Autom Softw Eng (2015) 22:561–602
DOI 10.1007/s10515-014-0169-8

To what extent could we detect field defects? An extended empirical study of false negatives in static bug-finding tools

Ferdian Thung · Lucia · David Lo · Lingxiao Jiang · Foyzur Rahman · Premkumar T. Devanbu

Received: 7 September 2013 / Accepted: 16 August 2014 / Published online: 19 September 2014

Abstract Software defects can cause much loss. Static bug-finding tools are designed to detect and remove software defects and believed to be effective. However, do such tools 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 five state-of-the-art static bug-finding tools (FindBugs, JLint, PMD, CheckStyle, and JCSC) 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 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

F. Thung (✉) · Lucia · D. Lo · L. Jiang
School of Information Systems, Singapore Management University, Singapore, Singapore
e-mail: ferdiant.2013@phdis.smu.edu.sg

Lucia
e-mail: lucia.2009@phdis.smu.edu.sg

D. Lo
e-mail: davidlo@smu.edu.sg

L. Jiang
e-mail: lxjiang@smu.edu.sg

F. Rahman · P. T. Devanbu
University of California, Davis, CA, USA
e-mail: mfrahman@ucdavis.edu

P. T. Devanbu
e-mail: ptdevanbu@ucdavis.edu

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. Furthermore, we analyze the effectiveness of the tools in finding field defects of various severities, difficulties, and types.

Keywords False negatives · Static bug-finding tools · Empirical study

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 (Tassey 2002). Bugs are not merely economically harmful; they can also harm life and properties 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 (Hovemeyer and Pugh 2004), JLint (Artho 2006), PMD (Copeland 2005), CheckStyle (Burn 2007), and JCSC (Jocham 2005), have been shown to be helpful in detecting many bugs, even in mature software (Ayewah et al. 2007). 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 (i.e., false positives) are too many, and some have proposed techniques to prioritize warnings (Heckman and Williams 2011, 2009; Ruthruff et al. 2008; Heckman 2007). 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 (i.e., false negatives). 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 question. In this, we use the term "bug" and "defect" interchangeably, both of which refer to errors or flaws in software:

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

To answer this question, we make use of 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 (Dallmeier and Zimmermann 2007; Wu et al. 2011a). 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 field defects that are missed or captured by the tools.

In this work, we perform an exploratory study with five state-of-the-art static bug-finding tools (FindBugs, PMD, Jlint, CheckStyle, and JCSC) on three reasonably large open source Java programs (Lucene, Rhino, and AspectJ). We use bugs reported in JIRA for Lucene version 2.9, and the iBugs dataset provided by Dallmeier and Zimmermann (2007) for Rhino and AspectJ. Our manual analysis identifies 200 real-life defects that we can unambiguously locate faulty lines of code. We analyze these bugs to answer the following questions that elaborate the main question that we present earlier:

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?
RQ4 How effective are the tools in finding defects of various severity?
RQ5 How effective are the tools in finding defects of various difficulties?
RQ6 How effective are the tools in finding defects of various types?

The main contributions of this work are as follows:

1. We examine the number of real-life defects missed by five 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.
4. We analyze the effectiveness of the static bug-finding tools on defects of various severities, difficulties, and types.

The paper is structured as follows. In Sect. 2, we present introductory information on various static bug-finding tools. In Sect. 3, we present our experimental methodology. In Sect. 4, we describe the details of the projects and the defects that we analyze in this work. In Sects. 5–10, we present the answers to the six research questions. We present threats to validity in Sect. 11. In Sect. 12, we describe related work. We conclude with future work in Sect. 13.

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 five static bug-finding tools that we evaluate in this study, namely FindBugs, JLint, PMD, CheckStyle and JCSC.

2.1 Categorization of bug-finding tools

Many bug-finding tools are based on static analysis techniques (Nielson et al. 2005), such as type systems (Necula et al. 2005), constraint-based analysis (Xie and Aiken 2007), model checking (Holzmann et al. 2000; Corbett et al. 2000; Beyer et al. 2007), abstract interpretation (Cousot et al. 2009; Cousot and Cousot 2012), or a combination of various techniques (Ball et al. 2011; GrammaTech 2012; Visser and Mehlitz 2005; IBM 2012). 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 five 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. We pick these five bug finding tools since they are freely and publicly available. They are also popular and have been downloaded for thousands of times. They are well maintained and many versions of these tools have been released to the public. Also, these tools have been used in many past studies, e.g., (Kim and Ernst 2007; Ruthruff et al. 2008; Spacco et al. 2006; Liang et al. 2010; Ayewah et al. 2007).

Other bug-finding tools also use dynamic analysis techniques, such as dynamic slicing (Weeratunge et al. 2010), dynamic instrumentation (Nethercote and Seward 2007), directed random testing (Sen et al. 2005; Godefroid et al. 2005; Cadar et al. 2008), and invariant detection (Brun and Ernst 2004; Gabel and Su 2010). 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, Śliwerski et al. (2005) analyze code change patterns that may cause defects. Ostrand et al. (2005) use a regression model to predict defects. Nagappan et al. (2006) apply principal component analysis on the code complexity metrics of commercial software to predict failure-prone components. Kim et al. (2008) predict potential faults from bug reports and fix histories.

2.2 FindBugs

FindBugs was first developed by Hovemeyer and Pugh (2004). 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 (2006), is a tool to find defects, inconsistent code, and problems with synchronization in multi-threading applications. 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 grouped under three families: synchronization, inheritance, and data flow.

2.4 PMD

PMD, developed by Copeland (2005), is a tool that finds defects, dead code, duplicate code, sub-optimal code, and overcomplicated expressions. 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 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*.

2.5 CheckStyle

CheckStyle, developed by Burn (2007), is a tool to write Java code that enforces a coding standard; default configuration is the Sun Code Conventions. Similar to PMD, CheckStyle analyzes Java source code. CheckStyle also comes with a set of warning patterns and finds locations in code matching these patterns.

CheckStyle provides many warning patterns, such as layout issues, class design problems, duplicate code, Javadoc comments, metrics, modifiers, naming conventions, regular expression, size violations, whitespace, etc. These warning patterns fall into families, such as sizes, regular expression, whitespace, Java documentation, etc.

2.6 JCSC

JCSC, developed by Jocham (2005), is a tool to write Java code that enforces a customized coding standard and also check for potential bad code. Similar to CheckStyle, the default coding standard is the Sun Code Conventions. The standard can be defined for naming conventions for class, interfaces, fields, parameter, structural layout of the type (class/interface), etc. Similar to PMD and CheckStyle, JCSC analyzes Java source code. It also comes with a set of warning patterns and finds locations in code matching these patterns.

JCSC provides many warning patterns, such as empty catch/finally block, switch without default, slow code, inner classes in class/interface issues, constructor or method or field declaration issues, etc. These warning patterns fall into families, such as method, metrics, field, Java documentation, etc.

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 the bugs. 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 to find false negatives.

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.

3.1.1 Data extraction: CVS with Bugzilla

For the first configuration, (Dallmeier and Zimmermann 2007) have proposed an approach to automatically analyze CVS and Bugzilla to link information. Their approach is able to extract issue reports linked to corresponding CVS commit entries that correspond to the reports. They are also able to download the code before and after each change. The code to perform this has been publicly released for several software systems. As our focus is on defects, we remove issue reports that are marked as *enhancements* (i.e., they are not bug fixes) and the CVS commits that correspond to them.

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: }
 Line 71: }
	//insert a line between line 70 and 71 in the buggy version
	 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)) {

Fig. 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;

Fig. 2 Identification of root causes (Faulty Lines) from treatments [Simple Case]

3.1.2 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*.

3.1.3 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 (2011) have proposed an approach to remove non-essential changes 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. Firstly, not all changes fix the bug (Kawrykow and Robillard 2011); 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. Secondly, 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 Figs. 2 and 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 Fig. 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.

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();) {

Fig. 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) {	

Fig. 4 Identification of root causes (Faulty Lines) from treatments [Ambiguous Case]

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 Fig. 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 ith 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 Sect. 11).

As described in Sect. 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 ith 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.

To get these missed defects, for every ith 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 $\texttt{Warning}_T\texttt{[i]}$. If an intersection is an empty set, we say that the corresponding bug-finding tool *misses* the ith defect. If the intersection covers a true subset of the lines in `Faulty[i]`, we say that the bug-finding tool *partially captures* the ith defect. Otherwise, if the intersection covers all lines in `Faulty[i]`, we say that the bug-finding tool *fully captures* the ith 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 Fig. 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, JLint, CheckStyle, or JCSC). 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

```
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 $Common = Faulty_{br} \cap Warning_{br}$
14:     If $Common = \{\}$
15:         Add $\langle br, miss \rangle$ to $Stats$
16:     Else If $Common = Faulty_{br}$
17:         Add $\langle br, full \rangle$ to $Stats$
18:     Else
19:         Add $\langle br, partial \rangle$ to $Stats$
20: Output $Stats$
```

Fig. 5 Identification of missed defects

tool *misses*, *partially captures*, or *fully captures* a defect. The final statistics is output at line 20.

4 Study subjects

We evaluate five static bug-finding tools, namely FindBugs, JLint, PMD, CheckStyle, and JCSC on three open source projects: Lucene, Rhino, and AspectJ. Lucene from Apache Software Foundation[1] is a general purpose text search engine library. Rhino from Mozilla Foundation[2] is an implementation of JavaScript written in Java. AspectJ from Eclipse Foundation[3] is an aspect-oriented extension of Java. 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 (2007). 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.

[1] http://lucene.apache.org/core/

[2] http://www.mozilla.org/rhino/

[3] http://www.eclipse.org/aspectj/

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

5 RQ1: number of missed defects

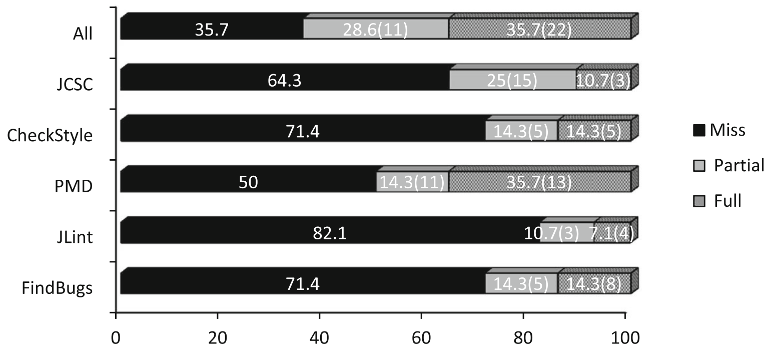
We show the number of missed defects by each and all of the five tools, for Lucene, Rhino, and AspectJ in Fig. 6. We do not summarize across software projects as the numbers of warnings for different projects differ greatly and summarization across projects may not be meaningful. We first show the results 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 five tools together.

5.1 Consider all defects

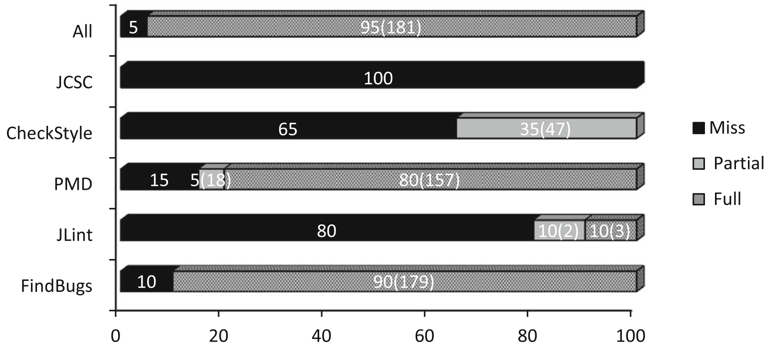
Lucene For Lucene, as shown in Fig. 6, we find that with all five tools, 64.3 % of all defects can be identified. Still, the remaining defects could not be flagged by the tools. Thus, the five tools are quite effective but are not very successful in capturing Lucene defects. Among the tools, PMD captures the most numbers of bugs, followed by FindBugs, CheckStyle, JCSC, and finally JLint.
Rhino For Rhino, as shown in Fig. 6, we find that with all five tools, the majority of all defects could be fully identified. Only 5 % of all 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, CheckStyle, JLint, and finally JCSC.
AspectJ For AspectJ, as shown in Fig. 6, we find that with all five tools, the majority of all defects could be partially or fully identified. Only 0.7 % of the defects could not be captured by any of the tools. Thus, the tools are very effective in capturing AspectJ defects. PMD captures the most numbers of defects, followed by CheckStyle, FindBugs, JCSC, and finally JLint.

Also, to compare the five bug-finding tools, we show the average numbers of lines *in one defect program version* (over all defects) that are flagged by the tools for our subject programs in Table 2. 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 3), the tools may flag 0.2–56.9 % of the whole program as buggy.

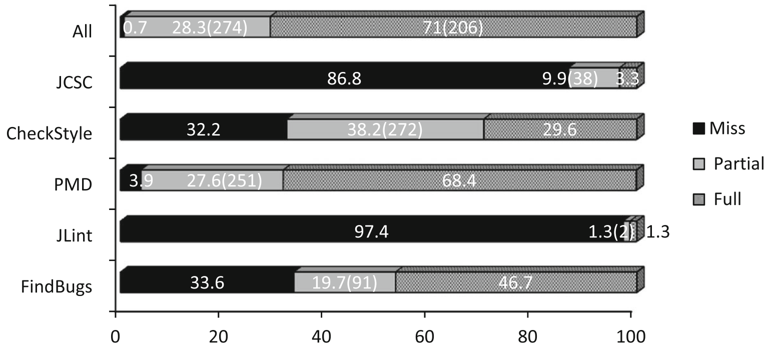
Note that there may be many other issues in a version besides the defects in our dataset in a program version, and a tool may generate many warnings for the version. These partially explain the high average numbers of lines flagged in Table 2. To further understanding these numbers, we present more statistics about the warnings generated



(a) Percentage of Lucene Defects



(b) Percentage of Rhino Defects



(c) Percentage of AspectJ Defects

Fig. 6 Percentages of defects that are missed, partially captured, and fully captured for : **a** Lucene, **b** Rhino, and **c** AspectJ. The numbers in the *parentheses* indicate the numbers of actual faulty lines captured, if any

by the tools for each of the three programs in Table 3. Column "Avg # Warning" provides the average numbers of reported warnings by each tool for each program. Column "Avg # Flagged Line Per Warning" is the average numbers of lines flagged by a warning report. Column "Avg # Unfixed" is the average numbers of lines of code that are flagged but are not the buggy lines fixed in a version. Column "Avg # LOC" is the numbers of lines of code in a particular program (across all buggy versions).

Table 2 Average numbers of lines flagged per defect version by various static bug-finding tools

Tools versus 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
Checkstyle	16,180.6	11,144.65	55,900.13
JCSC	16,682.46	3,573.55	22,010.77
All	126,367.75	52,123.05	220,263

Table 3 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	
	Checkstyle	34,958.75	1	11,144.65	
	JCSC	24,087	1	3,573.55	
	All	71,726.8	2.95	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	
	Checkstyle	30,779.71	1	16,680.25	
	JCSC	29,056.96	1	16,681.82	
	All	100,513.57	1.46	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	
	Checkstyle	93,341.5	1	55,897.93	
	JCSC	198.518,65	1	22,010.49	
	All	326,040.72	1.16	220,260.31	

We notice that the average numbers 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. Heckman and Williams 2011). 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 numbers of the lines flagged by each warning generated by each of the bug-finding tools, and

Table 4 p-values comparing random tools against 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
Checkstyle	Lucene	0.0471	0.0152
	Rhino	0.5085	1
	AspectJ	<0.0001	<0.0001
JCSC	Lucene	0.0152	1
	Rhino	0.3636	1
	AspectJ	0.0006	1

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 numbers 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 4. 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.

5.2 Consider localized defects

Many of the defects that we analyze span more than a few lines of code. Thus, 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 five tools, for Lucene, Rhino, and AspectJ in Fig. 7.
Lucene Fig. 7 shows that the tools together could identify 61.9 % of all localized defects. The ordering of the tools based on their ability to fully capture localized defects remain the same.

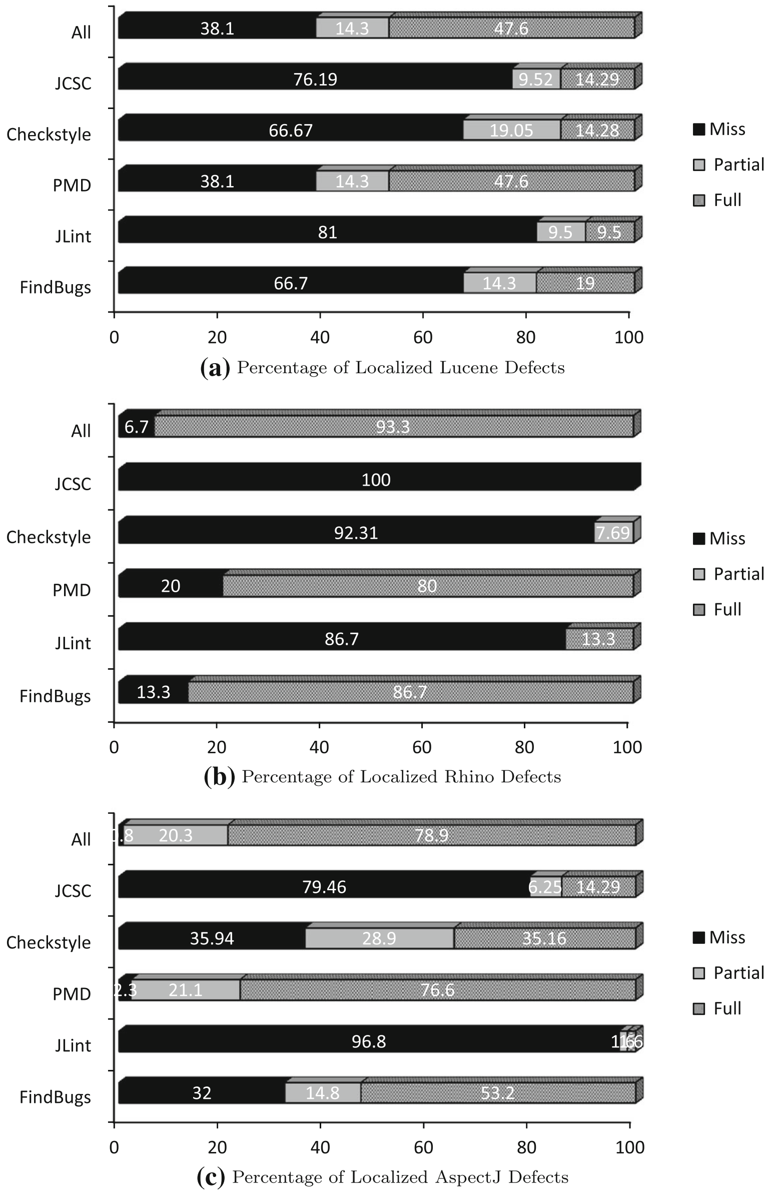


Fig. 7 Percentages of localized defects that are missed, partially captured, and fully captured for : **a** Lucene, **b** Rhino, and **c** AspectJ

Rhino As shown in Fig. 7, all five tools together could fully identify 93.3 % of all localized defects. This is slightly lower than the percentage for all defects (see Fig. 6).

AspectJ Fig. 7 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 Fig. 6).

Table 5 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
Checkstyle	0	3	0	0	0	33
JCSC	0	3	0	0	0	5

5.3 Consider stricter analysis

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. Warning reports that cover this line but do not have reasonable explanation for the warning would not be counted as capturing the fault. 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. 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 5. 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.

Although the majority of field defects can be fully or partially captured by existing bug finding tools, many defects cannot be captured. Overall, we find that PMD could outperform other tools in capturing defects. The effectiveness of the other tools varies across the datasets; for example, Findbugs are more effective in capturing defects in Rhino, while Checktyle is more effective in capturing defects in AspectJ. JLint and JCSC are often the most ineffective in capturing defects in the datasets.

We find that these tools flag many lines of code as potentially buggy. In the worst case, the tools can flag more than half of the lines in a program as buggy. Still, in many cases, existing bug finding tools are statistically significantly better than a tool that randomly flags a certain number of program lines as buggy. Another observation is although many warnings flag correct buggy lines, often the warning messages do not contain sufficient information to help developers debug

6 RQ2: effectiveness of different warnings

We show the effectiveness of various warning families of FindBugs, PMD, JLint, CheckStyle, and JCSC in flagging defects in Figs. 8, 9, 10, 11 and 12 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 8, 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 concatenates strings using + instead of StringBuffer, etc.

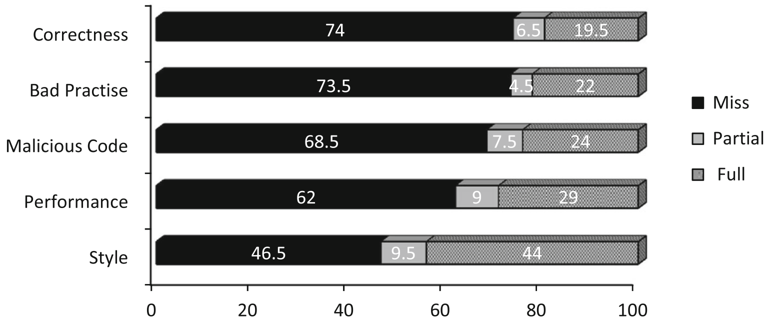


Fig. 8 Percentages of defects that are missed, partially captured, and fully captured for different warning families of FindBugs

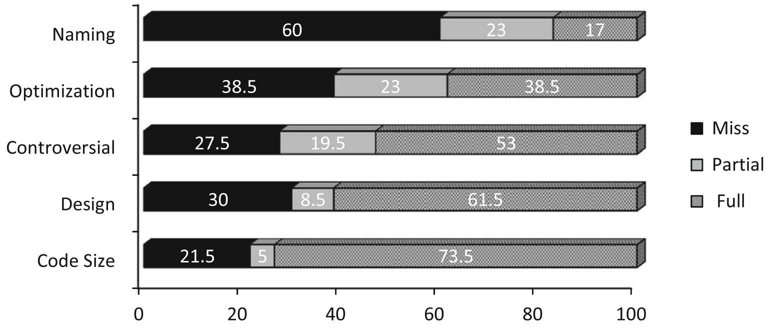


Fig. 9 percentages of defects that are missed, partially captured, and fully captured for different warning families of PMD

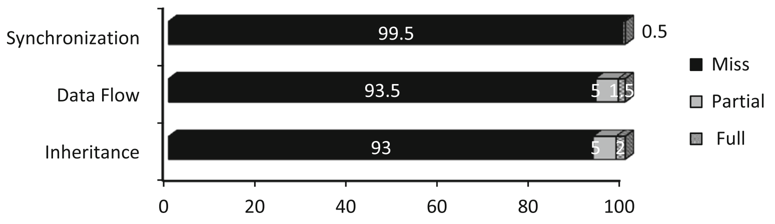


Fig. 10 percentages of defects that are missed, partially captured, and fully captured for different warning families of JLint

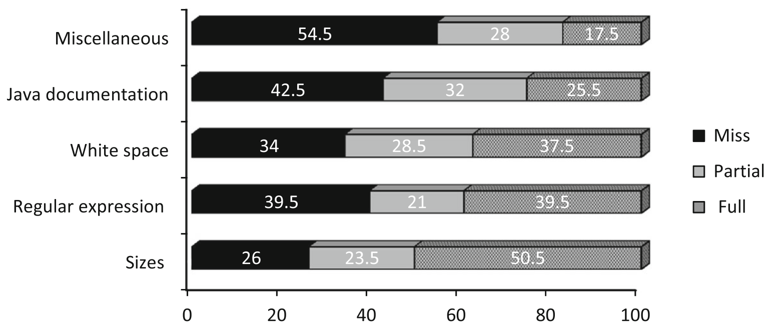


Fig. 11 percentages of defects that are missed, partially captured, and fully captured for different warning families of checkstyle

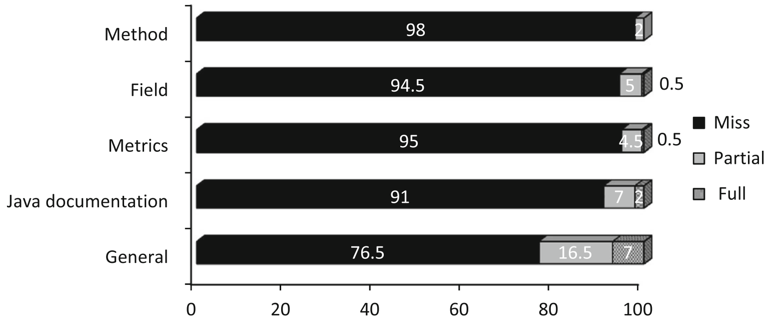


Fig. 12 Percentages of defects that are missed, partially captured, and fully captured for different warning families of JCSC

Correctness related warnings include a field that masks a superclass's field, invocation of toString to an array, etc.

PMD For PMD, as shown in Fig. 9, 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.

JLint For JLint, as shown in Fig. 10, 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.

CheckStyle For CheckStyle, as shown in Fig. 11, we have five most effective categories: Sizes, Regular expression, White space, Java documentation, and Miscellaneous. Sizes warning occurred when a considerable limit of good program size is exceeded. Regular expression warning occurred when a specific standard pattern exist or not exist in the code file. White space warning occurred if there is an incorrect whitespace around generic tokens. Java documentation warning occurred when there is no javadoc or the javadoc does not conform to standard. Miscellaneous warning includes other issues that are not included in the other warning category. Although likely unrelated to the actual cause of the defects, the top warning family is quite effective on catching the defects.

JCSC For JCSC, as shown in Fig. 12, we have five most effective categories: General, Java documentation, Metrics, Field, and Method. General warning includes common issues in coding, such as the length limit of the code, missing default in switch statement, empty catch statement, etc. Java documentation warning includes issues related to the existence of javadoc and its conformance to standard. Metrics warning includes issues related to metrics used for measuring code quality, such as NCSS and NCC. Field warning includes issues related to the field of the class, such as bad modifier order. Method warning includes issues related to the method of the class, such as bad number of arguments of a method. The warnings generated by JCSC are generally not effective and very bad in detecting defects in the three systems.

The types of warnings of FindBugs, PMD, JLint, Checkstyle, and JCSC that are the most effective in (fully or partially) capturing actual defects are Style, Code Size, Inheritance, Sizes, and General, respectively.

7 RQ3: characteristics of missed defects

There are 12 defects that are missed by all five 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 catego-

Table 6 Distribution of missed defect types

Type	Number of defects
Assigment related defects	1
Conditional checking related defects	6
Return value related defects	1
Method related defects	3
Object usage related defects	1

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, "");

Fig. 13 Example of a defect missed by all tools

rize the defects into several categories: method related defects (addition, removal of method calls, changes to parameters passed in to methods), conditional checking related defects (addition, removal, or changes to pre-condition checks), assignment related defects (wrong value being assigned to variables), return value related defects (wrong value being returned) and object usage related defects (missing type cast, etc). The distribution of such defect categories that are missed by all five tools are listed in Table 6.

A sample defect missed by all five tools is shown in Fig. 13, which involves an invocation of a wrong method.

There are only a few defects that are missed by all tools. These defects are related to logical/functionality errors which are difficult to detect without knowledge of the system specifications.

8 RQ4: effectiveness on defects of various severity

Different defects are often labeled with different severity levels. Lucene is tracked using Jira bug tracking systems, while Rhino and AspectJ are tracked using Bugzilla bug tracking systems. Jira has the following predefined severity levels: blocker, critical, major, minor, trivial. Bugzilla has the following predefined severity levels: blocker, critical, major, normal, minor, trivial. In Jira, there is no normal severity level. To standardize the two, we group blocker, critical, and major as *severe* group, and normal, minor, and trivial as *non-severe* group. Table 7 describes the distribution of severe and non-severe defects among the 200 defects that we analyze in this study.

We investigate the number of false negatives for each category of defects. We show the results for severe and non-severe defects in Figs. 14, 15 respectively. We find that PMD captures the most numbers of severe and non-severe defects in most datasets. For almost all the datasets that we analyze, we also find that the tools that could more

Table 7 Distribution of defects of various severity levels

Type	Number of defects
Severe	45
Non-Severe	155

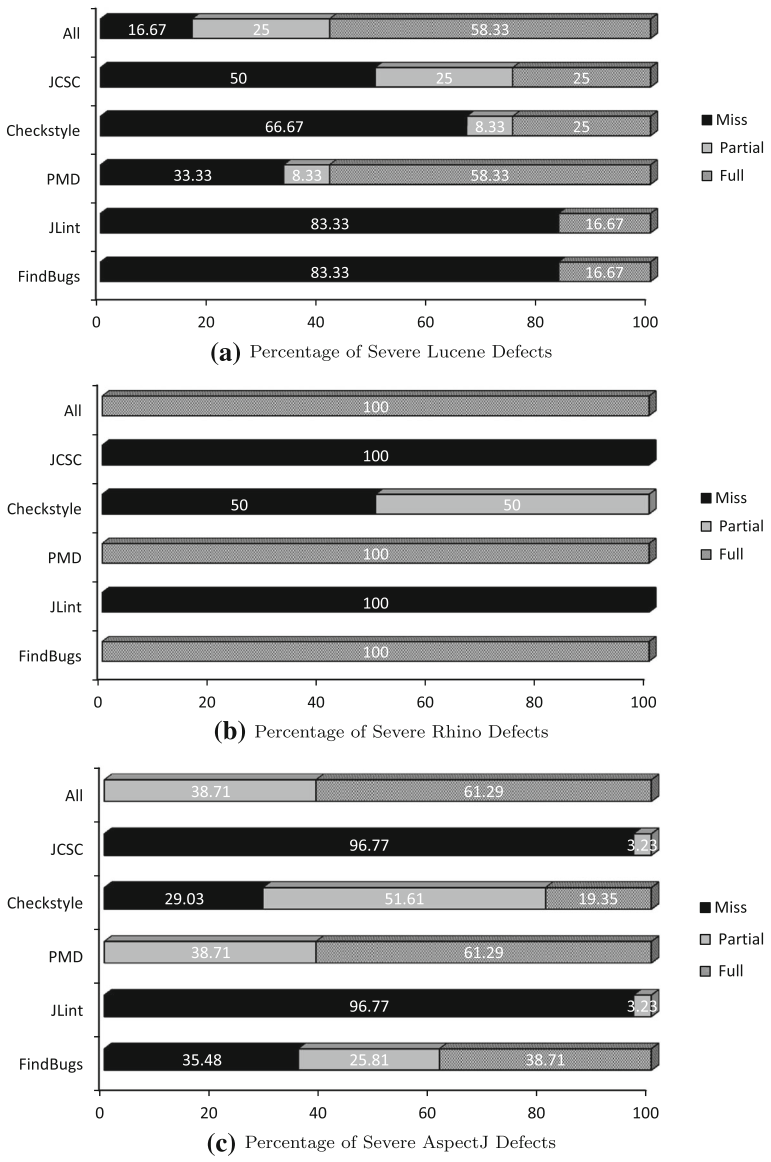


Fig. 14 Percentages of Severe Defects that are Missed, Partially Captured, and Fully Captured for : **a** Lucene, **b** Rhino, and **c** AspectJ

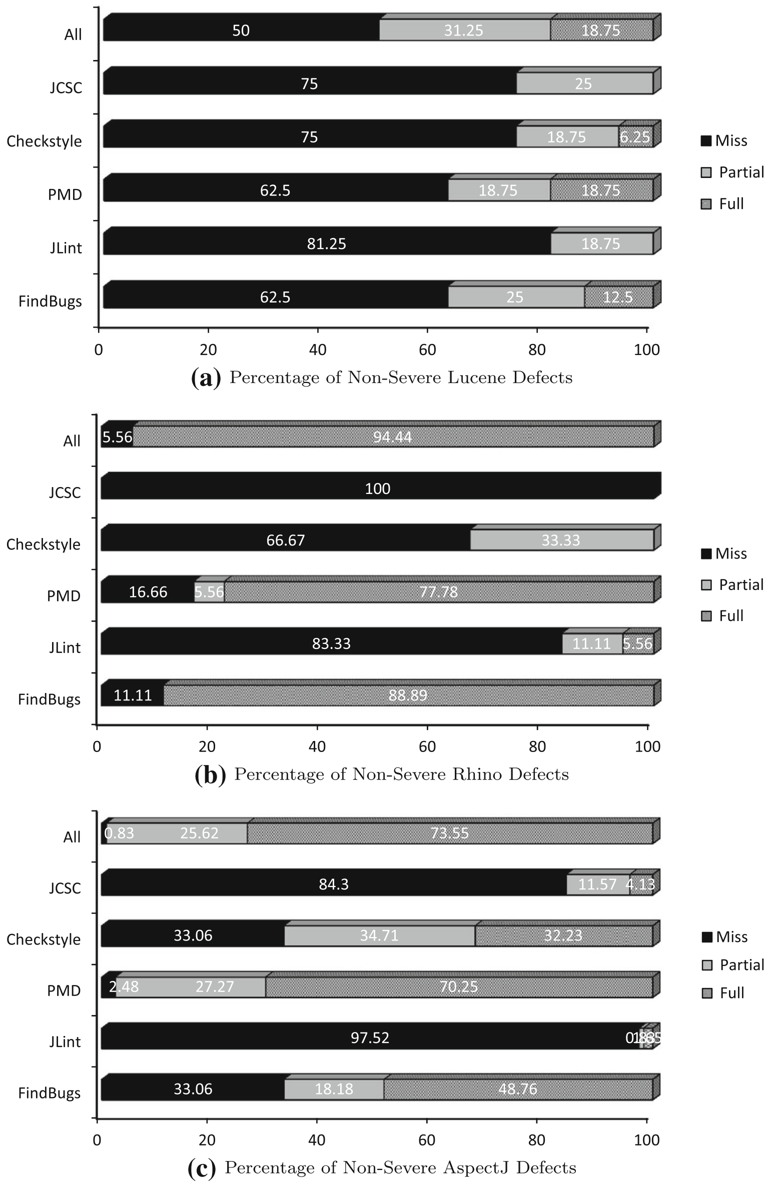


Fig. 15 Percentages of non-severe defects that are missed, partially captured, and fully captured for : **a** Lucene, **b** Rhino, and **c** AspectJ

effectively capture severe defects in a dataset, could also more effectively capture non-severe defects in the same dataset. For Lucene, PMD and JCSC are the top two tools that could more effectively capture severe defects. For non-severe defects in Lucene, PMD and FindBugs are the top two tools that effectively capture non-severe bugs. For Rhino, PMD and FindBugs are the top two tools that could more effectively capture severe defects without missing any defects and capture non-severe defects. For AspectJ, PMD and CheckStyle are the top two tools that could more effectively capture both severe and non-severe defects. FindBugs could also effectively capture non-severe defects in AspectJ.

Based on the percentage of bugs that could be partially or fully captured by all the five tools together, the tools could capture severe defects more effectively than non-severe defects. For Lucene, only 16.67 % of severe defects and 50 % of non-severe defects could not be captured with all the five tools. While for Rhino and AspectJ, all severe defects could be captured using all the five tools, and only 5.56 % of non-severe defects of Rhino and 0.83 % of non-severe defects of AspectJ could not be captured.

The bug finding tools could capture severe defects more effectively as compared to non-severe defects.

9 RQ5: effectiveness on defects of various difficulties

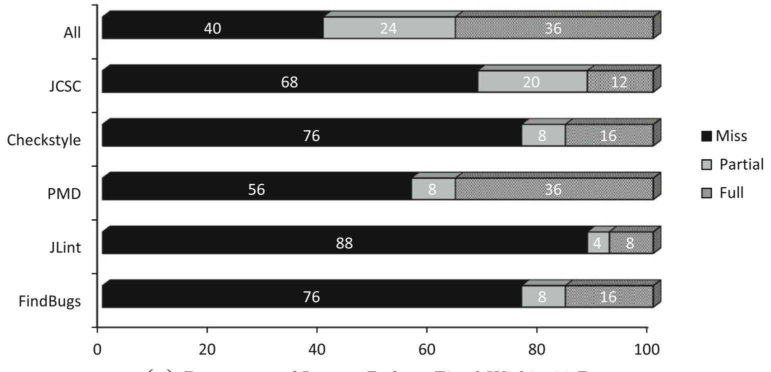
Some defects are more difficult to fix than others. Difficulty could be measured in various ways: time needed to fix a defect, number of bug resolvers involved in the bug fixing process, number of lines of code churned to fix a bug, and so on. We investigate these different dimensions of difficulty and analyze the number of false negatives on defects of various difficulties in the following paragraphs.

9.1 Consider time to fix a defect

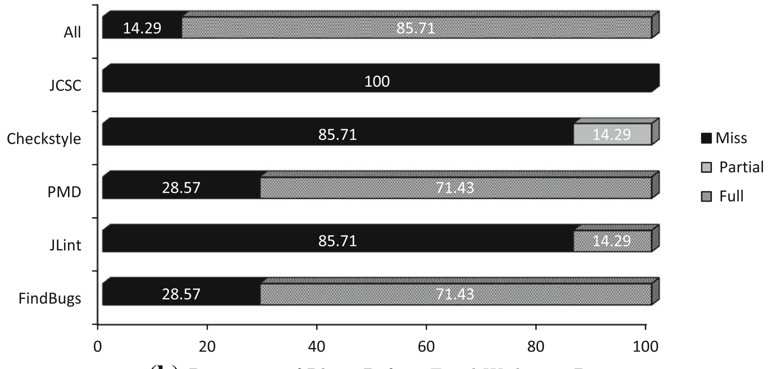
First, we measure difficulty in terms of the time it takes to fix a defect. Time to fix defect has been studied in a number of previous studies (Kim and EJW 2006; Weiss et al. 2007; Hosseini et al. 2012). Following Thung et al. (2012), we divide time to fix a defect into 2 classes: ≤30 days, and >30 days.

We show the results for ≤30 days and and >30 days defects in Figs. 16, 17 respectively. Generally, we find that >30 days defects can be better captured by the tools than ≤30 days defects. By using the five tools together, for defects that are fixed within 30 days, we could partially or fully capture all defects of AspectJ, but miss a substantial number of AspectJ and Lucene defects. For defects that are fixed in more than 30 days, all defects of Lucene and Rhino and almost all of AspectJ defects can be captured.

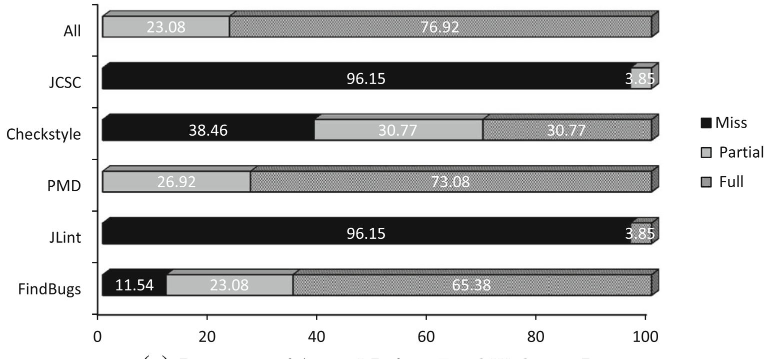
Also we find that among all the five tools, PMD could more effectively captures both defects that are fixed within 30 days and those fixed in more than 30 days. The top two most effective tools in capturing defects fixed within 30 days are similar with those capturing defects fixed in more than 30 days. For Lucene's defects, PMD and JCSC are the most effective tools to identify both defects fixed within 30 days and those fixed in more than 30 days; also, PMD could capture all of the defects fixed in more than 30 days, while JCSC, Findbugs, JLint, and CheckStyles could not capture 33.33 % of the >30 days defects. For Rhino's defects, PMD and FindBugs are the most effective tools to identify both defects fixed within 30 days and those fixed in



(a) Percentage of Lucene Defects Fixed Within 30 Days



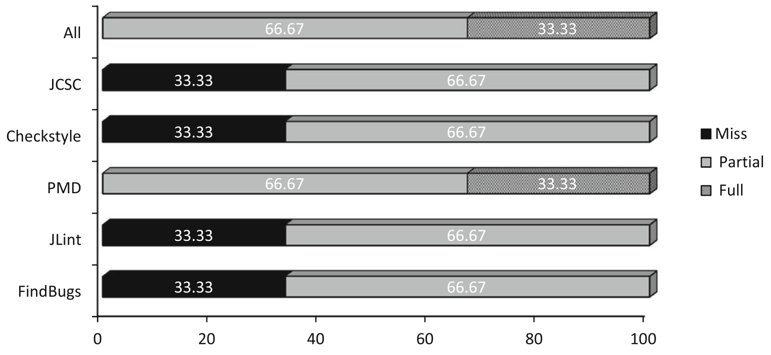
(b) Percentage of Rhino Defects Fixed Within 30 Days



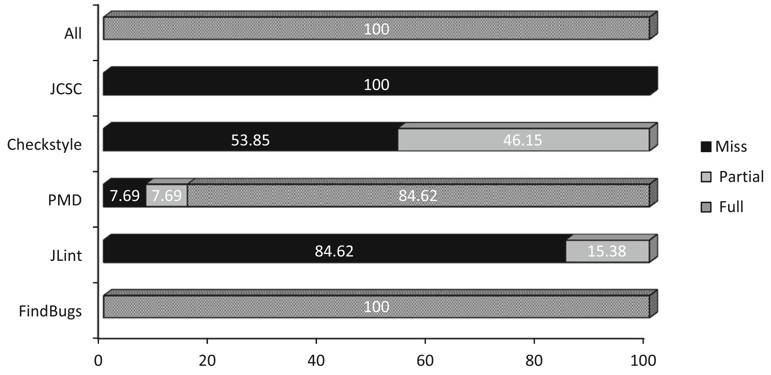
(c) Percentage of AspectJ Defects Fixed Within 30 Days

Fig. 16 percentages of defects fixed within 30 days that are missed, partially captured, and fully captured for : **a** Lucene, **b** Rhino, and **c** AspectJ

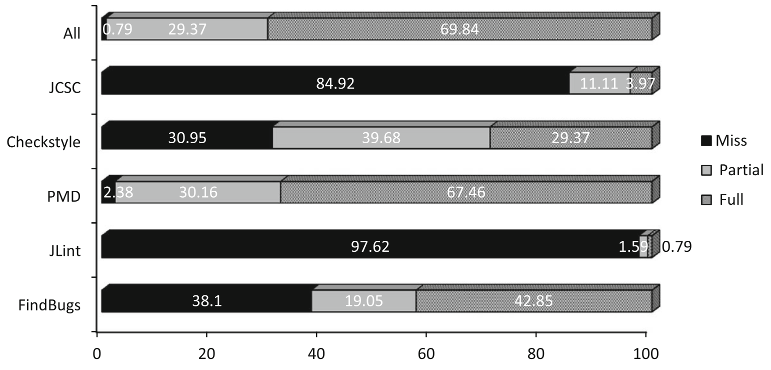
more than 30 days; also, FindBugs could capture all Rhino's defects that are fixed in more than 30 days. For AspectJ's defects, PMD, FindBugs, and CheckStyles are the most effective tools to identify both defects fixed within 30 days and those fixed in more than 30 days.



(a) Percentage of Lucene Defects Fixed Above 30 Days



(b) Percentage of Rhino Defects Fixed Above 30 Days



(c) Percentage of AspectJ Defects Fixed Above 30 Days

Fig. 17 Percentages of defects fixed above 30 days that are missed, partially captured, and fully captured for : **a** Lucene, **b** Rhino, and **c** AspectJ

Summary Overall, we find that the five tools used together are more effective in capturing defects that are fixed in more than 30 days than those that are fixed within 30 days. This shows that the tools are beneficial as they can capture more difficult bugs. Also, among all the five tools, PMD could more effectively capture both defects fixed within 30 days and those fixed in more than 30 days.

9.2 Consider number of resolvers for a defect

Next, we measure difficulty in terms of the number of bug resolvers. Number of bug resolvers has been studied in a number of previous studies (e.g. Wu et al. 2011b; Xie et al. 2012; Xia et al. 2013). Based on the number of bug resolvers, we divide bugs into 2 classes: $\leq$2 people, and $>$2 people. A simple bug typically would have only 2 bug resolvers: a triager who assigns the bug report to a fixer, and the fixer who fixes the bug.

We show the results for $\leq$2 and $>$2 resolvers defects in Figs. 18, 19 respectively. When using the 5 tools together, we find that they are almost equally effective in capturing $\leq$2 and $>$2 resolvers defects for Lucene and AspectJ datasets. For the Lucene dataset, by using all the 5 tools, we miss the same percentage of $\leq$2 and $>$2 resolvers defects. For the AspectJ dataset, by using all the 5 tools, we miss only a small percentage of defects. However, for the Rhino dataset, by using all the 5 tools together, we can more effectively capture $>$ 2 resolvers defects than $\leq$2 resolvers defects.

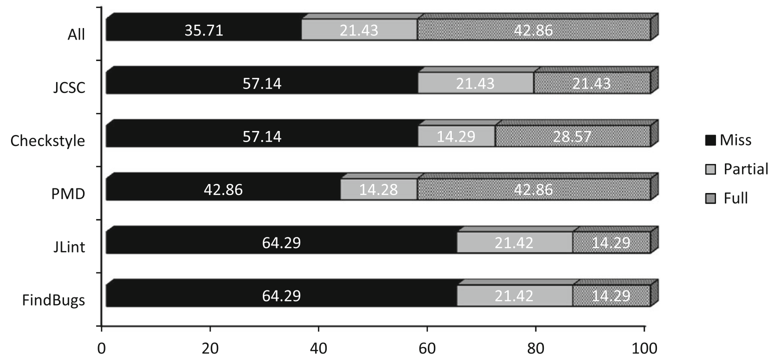
Also, we find that among all the five tools, PMD could more effectively capture both $\leq$2 and $>$2 resolvers defects. The top two most effective tools in capturing $\leq$2 resolvers defects are similar with those capturing $>$2 resolvers defects. For Lucene's defects, PMD and JCSC are the most effective tools to capture for both $\leq$2 and $>$2 resolvers defects. For Rhino's defects, PMD and FindBugs are the most effective tools to capture both $\leq$2 and $>$2 resolvers defects. For AspectJ's defects, PMD, FindBugs, and CheckStyles are the most effective tools to capture both $\leq$2 and $>$2 resolvers defects.

Summary Overall, we find that by using the five tools together, we can capture $>$2 resolvers defects more effectively than $\leq$2 resolvers defects (at least on Rhino dataset). This shows that the tools are beneficial as they can capture more difficult bugs. Also among all the five tools, PMD could more effectively capture for both $\leq$2 and $>$2 resolvers defects.

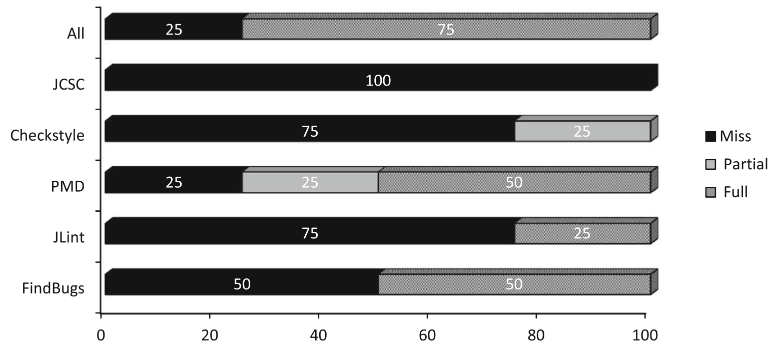
9.3 Consider number of lines of code churned for a defect

We also measure difficulty in terms of the number of lines of code churned. Number of lines of code churned has been investigated in a number of previous studies, e.g., (Nagappan and Ball 2005; Giger et al. 2011). Based on the number of lines of code churned, we divide the bugs into 2 classes: $\leq$10 lines churned, and $>$10 lines churned.

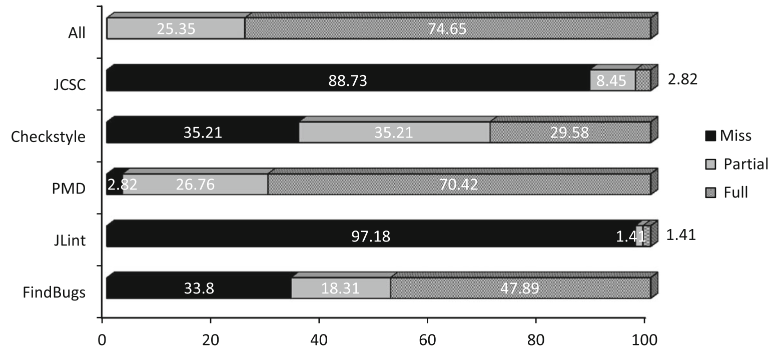
We show the results for $\leq$10 and $>$10 lines churned defects in Figs. 20, 21. The performance of the 5 tools (used together) in capturing $\leq$10 lines churned defects is similar to the performance in capturing $>$10 lines churned defects. For Lucene, using the 5 tools together, we can capture $\leq$10 lines churned defects better than $>$10 lines churned defects. However, for Rhino, using the 5 tools together, we can capture $>$10 lines churned defects better than $\leq$10 lines churned defects. For AspectJ, using the 5 tools together, we can capture $>$10 lines churned defects and $\leq$10 lines churned defects equally well.



(a) Percentage of Lucene Defects Fixed by More than 2 Resolvers



(b) Percentage of Rhino Defects Fixed by More than 2 Resolvers

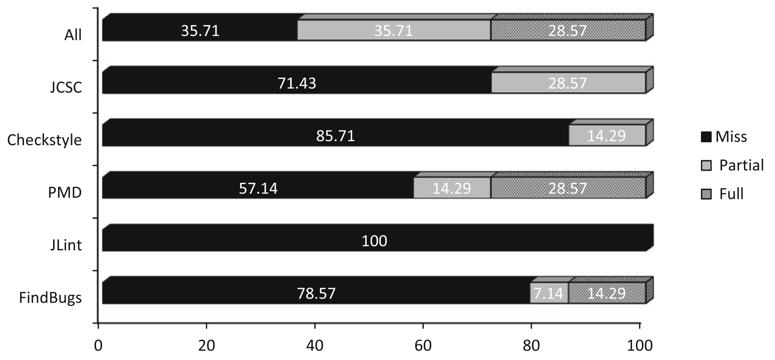


(c) Percentage of AspectJ Defects Fixed by More than 2 Resolvers

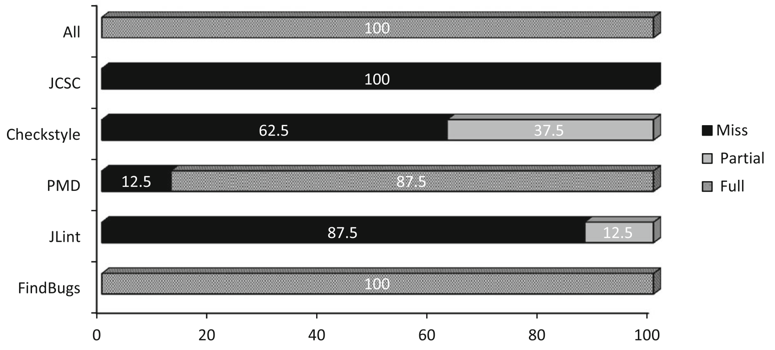
Fig. 18 Percentages of defects fixed by more than 2 resolvers that are missed, partially captured, and fully captured for : **a** Lucene, **b** Rhino, and **c** AspectJ

We find that among all the five tools, PMD could more effectively capture both ≤ 10 and >10 lines churned defects. The top two most effective tools in capturing ≤ 10 lines churned defects however may vary with those capturing >10 lines churned defects.

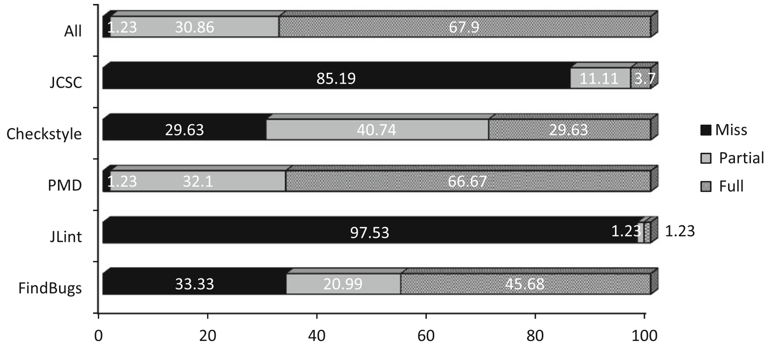
For Lucene's defects, PMD and CheckStyles are the most effective tools to capture ≤ 10 lines churned defects, while PMD and JCSC are the most effective tools to identify >10 lines churned defects. For Rhino's defects, PMD and FindBugs are the



(a) Percentage of Lucene Defects Fixed by at Most 2 Resolvers



(b) Percentage of Rhino Defects Fixed by at Most 2 Resolvers

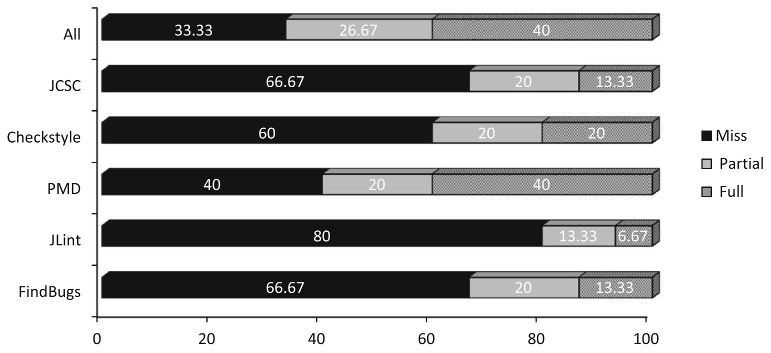


(c) Percentage of AspectJ Defects Fixed by at Most 2 Resolvers

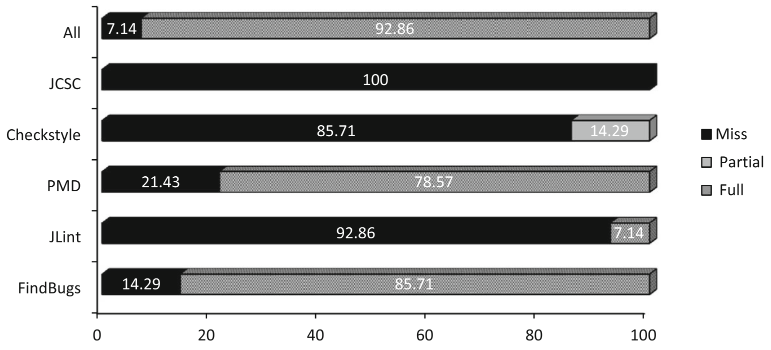
Fig. 19 Percentages of defects fixed by at most 2 resolvers that are missed, partially captured, and fully captured for : **a** Lucene, **b** Rhino, and **c** AspectJ

most effective tools to capture both $\leq$10 and $>$10 lines churned defects. For AspectJ's defects, PMD, FindBugs, and CheckStyles are the most effective tools to identify for both $\leq$10 and $>$10 lines churned defects.

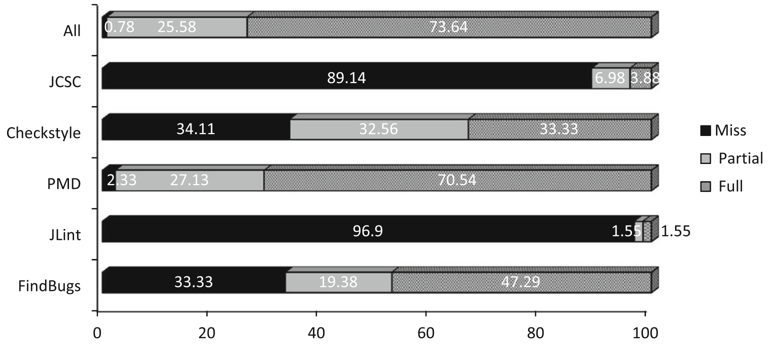
Summary Overall, we find that the performance of the 5 tools (used together) in capturing $\leq$10 lines churned defects are similar to that in capturing $>$10 lines churned



(a) Percentage of Lucene Defects Fixed by Churning at Most 10 Lines



(b) Percentage of Rhino Defects Fixed by Churning at Most 10 Lines

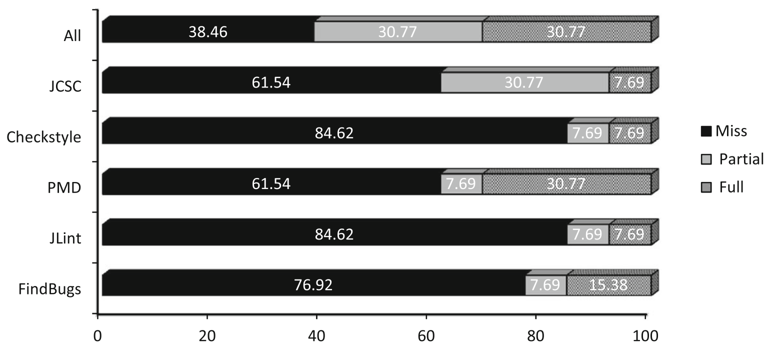


(c) Percentage of AspectJ Defects Fixed Churning at Most 10 Lines

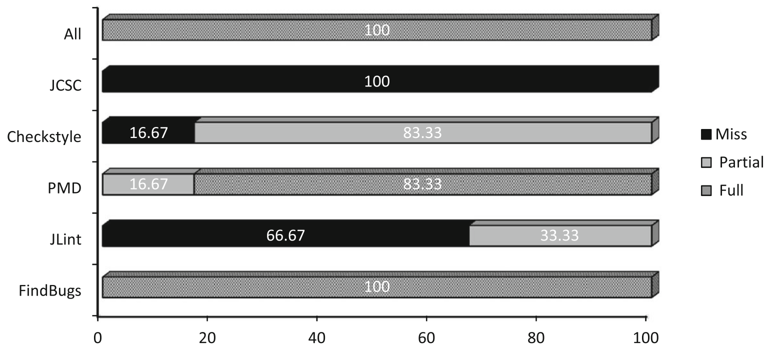
Fig. 20 percentages of defects fixed by churning at most 10 lines that are missed, partially captured, and fully captured for : **a** Lucene, **b** Rhino, and **c** AspectJ

defects. Also among the five tools, PMD could more effectively identify ≤ 10 and >10 lines churned defects.

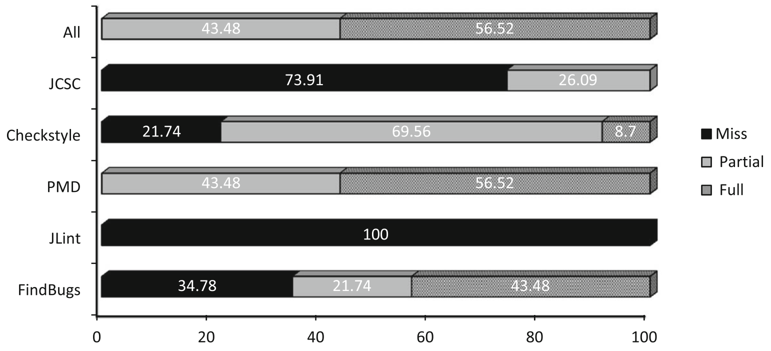
> *The tools are generally more effective in finding defects that require more time to be fixed and more people to be involved in the bug fixing process.*



(a) Percentage of Lucene Defects Fixed by Churning More than 10 Lines



(b) Percentage of Rhino Defects Fixed by Churning More than 10 Lines



(c) Percentage of AspectJ Defects Fixed Churning More than 10 Lines

Fig. 21 percentages of defects fixed by churning more than 10 lines that are missed, partially captured, and fully captured for : **a** Lucene, **b** Rhino, and **c** AspectJ

10 RQ6: effectiveness on defects of various types

We consider a rough categorization of defects based on the type of the program elements that are affected by a defect. We categorize defects into the following families (or types): assignment, conditional, return, method invocation, and object usage. A

Table 8 Distribution of defect types

Type	Number of Defects
Assignment related defects	27
Conditional related defects	127
Return related defects	11
Method invocation related defects	28
Object usage related defects	7

defect belongs to a family if the statement responsible for the defect contains a corresponding program type. For example, if a defect affects an "if" statement it belongs to the conditional family. Since a defect can span multiple program statements of different types, it can belong to multiple families, for such cases we analyze the bug fix and choose one dominant family. Table 8 describes the distribution of defect types among the 200 defects that we analyze in this study.

We investigate the number of false negatives for each category of defects. We show the results for assignment related defects in Fig. 22. We find that using the five tools together we could effectively capture (partially or fully) assignment related defects in all datasets: we could capture all assignment related defects in Rhino and AspectsJ, and only 16.67 % of Lucene's assignment related defects could not be captured by the tools. We also find that PMD is the most effective tool in capturing assignments related defects, while JLint and JCSC are the most ineffective. PMD and Findbugs could effectively capture Rhino's assignment related defects without missing any defects, and only 11.11 % of AspectJ's assignment related defects could not be captured by these tools. Thus, besides PMD, FindBugs is also effective in capturing assignment related defects.

We show the results for conditional related defects in Fig. 23. We find that using the five tools together, we could capture all conditional related defects in Rhino, and only a small percentage of AspectJ's conditional related defects could not be captured by the tools; while for Lucene, a substantial proportion of conditional related defects could not be captured. We also find that PMD is the most effective tool in capturing conditional related defects, while JLint is the most ineffective in capturing the defects. PMD and Findbugs could effectively capture Rhino's conditional related defects with less than 16 % missed defects, and only around 30 % of AspectJ's conditional related defects could not be captured by these tools. Thus, aside from PMD, FindBugs is also effective in capturing conditional related defects.

We show the results for return related defects in Fig. 24. We find that using the five tools together is effective enough to capture all return related defects in AspectJ and Lucene, however they are less effective in capturing return related defects in Rhino. We also find that PMD is the most effective tool in capturing return related defects. For Lucene, all tools could capture (partially or fully) all the return related defects. For Rhino, half of return related defects could not be captured by PMD, FindBugs, and JLint, while the other tools could not capture all the defects. For AspectJ, PMD could effectively capture all return related defects without any missed defects,

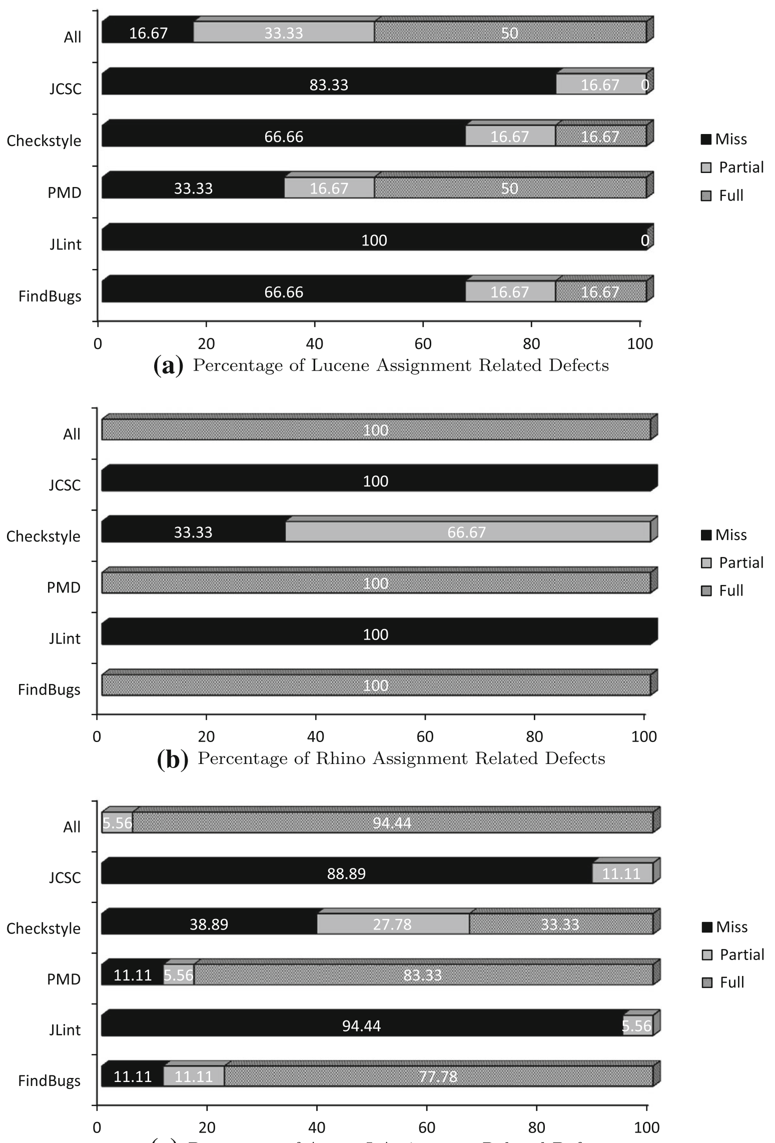
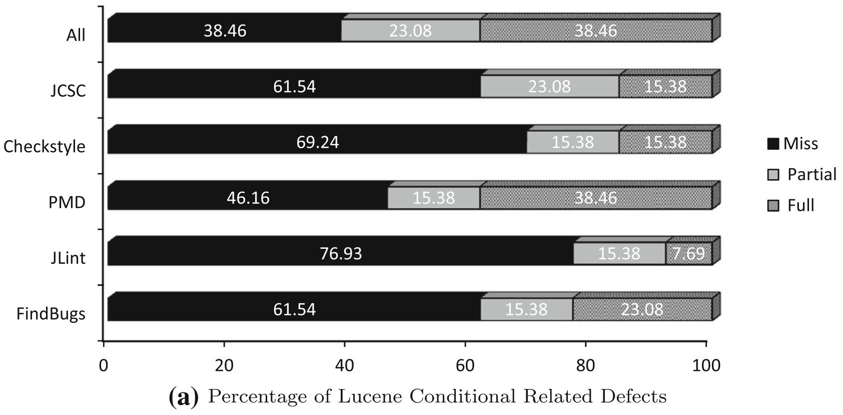
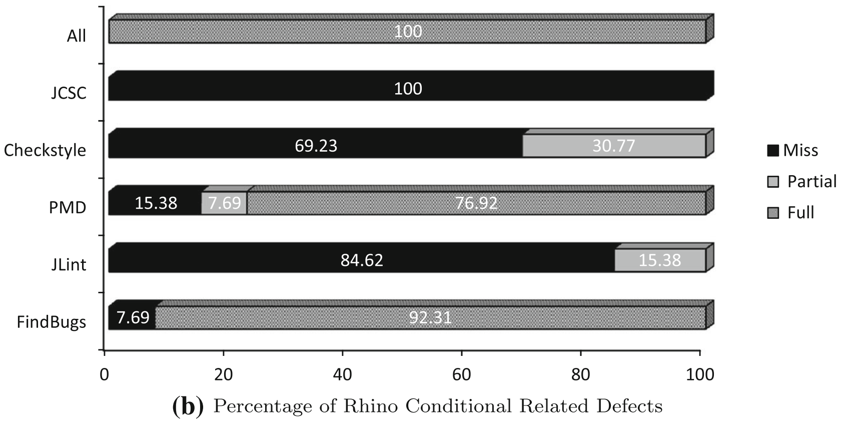


Fig. 22 Percentages of Assignment Related Defects that are Missed, Partially Captured, and Fully Captured for : **a** Lucene, **b** Rhino, and **c** AspectJ

and CheckStyle only missed around a third of the defects. The other tools could not effectively capture AspectJ's return related defects as they miss the majority of the defects.

We show the results for method invocation related defects in Fig. 25. We find that using the five tools together is effective enough to capture all method invocation defects of Rhino and AspectJ, but they are less effective in capturing method invocation related





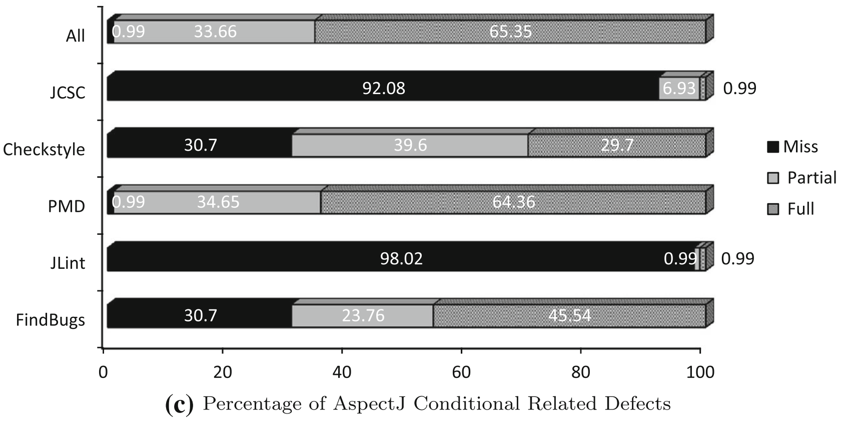
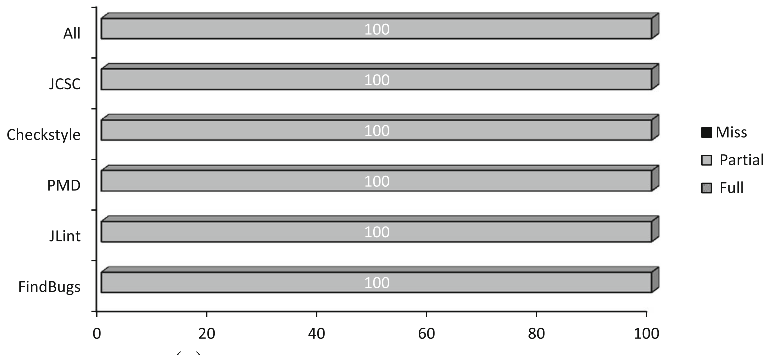
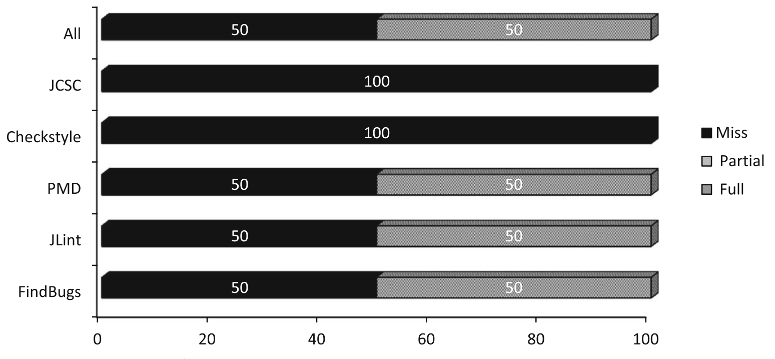


Fig. 23 Percentages of conditional related defects that are missed, partially captured, and fully captured for : **a** Lucene, **b** Rhino, and **c** AspectJ

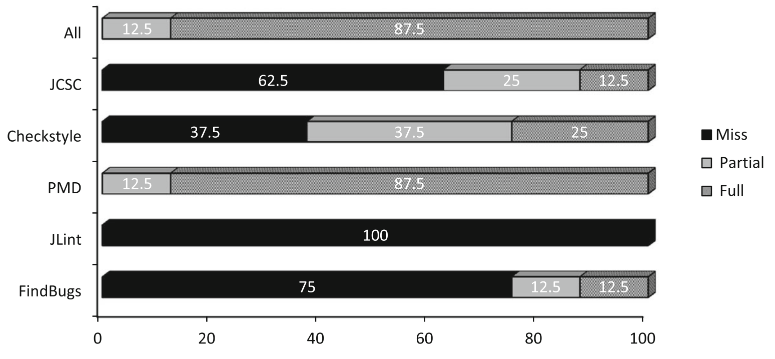
defects in Lucene. We also find that PMD is the most effective tool in capturing method invocation related defects. PMD could effectively capture all method invocation related defects in Rhino and AspectJ, but it misses two thirds of Lucene's method invocation



(a) Percentage of Lucene Return Related Defects



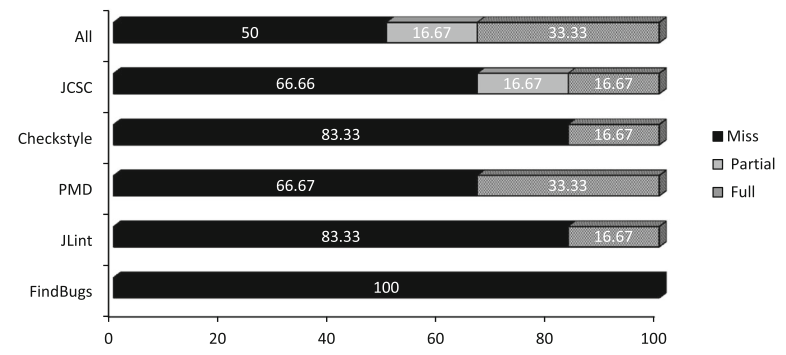
(b) Percentage of Rhino Return Related Defects



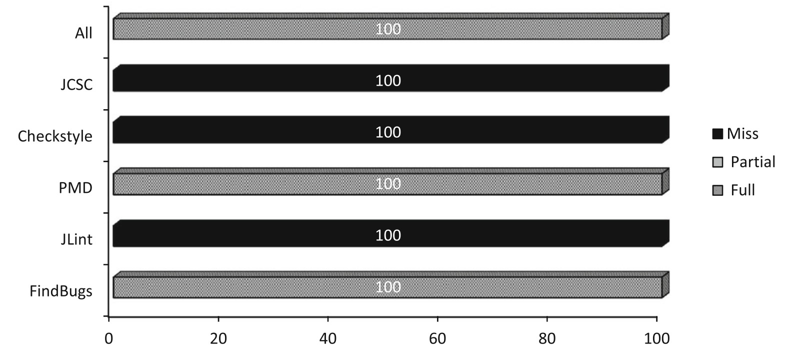
(c) Percentage of AspectJ Return Related Defects

Fig. 24 Percentages of return related defects that are missed, partially captured, and fully captured for : **a** Lucene, **b** Rhino, and **c** AspectJ

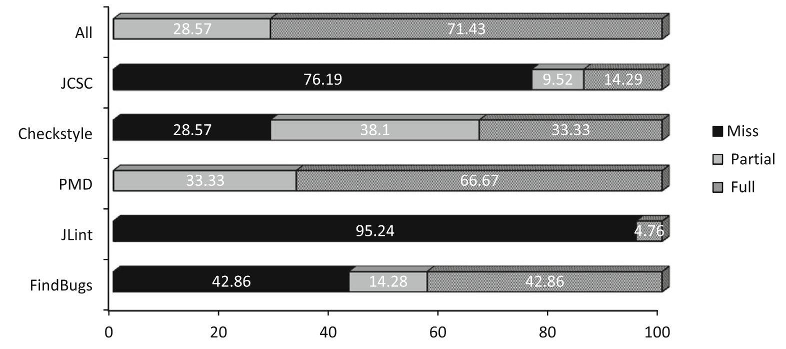
related defects. The other 4 tools could not capture more than 70 % of Lucene's method invocation related defects. Other than PMD, Findbugs could also effectively capture all Rhino's method invocation related defects without any missed defects, while the other 3 tools could not capture all Rhino's method invocation related defects. For AspectJ, CheckStyle could also effectively capture the defects with only around a quarter of



(a) Percentage of Lucene Method Invocation Related Defects



(b) Percentage of Rhino Method Invocation Related Defects

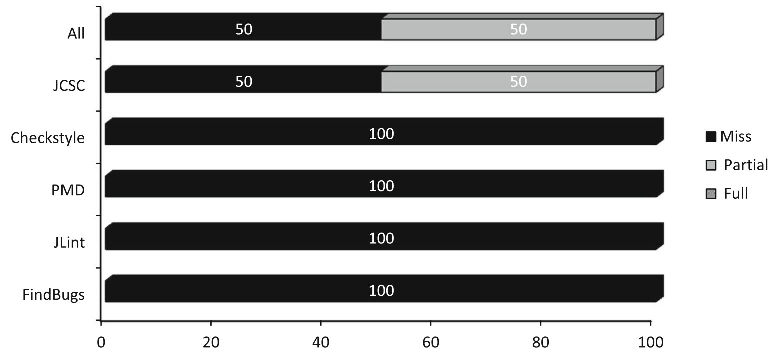


(c) Percentage of AspectJ Method Invocation Related Defects

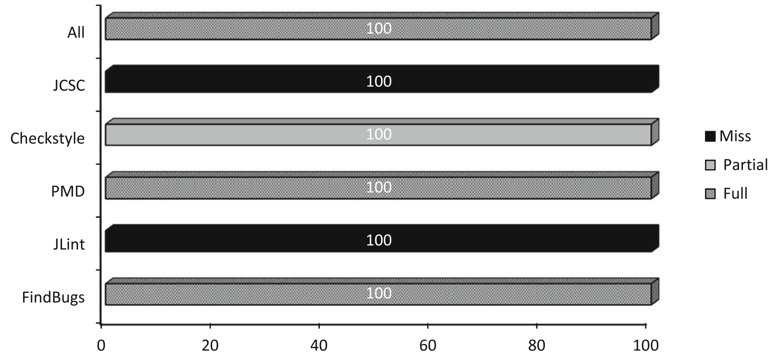
Fig. 25 Percentages of method invocation related defects that are missed, partially captured, and fully captured for : **a** Lucene, **b** Rhino, and **c** AspectJ

defects that are missed, while the other 3 tools could not capture more than 40 % of AspectJ's method invocation related defects.

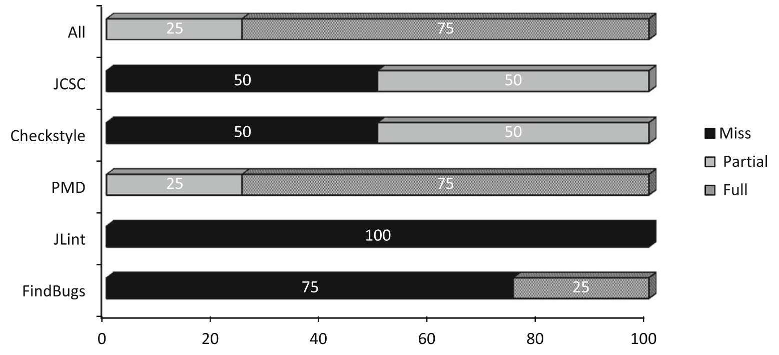
We show the results for object usage related defects in Fig. 26. We find that using the five tools together, we can capture all object usage related defects of Rhino and AspectJ, but they are less effective in capturing object usage related defects of Lucene.



(a) Percentage of Lucene Object Usage Related Defects



(b) Percentage of Rhino Object Usage Related Defects



(c) Percentage of AspectJ Object Usage Related Defects

Fig. 26 Percentages of object usage related defects that are missed, partially captured, and fully captured for : **a** Lucene, **b** Rhino, and **c** AspectJ

We also find that generally PMD is the most effective tool in capturing object usage related defects. PMD could effectively capture all object usage related defects in Rhino and AspectJ, but it misses all Lucene's object usage related defects. Only JCSC could capture half of Lucene's object usage related defects, while the other tools could not capture these defects at all. Other than PMD, Findbugs and CheckStyle could also effectively capture all Rhino's method invocation defects without any missed defects, while the other tools could not capture all Rhino's method invocation defects. For AspectJ, the other tools except PMD could not effectively capture object usage related defects in this dataset—they could not capture 50 % or more of the defects.

Based on the percentage of missed defects, averaging across the defect families, the 5 bug-finding tools (used together) could more effectively capture assignment defects, followed by conditional defects. They are less effective in capturing return, method invocation, and object usage related defects.

11 Threats to validity

Threats to internal validity may include experimental biases. We manually extract faulty lines from changes that fix them; this manual process might be error-prone. We also manually categorize defects into their families; this manual process might also be error prone. To reduce this threat, we have checked and refined the results. We also exclude defects that could not be unambiguously localized to the responsible faulty lines of code. In this work, we omit complex ambiguous bugs. This might bias the sample towards simpler types of bugs. However, we need to remove these complex ambiguous bugs from our dataset since we are not able to infer the lines of code that are affected by these bugs. Also, we exclude some versions of our subject programs that cannot be compiled. There might also be implementation errors in the various scripts and code used to collect and evaluate defects.

Threats to external validity are related to the generalizability of our findings. In this work, we only analyze five 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.

12 Related work

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

12.1 Bug finding

There are many bug-finding tools proposed in the literature (Nielson et al. 2005; Necula et al. 2005; Holzmann et al. 2000; Sen et al. 2005; Godefroid et al. 2005; Cadar et al. 2008; Śliwerski et al. 2005). A short survey of these tools has been provided in Sect. 2. In this paper, we focus on five static bug-finding tools for Java, including FindBugs, PMD, JLint, Checkstyle and JCSC. 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.

12.2 Warning prioritization

Many studies deal with the many false positives produced by various bug-finding tools. Kremenek and Engler (2003) use *z-ranking* to prioritize warnings. Kim and Ernst (2007) prioritize warning categories using historical data. Ruthruff et al. (2008) predict actionable static analysis warnings by proposing a logistic regression model that differentiates false positives from actionable warnings. Liang et al. (2010) propose a technique to construct a training set for better prioritization of static analysis warnings. A comprehensive survey of static analysis warnings prioritization has been written by Heckman and Williams (2011). There are large scale studies on the false positives produced by FindBugs (Ayewah and Pugh 2010; Ayewah et al. 2008).

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.

12.3 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 and Murphy (2004) propose a method that uses text categorization to triage bug reports. Anvik et al. (2006) use machine learning techniques to triage bug reports. Hooimeijer and Weimer (2007) construct a descriptive model to measure the quality of bug reports. Park et al. (2011) propose *CosTriage* that further takes the cost associated with bug reporting and fixing into consideration. Sun et al. (2010) 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.

12.4 Empirical studies on defects

Many studies investigate the nature of defects. Pan and Kim (2009) investigate different bug fix patterns for various software systems. They highlight patterns such as method calls with different actual parameter values, change of assignment expressions, etc. Chou et al. (2001) perform an empirical study of operating system errors.

Closest to our work is the study by Rutar et al. (2004) on the comparison of bug-finding tools in Java. They compare the number of warnings generated by various bug-finding tools. 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.

13 Conclusion and future work

Defects can harm software vendors and users. A number of static bug-finding tools have been developed to catch defects. In this work, we empirically study the effectiveness of state-of-the-art static bug-finding tools in preventing real-life defects. We investigate five bug-finding tools, FindBugs, JLint, PMD, CheckStyle, and JCSC on three programs, Lucene, Rhino, and AspectJ. We analyze 200 fixed real defects and extract faulty lines of code responsible for these defects from their treatments.

We find that most defects in the programs could be *partially or fully captured* by combining reports generated by all bug-finding tools. However, a substantial proportion of defects (e.g., about 36 % for Lucene) are still *missed*. We find that FindBugs and PMD are the best among the bug-finding tools in preventing false negatives. 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 bug-finding tools—these bugs involve logical or functionality errors that are difficult to be detected without any specification of the system. We also find that the bug-finding tools are more effective in capturing severe than non-severe defects. Also, based on various measures of difficulty, the bug-finding tools are more effective in capturing more difficult bugs. Furthermore, the bug-finding tools could perform differently for different bug types: they could be more effectively in capturing assignment and conditional defects than return, method invocation, and object usage related defects.

As future work, we would like to reduce the threats to external validity by experimenting with even more bugs from more software systems. Based on the characteristics of the missed bugs, we also plan to build a new tool that could help to reduce false negatives further.

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

References

Anvik, J., Hiew, L., Murphy, G.C.: Who should fix this bug? In: Proceedings of the 28th International Conference on Software Engineering, pp. 361–370 (2006)

Artho, C.: Jlint—find bugs in java programs (2006). http://jlint.sourceforge.net/. Accessed 15 Aug 2013

Ayewah, N., Pugh, W., Morgenthaler, J.D., Penix, J., Zhou, Y.: Evaluating static analysis defect warnings on production software. In: Proceedings of the 7th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering, pp. 1–8 (2007)

Ayewah, N., Pugh, W.: The google findbugs fixit. In: Proceedings of the 19th International Symposium on Software Testing and Analysis, pp. 241–252 (2010)
Ayewah, N., Hovemeyer, D., Morgenthaler, J.D., Penix, J., Pugh, W.: Using static analysis to find bugs. IEEE Softw. **25**(5), 22–29 (2008)
Ball, T., Levin, V., Rajamani, S.K.: A decade of software model checking with slam. Commun. ACM **54**(7), 68–76 (2011)
Beyer, D., Henzinger, T.A., Jhala, R., Majumdar, R.: The software model checker blast. STTT **9**(5–6), 505–525 (2007)
Brun, Y., Ernst, M.D.: Finding latent code errors via machine learning over program executions. In: Proceedings of the 26th International Conference on Software Engineering, pp. 480–490 (2004)
Burn, O.: Checkstyle (2007). http://checkstyle.sourceforge.net/. Accessed 15 Aug 2013
Cadar, C., Ganesh, V., Pawlowski, P.M., Dill, D.L., Engler, D.R.: EXE: automatically generating inputs of death. ACM Trans. Inf. Syst. Secur. **12**(2), 10 (2008)
Chou, A., Yang, J., Chelf, B., Hallem, S., Engler, D.R.: An empirical study of operating system errors. In: Symposium on Operating Systems Principles (SOSP), pp. 73–88 (2001)
Copeland, T.: PMD Applied. Centennial Books, Kewaskum (2005)
Corbett, J.C., Dwyer, M.B., Hatcliff, J., Laubach, S., Pasareanu, C.S., Robby, Zheng, H.: Bandera: extracting finite-state models from java source code. In: Proceedings of the nternational Conference on Software Engineering ICSE, pp. 439–448 (2000)
Cousot, P., Cousot, R.: An abstract interpretation framework for termination. In: POPL, pp. 245–258 (2012)
Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Rival, X.: Why does astrée scale up? Formal Methods Syst. Design **35**(3), 229–264 (2009)
Cubranic, D., Murphy, G.C.: Automatic bug triage using text categorization. In: Proceedings of the Sixteenth International Conference on Software Engineering and Knowledge Engineering, pp. 92–97 (2004)
Dallmeier, V., Zimmermann, T.: Extraction of bug localization benchmarks from history. In: Automated Software Engineering (ASE), pp. 433–436, 2007
Gabel, M., Su, Z.: Online inference and enforcement of temporal properties. In: Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering (ICSE), pp. 15–24 (2010)
Giger, E., Pinzger, M., Gall, H.: Comparing fine-grained source code changes and code churn for bug prediction. In: Proceedings of the 8th Working Conference on Mining Software Repositories, MSR, pp. 83–92 (2011)
Godefroid, P., Klarlund, N., Sen, K.: Dart: directed automated random testing. In: Programming Language Design and Implementation PLDI, pp. 213–223 (2005)
GrammaTech (2012) Codesonar. http://www.grammatech.com/products/codesonar/overview.html. Accessed 15 Aug 2013
Heckman, S.S., Williams, L.A.: A model building process for identifying actionable static analysis alerts. In: International Conference on Software Testing Verification and Validation, ICST, pp. 161–170 (2009)
Heckman, S.S.: Adaptively ranking alerts generated from automated static analysis. ACM Crossroads **14**(1), 7 (2007)
Heckman, S.S., Williams, L.A.: A systematic literature review of actionable alert identification techniques for automated static code analysis. Inf. Softw. Technol. **53**(4), 363–387 (2011)
Holzmann, G.J., Najm, E., Serhrouchni, A.: Spin model checking: an introduction. STTT **2**(4), 321–327 (2000)
Hooimeijer, P., Weimer, W.: Modeling bug report quality. In: Automated Software Engineering ASE, pp. 34–43 (2007)
Hosseini, H., Nguyen, R., Godfrey, M.W.: A market-based bug allocation mechanism using predictive bug lifetimes. In: Software Maintenance and Reengineering (CSMR), 2012 16th European Conference on, IEEE, pp. 149–158 (2012)
Hovemeyer, D., Pugh, W.: Finding bugs is easy. In: Object-oriented programming systems, languages, and applications (OOPSLA) (2004)
IBM (2012) T.J. Watson Libraries for Analysis (WALA). http://wala.sourceforge.net. Accessed 15 Aug 2013
Jocham R (2005) Jcsc - java coding standard checker. http://jcsc.sourceforge.net/. Accessed 15 Aug 2013
Kawrykow, D., Robillard, M.P.: Non-essential changes in version histories. In: Proceedings of the 33rd International Conference on Software Engineering ICSE (2011)
Kim, S., Ernst, M.D.: Prioritizing warning categories by analyzing software history. In: Mining Software Repositories MSR (2007)

Kim, S., Whitehead, E.J.: How long did it take to fix bugs? In: Proceedings of the 2006 International Workshop on Mining Software Repositories, ACM, pp. 173–174 (2006)

Kim, S., Zimmermann, T., Whitehead, E.J., Zeller, A.: Predicting faults from cached history. In: ISEC, pp. 15–16 (2008)

Kremenek, T., Engler, D.R.: Z-ranking: using statistical analysis to counter the impact of static analysis approximations. In: Proceedings of the Static Analysis: 10th International Symposium SAS (2003)

Liang, G., Wu, L., Wu, Q., Wang, Q., Xie, T., Mei, H.: Automatic construction of an effective training set for prioritizing static analysis warnings. In: Automated Software Engineering ASE (2010)

Nagappan, N., Ball, T., Zeller, A.: Mining metrics to predict component failures. In: Proceedings of the International Conference on Software Engineering ICSE, pp. 452–461 (2006)

Nagappan, N., Ball, T.: Use of relative code churn measures to predict system defect density. In: Proceedings of the 27th International Conference on Software Engineering ICSE, pp. 284–292 (2005)

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

Nethercote, N., Seward, J.: Valgrind: a framework for heavyweight dynamic binary instrumentation. In: Proceedings of the 2007 Programming Language Design and Implementation Conference PLDI, pp. 89–100 (2007)

Nielson, F., Nielson, H.R., Hankin, C.: Principles of Program Analysis. Springer, Berlin (2005)

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

Pan, K., Kim, S., Jr, E.J.W.: Toward an understanding of bug fix patterns. Emp. Softw. Eng. **14**(3), 286–315 (2009)

Park, J.W., Lee, M.W., Kim, J., won Hwang, S., Kim, S.: CosTriage: a cost-aware triage algorithm for bug reporting systems. In: Proceedings of the Association for the Advancement of. Artificial Intelligence (AAAI) (2011)

Rutar, N., Almazan, C.B., Foster, J.S.: A comparison of bug finding tools for java. In: Proceedings of the 15th International Symposium on ISSRE, pp. 245–256 (2004)

Ruthruff, J.R., Penix, J., Morgenthaler, J.D., Elbaum, S.G., Rothermel, G.: Predicting accurate and actionable static analysis warnings: an experimental approach. In: Proceedings of the International Conference on Software Engineering (ICSE), pp. 341–350 (2008)

Sen, K., Marinov, D., Agha, G.: CUTE: a concolic unit testing engine for C. In: ESEC/SIGSOFT FSE, pp. 263–272 (2005)

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

Spacco, J., Hovemeyer, D., Pugh, W.: Tracking defect warnings across versions. In: Proceedings of the 2006 International Workshop on Mining Software Repositories, ACM, MSR '06, pp. 133–136 (2006)

Sun, C., Lo, D., Wang, X., Jiang, J., Khoo, S.C.: A discriminative model approach for accurate duplicate bug report retrieval. In: Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering ICSE, pp. 45–54 (2010)

Tassey, G.: The economic impacts of inadequate infrastructure for software testing. National Institute of Standards and Technology Planning Report 02–32002 (2002)

Thung, F., Lo, D., Jiang, L., Lucia, Rahman, F., Devanbu, P.T.: When would this bug get reported? In: Proceedings of the 28th IEEE International Conference on Software Maintenance ICSM, pp. 420–429 (2012)

Visser, W., Mehlitz, P.: Model checking programs with Java PathFinder. In: SPIN (2005)

Weeratunge, D., Zhang, X., Sumner, W.N., Jagannathan. S.: Analyzing concurrency bugs using dual slicing. In: Proceedings of the 19th International Symposium on Software Testing and Analysis ISSTA, pp. 253–264 (2010)

Weiss, C., Premraj, R., Zimmermann, T., Zeller, A.: How long will it take to fix this bug? In: Proceedings of the Fourth International Workshop on Mining Software Repositories, MSR '07 (2007)

Wu, R., Zhang, H., Kim, S., Cheung, S.C.: Relink: recovering links between bugs and changes. In: Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering SIGSOFT FSE, pp. 15–25 (2011a)

Wu, W., Zhang, W., Yang, Y., Wang, Q.: Drex: developer recommendation with k-nearest-neighbor search and expertise ranking. In: Proceedings of the Software Engineering Conference (APSEC), 2011 18th Asia Pacific, IEEE, pp. 389–396 (2011b)

Xia, X., Lo, D., Wang, X., Zhou, B.: Accurate developer recommendation for bug resolution. In: Proceedings of the 20th Working Conference on Reverse Engineering (2013)

Xie, Y., Aiken, A.: Saturn: a scalable framework for error detection using boolean satisfiability. ACM Trans. Program. Lang. Syst. **29**(3), 16 (2007)

Xie, X., Zhang, W., Yang, Y., Wang, Q.: Dretom: developer recommendation based on topic models for bug resolution. In: Proceedings of the 8th International Conference on Predictive Models in Software Engineering, ACM, pp. 19–28 (2012)