturing the bugs. The values in the table imply the actual tools may indeed detect more bugs than random tools, although they may produce many false positives. However, some tools for some programs, such as PMD for Lucene, may not be much better than random tools. If there is no correlation between the warnings generated by the actual tools with actual defects, the tools should not perform differently from the random tools. Our results show that this is not the case at least for some tools with some programs.

Table 5: p-values among random and actual tools.

Tool	Program	Full	Partial or Full
FindBugs	Lucene	0.2766	0.1167
	Rhino	<0.0001	0.0081
	AspectJ	0.5248	0.0015
JLint	Lucene	0.0011	<0.0001
	Rhino	<0.0001	<0.0001
	AspectJ	0.0222	0.0363
PMD	Lucene	0.9996	1
	Rhino	0.9996	1
	AspectJ	0.9996	<0.0001

4.2.1.2 Localized Defects.

Many of the defects that we analyze span more than a few lines of code. We further focus only on defects that can be localized to a few lines of code (at most five lines of code, which we call *localized defects*), and investigate the effectiveness of the various bug finding tools. We show the numbers of missed localized defects by the three tools, for Lucene, Rhino, and AspectJ in Tables 6.

Lucene. Table 6 shows that the tools together fully identify 47.6% of all localized defects and partially identify another 14.3%. This is only slightly higher than the percentage for all defects (see Table 2). The ordering of the tools based on their ability to fully capture localized defects is the same.

Rhino. As shown in Table 6, all three tools together could fully identify 93.3% of all localized defects. This is slightly lower than the percentage for all defects (see Table 2).

AspectJ. Table 6 shows that the tools together fully capture 78.9% of all localized defects and miss only 0.8%. These are better than the percentages for all defects (see Table 2).

Table 4: Unfixed Warnings by Various Defect Finding Tools

Software	Tool	Avg # Warning	Avg # Flagged Line Per Warning	Avg # Unfixed	Avg # LOC
Rhino	FindBugs	177.9	838.72	40,502.6	75,543.8
	JLint	34	1	330.35	
	PMD	11,864.95	21.12	42,990.55	
	All	12,385.85	32.32	52,114.3	
Lucene	FindBugs	270.25	469.4	33,208.36	265,821.75
	JLint	685.21	1	515.29	
	PMD	39,993.68	11.74	124,280.64	
	All	40,949.14	14.58	126,366.89	
AspectJ	FindBugs	802.29	381.76	83,318.59	448,175.94
	JLint	3,937	1	720.41	
	PMD	73,641.86	2.94079	198,062.57	
	All	78,381.15	3.05921	220,260.31	

Table 6: Percentages of Localized Defects that are Missed, Partially Captured, and Fully Captured

Tools vs. Programs	Lucene			Rhino			AspectJ		
	Miss	Partial	Full	Miss	Partial	Full	Miss	Partial	Full
FindBugs	66.7%	14.3%	19.0%	13.3%	0%	86.7%	32.0%	14.8%	53.1%
JLint	80.9%	9.5%	9.5%	86.7%	0%	13.3%	96.9%	1.6%	1.6%
PMD	38.1%	14.3%	47.6%	20.0%	0%	80.0%	2.3%	21.1%	76.6%
All	38.1%	14.3%	47.6%	6.7%	0%	93.3%	0.8%	20.3%	78.9%

4.2.1.3 *Severe Defects.*

Different defects are often labeled with different severity levels. We further focus only on defects at severe levels: `blocker`, `critical`, and `major`, which we collectively call *severe defects*, and investigate the effectiveness of the various bug finding tools on detecting severe defects. We show the numbers of missed severe defects by the three tools, for Lucene, Rhino, and AspectJ in Table 7.

Lucene. As shown in Table 7, all three tools together could fully identify 58.3% of all severe bugs. This is higher than the percentage for all bugs (see Table 2). Considering severe defects in Lucene only, FindBugs is equally good as JLint.

Rhino. As shown in Table 7, all three tools together could fully identify all severe defects. Note that although the percentage is very good, there are only two severe defects in the Rhino dataset, and JLint misses both of these two defects.

AspectJ. As shown in Table 7, all three tools together could fully capture 61.3% of all severe defects and only miss 3.2%. The relative performance of the three tools is the same as the relative performance for all defects (see Table 2): PMD, followed by, FindBugs, followed by JLint. JLint could not fully capture any severe defect in the AspectJ dataset.

4.2.1.4 *Stricter Defects.*

We also perform a stricter analysis that requires not only that the faulty lines are covered by the warnings, but also the type of the warnings must be directly related to the faulty lines. For example, if the faulty line is a null pointer dereference, then the warning must explicitly say so. Again we manually analyze to see if the warnings *strictly captures* the defect. We focus on defects that are localized to one line of code due to limited manpower. There are 7 bugs, 5 bugs, and 66 bugs for Lucene, Rhino, and AspectJ respectively that can be localized to one line of code.

We show the results of our analysis for Lucene, Rhino, and AspectJ in Table 8. We notice that under this stricter requirement, very few of the defects fully captured by the tools (see Column "Full") are strictly captured by the same tools (see Column "Strict"). Thus, although the faulty lines might be flagged by the tools, the warning messages from the tools may not have sufficient information for the developers to understand the defects.

Table 8: Numbers of One-Line Defects that are Strictly Captured versus Fully Captured

Tool	Lucene		Rhino		AspectJ	
	Strict	Full	Strict	Full	Strict	Full
FindBugs	0	4	1	4	5	42
PMD	0	7	0	5	1	65
JLint	0	0	0	1	1	2

4.2.2 *RQ2: Effectiveness of Different Warnings*

We show the effectiveness of various warning families of FindBugs, PMD, and JLint in flagging defects in Tables 9, 10, and 11 respectively. We highlight the top-5 warning families in terms of their ability in fully capturing the root causes of the defects.

FindBugs. For FindBugs, as shown in Table 9, in terms of low false negative rates, we find that the best warning families are: Style, Performance, Malicious Code, Bad Practice, and Correctness. Warnings in the style category include switch statement having one case branch to fall through to the next case branch, switch statement having no default case, assignment to a local variable which is never used, unread public or protected field, a referenced variable contains null value, etc. Warnings belonging to the malicious code category include a class attribute should be made final, a class attribute should be package protected, etc. Furthermore, we find that a violation of each of these warnings would likely flag an entire class that violates it. Thus, a lot of lines of code would be flagged, making it having a higher chance of capturing the defects. Warnings belonging to the bad practice category include comparison of String objects using == or !=, a method ignores exceptional return value, etc. Performance related warnings include method

Table 7: Percentages of Severe Defects that are Missed, Partially Captured, and Fully Captured

Tools vs. Programs	Lucene			Rhino			AspectJ		
	Miss	Partial	Full	Miss	Partial	Full	Miss	Partial	Full
FindBugs	83.3%	0%	16.7%	0%	0%	100.0%	35.5%	25.8%	38.7%
JLint	83.3%	0%	16.7%	100.0%	0%	0%	96.8%	3.2%	0%
PMD	33.3%	8.3%	58.3%	0%	0%	100.0%	3.2%	35.5%	61.3%
All	33.3%	8.3%	58.3%	0%	0%	100.0%	3.2%	35.5%	61.3%

concatenates strings using + instead of StringBuffer, etc. Correctness related warnings include a field that masks a superclass's field, invocation of toString to an array, etc.

Table 9: Percentages of Defects that are Missed, Partially Captured, and Fully Captured for Different Warning Families of FindBugs

Warning Family	Miss	Partial	Full
Style	46.5%	9.5%	44.0%
Performance	62.0%	9.0%	29.0%
Malicious Code	68.5%	7.5%	24.0%
Bad Practise	73.5%	4.5%	22.0%
Correctness	74.0%	6.5%	19.5%

PMD. For PMD, as shown in Table 10, we find that the warning categories that are the most effective are: Code Size, Design, Controversial, Optimization, and Naming. Code size warnings include problems related to a code being too large or complex, e.g., number of acyclic execution paths is more than 200, method length is long, etc. Code size is correlated with defects but does not really inform the types of defects that needs a fix. Design warnings identify suboptimal code implementation; these include the simplification of boolean return, missing default case in a switch statement, deeply nested if statement, etc which may not be correlated with defects. Controversial warnings include unnecessary constructor, null assignment, assignments in operands, etc. Optimization warnings include best practices to improve the efficiency of the code. Naming warnings include rules pertaining to the preferred names of various program elements.

Table 10: Percentages of Warnings that are Missed, Partially Captured, and Fully Captured for Different Warning Families of PMD

Warning Family	Miss	Partial	Full
Code Size	21.5%	5.0%	73.5%
Design	30.0%	8.5%	61.5%
Controversial	27.5%	19.5%	53.0%
Optimization	38.5%	23.0%	38.5%
Naming	60.0%	23.0%	17.0%

JLint. For JLint, as shown in Table 11, we have three categories: Inheritance, Synchronization, and Data Flow. We find that inheritance is more effective than data flow which in turn is more effective than synchronization in detecting defects. Inheritance warnings relate to class inheritance issues, such as: new method in a sub-class is created with identical name but different parameters as one inherited from the super class, etc. Synchronization warnings relate to erroneous multi-threading applications in particular problems related to conflicts on shared data usage by multiple threads, such as potential deadlocks, required but missing synchronized keywords, etc. Data flow warnings relate to problems that JLint detects by performing data flow analysis on Java bytecode, such as null referenced variable, type cast misuse, comparison of String with object references, etc.

Table 11: Percentages of Defects that are Missed, Partially Captured, and Fully Captured for Different Warning Families of JLint

Warning Family	Miss	Partial	Full
Inheritance	93.0%	5.0%	2.0%
Data Flow	93.5%	5.0%	1.5%
Synchronization	99.5%	0%	0.5%

4.2.3 RQ3: Characteristics of Missed Defects

There are 19 defects that are missed by all three tools. These defects involve logical or functionality errors and thus they are difficult to be detected by static bug finding tools without the knowledge of the specification of the systems. We can categorize the defects into several categories: method calls (addition, removal of method calls, changes to parameters passed in to methods), pre-condition checks (addition, removal, or changes to pre-condition checks), assignment operations (wrong value being assigned to variables), and others (missing type cast, etc). Often, one defect can be classified into different categories.

A sample defect missed by all three tools is shown in Figure 6, which involves an invocation of a wrong method.

4.3 Threats to Validity

Threat to internal validity includes experimental bias. We manually extract faulty lines from changes that fix them that might have error prone. To reduce this threat, we have checked and refined the result of the analysis several times. We also exclude the defects that ambiguously localize to a set of faulty lines of code. Also, we exclude some versions of our subject programs that cannot be compiled. There might also be implementation errors in various scripts and code used to collect and evaluate bugs.

Threat to external validity is related to the generalizability of our findings. In this work, we only analyze three static bug finding tools and three open source Java programs. We analyze only one version of Lucene dataset; for Rhino and AspectJ, we only analyze defects that are available in iBugs. Also, we investigate only defects that get reported and fixed. In the future, we could analyze more programs and more defects to reduce selection bias. Also, we plan to investigate more bug finding tools and programs in various languages.

5. RELATED WORK

We summarize related studies on bug finding, warning prioritization, bug triage, and empirical study on defects.

5.1 Bug Finding

There are many bug finding tools proposed in the literature [12, 21, 26, 36, 38, 44, 45]. A short survey of these tools

Rhino - mozilla/js/rhino/xmlimplsrc/org/mozilla/javascript/xmlimpl/XML.java	
Buggy version	**Fixed version**
Line 3043: return createEmptyXML(lib);	Line 3043: return createFromJS(lib, "");

Figure 6: Example of A Defect Missed By All Tools

has been provided in Section 2. In this study, we focus on three static bug finding tools for Java, including FindBugs, PMD, and JLint. These tools are relatively lightweight, and have been applied to large systems.

None of these bug finding tools is able to detect all kinds of bugs. To the best of our knowledge, there is no study on the false negative rates of these tools. We are the first to carry out an empirical study to answer this issue based on a few hundreds of real-life bugs from three Java programs. In the future, we plan to investigate other bug finding tools with more programs written in different languages.

5.2 Warning Prioritization

Many studies deal with the many false positives produced by various bug finding tools. Kremenek et al. use *z-ranking* to prioritize warnings [33]. Kim et al. prioritize warning categories using historical data [31]. Ruthruff et al. predict actionable static analysis warnings by proposing a logistic regression model that differentiates false positives from actionable warnings [43]. Liang et al. propose a technique to construct a training set for better prioritization of static analysis warnings [34]. A comprehensive survey of static analysis warnings prioritization has been written by Heckman and Williams [25]. There are large scale studies on the false positives produced by FindBugs [6, 7].

While past studies on warning prioritization focus on coping with *false positives* with respect to *actionable* warnings, we investigate an orthogonal problem on *false negatives*. False negatives are important as it can cause bugs to go unnoticed and cause harm when the software is used by end users. Analyzing false negatives is also important to guide future research on building additional bug finding tools.

5.3 Bug Triage

Even when the bug reports, either from a tool or from a human user, are all for real bugs, the sheer number of reports can be huge for a large system. In order to allocate appropriate development and maintenance resources, project managers often need to triage the reports.

Cubranic *et al.* [18] propose a method that uses text categorization to triage bug reports. Anvik *et al.* [4] use machine learning techniques to triage bug reports. Hooimeijer *et al.* [27] construct a descriptive model to measure the quality of bug reports. Park *et al.* [41] propose *CosTriage* that further takes the cost associated with bug reporting and fixing into consideration. Sun *et al.* [46] use data mining techniques to detect duplicate bug reports.

Different from warning prioritization, these studies address the problem of *too many true positives*. They are also different from our work which deals with false negatives.

5.4 Empirical Studies on Defects

Many studies investigate the nature of defects. Pan et al. investigate different bug fix patterns for various software systems [40]. They highlight patterns such as method calls with different actual parameter values, change of assignment expressions, etc. Chou et al. perform an empirical study of operating system errors [13].

Closest to our work is the study by Rutar et al. on the comparison of number of warnings generated by various bug finding tools in Java [42]. However, no analysis have been made as to whether there are false positives or false negatives. The authors commented that: "An interesting area of future work is to gather extensive information about the actual faults in programs, which would enable us to precisely identify false positives and false negatives." In this study, we address the false negatives mentioned by them.

6. CONCLUSION AND FUTURE WORK

Defects can harm software houses and end-users. A number of bug finding tools have been proposed to catch the defects. In this work, we empirically study the effectiveness of several state-of-the-art static bug finding tools in preventing real-life defects. We investigate 3 bug finding tools, FindBugs, JLint, and PMD, on 3 programs, Lucene, Rhino, and AspectJ. We analyze 200 fixed defects and extract faulty lines of code responsible for these defects from their treatments. We find that for Rhino and AspectJ, most defects could be *partially or fully captured* by combining the bug finding tools. For Lucene, a substantial proportion of defects (i.e., 50%) are *missed*. We find that FindBugs and PMD are the best among the three bug finding tools in preventing false negatives. However, these two tools flag more lines of code than JLint. Our stricter analysis sheds light that although many of these warnings cover faulty lines, often the warnings are too generic and developers need to inspect the code to find the defects. We find that some bugs are not flagged by any of the three bug finding tools – these bugs involve logical or functionality errors that are difficult to be detected without any specification of the system.

As a future work, we would like to build a new static analysis tool that automatically extracts the faulty lines of code responsible for the defects from their symptoms.

7. ACKNOWLEDGMENT

We thank the researchers creating and maintaining the iBugs repository which is publicly available at `http://www.st.cs.uni-saarland.de/ibugs/`. We appreciate very much the valuable comments from anonymous reviewers and our shepherd Andreas Zeller for improving this paper.

8. REFERENCES

[1] Apache Lucene—Apache Lucene Core. `http://lucene.apache.org/core/`.

[2] Rhino—JavaScript for Java. `http://www.mozilla.org/rhino/`.

[3] The AspectJ Project. `http://www.eclipse.org/aspectj/`.

[4] J. Anvik, L. Hiew, and G. C. Murphy. Who should fix this bug? In *ICSE*, pages 361–370, 2006.

[5] C. Artho. Jlint - find bugs in java programs. http://jlint.sourceforge.net/, 2006.

[6] N. Ayewah, D. Hovemeyer, J. D. Morgenthaler, J. Penix, and W. Pugh. Using static analysis to find bugs. *IEEE Software*, 25(5):22–29, 2008.

[7] N. Ayewah and W. Pugh. The google findbugs fixit. In *ISSTA*, pages 241–252, 2010.

[8] N. Ayewah, W. Pugh, J. D. Morgenthaler, J. Penix, and Y. Zhou. Evaluating static analysis defect warnings on production software. In *PASTE*, pages 1–8, 2007.

[9] T. Ball, V. Levin, and S. K. Rajamani. A decade of software model checking with slam. *Commun. ACM*, 54(7):68–76, 2011.

[10] D. Beyer, T. A. Henzinger, R. Jhala, and R. Majumdar. The software model checker blast. *STTT*, 9(5-6):505–525, 2007.

[11] Y. Brun and M. D. Ernst. Finding latent code errors via machine learning over program executions. In *ICSE*, pages 480–490, 2004.

[12] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler. EXE: Automatically generating inputs of death. *ACM Trans. Inf. Syst. Secur.*, 12(2), 2008.

[13] A. Chou, J. Yang, B. Chelf, S. Hallem, and D. R. Engler. An empirical study of operating system errors. In *SOSP*, pages 73–88, 2001.

[14] T. Copeland. *PMD Applied*. Centennial Books, 2005.

[15] J. C. Corbett, M. B. Dwyer, J. Hatcliff, S. Laubach, C. S. Pasareanu, Robby, and H. Zheng. Bandera: extracting finite-state models from java source code. In *ICSE*, pages 439–448, 2000.

[16] P. Cousot and R. Cousot. An abstract interpretation framework for termination. In *POPL*, pages 245–258, 2012.

[17] P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, and X. Rival. Why does astrée scale up? *Formal Methods in System Design*, 35(3):229–264, 2009.

[18] D. Cubranic and G. C. Murphy. Automatic bug triage using text categorization. In *SEKE*, pages 92–97, 2004.

[19] V. Dallmeier and T. Zimmermann. Extraction of bug localization benchmarks from history. In *ASE*, pages 433–436, 2007.

[20] M. Gabel and Z. Su. Online inference and enforcement of temporal properties. In *ICSE*, pages 15–24, 2010.

[21] P. Godefroid, N. Klarlund, and K. Sen. Dart: directed automated random testing. In *PLDI*, pages 213–223, 2005.

[22] GrammaTech. Codesonar. `http://www.grammatech.com/products/codesonar/overview.html`.

[23] S. S. Heckman. Adaptively ranking alerts generated from automated static analysis. *ACM Crossroads*, 14(1), 2007.

[24] S. S. Heckman and L. A. Williams. A model building process for identifying actionable static analysis alerts. In *ICST*, pages 161–170, 2009.

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

[26] G. J. Holzmann, E. Najm, and A. Serhrouchni. Spin model checking: An introduction. *STTT*, 2(4):321–327, 2000.

[27] P. Hooimeijer and W. Weimer. Modeling bug report quality. In *ASE*, pages 34–43, 2007.

[28] D. Hovemeyer and W. Pugh. Finding bugs is easy. In *OOPSLA*, 2004.

[29] IBM. T.J. Watson Libraries for Analysis (WALA). `http://wala.sourceforge.net`.

[30] D. Kawrykow and M. P. Robillard. Non-essential changes in version histories. In *ICSE*, 2011.

[31] S. Kim and M. D. Ernst. Prioritizing warning categories by analyzing software history. In *MSR*, 2007.

[32] S. Kim, T. Zimmermann, E. J. W. Jr., and A. Zeller. Predicting faults from cached history. In *ISEC*, pages 15–16, 2008.

[33] T. Kremenek and D. R. Engler. Z-ranking: Using statistical analysis to counter the impact of static analysis approximations. In *SAS*, 2003.

[34] G. Liang, L. Wu, Q. Wu, Q. Wang, T. Xie, and H. Mei. Automatic construction of an effective training set for prioritizing static analysis warnings. In *ASE*, 2010.

[35] N. Nagappan, T. Ball, and A. Zeller. Mining metrics to predict component failures. In *ICSE*, pages 452–461, 2006.

[36] G. C. Necula, J. Condit, M. Harren, S. McPeak, and W. Weimer. Ccured: type-safe retrofitting of legacy software. *ACM Trans. Program. Lang. Syst.*, 27(3):477–526, 2005.

[37] N. Nethercote and J. Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *PLDI*, pages 89–100, 2007.

[38] F. Nielson, H. R. Nielson, and C. Hankin. *Principles of Program Analysis*. Springer, 2005.

[39] T. J. Ostrand, E. J. Weyuker, and R. M. Bell. Predicting the location and number of faults in large software systems. *IEEE Trans. Softw. Eng.*, 31:340–355, April 2005.

[40] K. Pan, S. Kim, and E. J. W. Jr. Toward an understanding of bug fix patterns. *Empirical Software Engineering*, 14(3):286–315, 2009.

[41] J.-W. Park, M.-W. Lee, J. Kim, S. won Hwang, and S. Kim. CosTriage: A cost-aware triage algorithm for bug reporting systems. In *AAAI*, 2011.

[42] N. Rutar, C. B. Almazan, and J. S. Foster. A comparison of bug finding tools for java. In *ISSRE*, pages 245–256, 2004.

[43] J. R. Ruthruff, J. Penix, J. D. Morgenthaler, S. G. Elbaum, and G. Rothermel. Predicting accurate and actionable static analysis warnings: an experimental approach. In *ICSE*, pages 341–350, 2008.

[44] K. Sen, D. Marinov, and G. Agha. CUTE: a concolic unit testing engine for C. In *ESEC/SIGSOFT FSE*, pages 263–272, 2005.

[45] J. Śliwerski, T. Zimmermann, and A. Zeller. When do changes induce fixes? *SIGSOFT Softw. Eng. Notes*, 30:1–5, 2005.

[46] C. Sun, D. Lo, X. Wang, J. Jiang, and S.-C. Khoo. A discriminative model approach for accurate duplicate bug report retrieval. In *ICSE*, pages 45–54, 2010.

[47] G. Tassey. The economic impacts of inadequate infrastructure for software testing. *National Institute of Standards and Technology. Planning Report 02-3.2002*, 2002.

[48] W. Visser and P. Mehlitz. Model checking programs with Java PathFinder. In *SPIN*, 2005.

[49] D. Weeratunge, X. Zhang, W. N. Sumner, and S. Jagannathan. Analyzing concurrency bugs using dual slicing. In *ISSTA*, pages 253–264, 2010.

[50] R. Wu, H. Zhang, S. Kim, and S.-C. Cheung. Relink: recovering links between bugs and changes. In *SIGSOFT FSE*, pages 15–25, 2011.

[51] Y. Xie and A. Aiken. Saturn: A scalable framework for error detection using boolean satisfiability. *ACM Trans. Program. Lang. Syst.*, 29(3), 2007.