Noname manuscript No.
(will be inserted by the editor)

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: date / Accepted: date

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

Ferdian Thung
School of Information Systems, Singapore Management University
E-mail: ferdiant.2013@phdis.smu.edu.sg

Lucia
School of Information Systems, Singapore Management University
E-mail: lucia.2009@phdis.smu.edu.sg

David Lo
School of Information Systems, Singapore Management University
E-mail: davidlo@smu.edu.sg

Lingxiao Jiang
School of Information Systems, Singapore Management University
E-mail: lxjiang@smu.edu.sg

Foyzur Rahman
University of California, Davis
E-mail: mfrahman@ucdavis.edu

Premkumar T. Devanbu
University of California, Davis
E-mail: ptdevanbu@ucdavis.edu

find that although many field defects could be detected by static bug-finding tools, a substantial proportion of defects could not be flagged. We also analyze the types of tool warnings that are more effective in finding field defects and characterize the types of missed defects. 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 research question, and 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 investigate this research 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 (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 find that many of these defects could be detected by FindBugs, PMD, JLint, CheckStyle, and JCSC, but a number of them remain undetected.

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

2 Bug-Finding Tools

In this section, we first provide a short survey of different bug-finding tools that could be grouped into: static, dynamic, and machine learning based. We then present the 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 several static bug-finding tools that make use of warning patterns for bug detection. These tools are lightweight and can scale to large programs. On the downside, these tools do not consider the specifications of a system, and may miss defects due to specification violations.

Other bug-finding tools also use dynamic analysis techniques, such as dynamic slicing (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 group 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.

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.

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

Identification of Faulty Lines. The above process gives us a set of real-life defects along with the set of changes that fix them. To find the corresponding root causes, we perform a manual process based on the treatments of the defects. Kawrykow and Robillard (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 Figures 2 and 4.

Figure 2 describes a bug fixing activity where one line of code is changed by modifying the operator == to !=. It is easy for us to identify the faulty line which is the line that gets changed. A more complicated case is shown in Figure 4. There are four sets of faulty lines that could be inferred from the *diff*: one is line 259 (marked with *), where an unnecessary method call needs to be removed; a similar fault is at line 262; the third set of faulty lines are at lines 838-340, and they are condition checks that should be removed; the fourth one is at line 887 and it misses a pre-condition check. For this case, the *diff* is much larger than the faulty lines that are manually identified.

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

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]

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

Fig. 3: Identification of Root Causes (Faulty Lines) from Treatments [Ambiguous Case]

AspectJ - file name= org.aspectj/modules/weaver/src/org/aspectj/weaver/bcel/LazyMethodGen.java

Buggy version	**Fixed version**
*Line 259: lng.setStart(null);	Line is deleted
*Line 262: lng.setEnd(null);	Line is deleted

*Line 838: if (i instanceof LocalVariableInstruction) {
*Line 839: int index = ((LocalVariableInstruction)i).getIndex();
*Line 840: if (lvt.getSlot() == index) {
Line 841: if (localVariableStarts.get(lvt) == null) {
Line 842: localVariableStarts.put(lvt, jh);
Line 843: }
Line 844: localVariableEnds.put(lvt, jh);
Line 845: }
Line 846: }
	Line 836: if (localVariableStarts.get(lvt) == null) {
	Line 837: localVariableStarts.put(lvt, jh);
	Line 838: }
	Line 839: localVariableEnds.put(lvt, jh);

Line 882: keys.addAll(localVariableStarts.keySet());
Line 883: Collections.sort(keys,new Comparator() {
Line 884: public int compare(Object a,Object b) {
Line 885: LocalVariableTag taga = (LocalVariableTag)a;
Line 886: LocalVariableTag tagb = (LocalVariableTag)b;
*Line 887: return taga.getName().compareTo(tagb.getName());
Line 888: }});
Line 889: for (Iterator iter = keys.iterator(); iter.hasNext();) {
	Line 870: keys.addAll(localVariableStarts.keySet());
	Line 871-874: *//these lines are commented codes*
	Line 875: Collections.sort(keys, new Comparator() {
	Line 876: public int compare(Object a, Object b) {
	Line 877: LocalVariableTag taga = (LocalVariableTag) a;
	Line 878: LocalVariableTag tagb = (LocalVariableTag) b;
	Line 879: if (taga.getName().startsWith("arg")) {
	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. 4: Identification of Root Causes (Faulty Lines) from Treatments [Complex Case]

using an index notation `Faulty[]`. We refer to the set of lines in `Faulty` that correspond to the i^{th} defect as `Faulty[i]`.

3.2 Extraction of Warnings

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

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

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

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

3.3 Extraction of Missed Defects

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

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

3.4 Overall Approach

Our overall approach is illustrated by the pseudocode in Figure 5. Our approach takes in a bug repository (e.g., Bugzilla or JIRA), a code repository (e.g., CVS or Git), and a bug-finding tool (e.g., FindBugs, PMD, 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 tool *misses*, *partially captures*, or *fully captures* a defect. The final statistics is output at line 20.

Procedure IdentifyMissedDefects
Inputs:
BugRepo : Bug Repository
CodeRepo : Code Repository
BFTool : Bug-Finding Tool
Output:
Statistics of Defects that are Missed and Captured (Fully or Partially)
Method:
1: Let *Stats* = {}
2: For each bug report *br* in `BugRepo`
3: *// Step 1: Extract faulty lines of code*
4: Let *fixC* = *br*'s corresponding fix commit in `CodeRepo`
5: Let *bugC* = Revision before *fixC* in `CodeRepo`
6: Let *diff* = The difference between *fixC* and *bugC*
7: Extract faulty lines from *diff*
8: Let $Faulty_{br}$ = Faulty lines in *bugC*
9: *// Step 2: Get warnings*
10: Run *BFTool* on *bugC*
11: Let $Warning_{br}$ = Flagged lines in *bugC* by *BFTool*
12: *// Step 3: Detect missed defects*
13: Let $Common = Faulty_{br} \cap Warning_{br}$
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.

4 Empirical Evaluation

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

4.1 Research Questions & Datasets

We would like to answer the following research questions:

RQ1 How many real-life reported and fixed defects from Lucene, Rhino, and AspectJ are missed by state-of-the-art static bug-finding tools?
RQ2 What types of warnings reported by the tools are most effective in detecting actual defects?
RQ3 What are some characteristics of the defects missed by the tools?
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?

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. 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). The average sizes of Lucene, Rhino, and AspectJ are around 265,822, 75,544, and 448,176 lines of code (LOC) respectively. We show the numbers of unambiguous defects that we are able to manually locate root causes from the three datasets in Table 1 together with the total numbers of defects available in the datasets and the average faulty lines per defect.

Table 1: Number of Defects for Various Datasets

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

[1] `http://lucene.apache.org/core/`
[2] `http://www.mozilla.org/rhino/`
[3] `http://www.eclipse.org/aspectj/`

4.2 Experimental Results

4.2.1 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 Table 2. We do not summarize across software projects as the numbers of warnings for different projects differ greatly and ummarization 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.

Consider All Defects.

Lucene. For Lucene, as shown in Table 2, we find that with all five tools, 35.7% of all defects could be fully identified (i.e., all faulty lines `Faulty[i]` of a defect i are flagged). Another 28.6% of all defects could also be partially identified (i.e., some but not all faulty lines in `Faulty[i]` are flagged). Still, 35.7% of all defects could not be flagged by the tools. Thus, the five tools are effective but are not very successful in capturing Lucene defects. Among the tools, PMD captures the most numbers of bugs, followed by FindBugs, CheckStyle, JCSC, and finally JLint.

Rhino. For Rhino, as shown in Table 2, we find that with all five tools, 95% 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 Table 2, we find that with all five tools, 71.1% of all defects could be fully captured. Also, another 28.3% of all defects could be partially captured. 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 3. We note that PMD flags the most numbers of lines of code, and likely produces more false positives than others, while JLint flags the least numbers of lines of code. With respect to the average sizes of programs (see Column "Avg # LOC" in Table 4), the tools may flag 0.2% to 56.9% of the whole program as buggy.

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

Tools vs.	**Lucene**			**Rhino**			**AspectJ**		
Programs	**Miss**	**Partial**	**Full**	**Miss**	**Partial**	**Full**	**Miss**	**Partial**	**Full**
FindBugs	71.4%	14.3%(5)	14.3%(8)	10.0%	0%	90.0%(179)	33.6%	19.7%(91)	46.7%(137)
JLint	82.1%	10.7%(3)	7.1%(4)	80.0%	10.0%(2)	10.0%(3)	97.4%	1.3%(2)	1.3%(2)
PMD	50.0%	14.3%(11)	35.7%(13)	15.0%	5.0%(18)	80.0%(157)	3.9%	27.6%(251)	68.4%(196)
CheckStyle	71.4%	14.3%(5)	14.3%(5)	65%	35%(47)	0%	32.2%	38.2%(272)	29.6%(61)
JCSC	64.3%	25%(15)	10.7%(3)	100%	0%	0%	86.8%	9.9%(38)	3.3%(5)
All	35.7%	28.6%(11)	35.7%(22)	5.0%	0%	95.0%(181)	0.7%	28.3%(274)	71.1%(206)

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

Tools vs. Programs	**Lucene**	**Rhino**	**AspectJ**
FindBugs	33,208.82	40,511.55	83,320.09
JLint	515.54	330.6	720.44
PMD	124,281.5	42,999.3	198,065.51
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

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 3. To further understanding these numbers, we present more statistics about the warnings generated by the tools for each of the three programs in Table 4. Column "Avg # Warning" provides the average 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).

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 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 5. A value x in each

Table 4: Unfixed Warnings by Various Defect Finding Tools

Software	Tool	Avg # Warning	Avg # Flagged Line Per Warning	Avg # Unfixed	Avg # LOC
Rhino	FindBugs	177.9	838.72	40,502.6	75,543.8
	JLint	34	1	330.35	
	PMD	11,864.95	21.12	42,990.55	
	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	

cell in the table means that our random tool would have $x \times 100\%$ chance to get at least as good results as the actual tool for either partially or fully capturing the bugs. The values in the table imply the actual tools may indeed detect more bugs than random tools, although they may produce many false positives. However, some tools for some programs, such as PMD for Lucene, may not be much better than random tools. If there is no correlation between the warnings generated by the actual tools with actual defects, the tools should not perform differently from the random tools. Our results show that this is not the case at least for some tools with some programs.

Table 5: p-values 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

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 Tables 6.

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

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

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

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

Tools vs. Programs	**Lucene**			**Rhino**			**AspectJ**		
	Miss	**Partial**	**Full**	**Miss**	**Partial**	**Full**	**Miss**	**Partial**	**Full**
FindBugs	66.7%	14.3%	19.0%	13.3%	0%	86.7%	32.0%	14.8%	53.1%
JLint	80.9%	9.5%	9.5%	86.7%	0%	13.3%	96.9%	1.6%	1.6%
PMD	38.1%	14.3%	47.6%	20.0%	0%	80.0%	2.3%	21.1%	76.6%
Checkstyle	66.67%	19.05%	14.29%	92.31%	7.69%	0%	35.94%	28.9%	35.16%
JCSC	76.19%	9.52%	14.29%	100%	0%	0%	89.84%	6.25%	14.29%
All	38.1%	14.3%	47.6%	6.7%	0%	93.3%	0.8%	20.3%	78.9%

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 7. We notice that under this stricter requirement, very few of the defects fully captured by the tools (see Column "Full") are strictly captured by the same tools (see Column "Strict"). Thus, although the faulty lines might be flagged by the tools, the warning messages from the tools may not have sufficient information for the developers to understand the defects.

Table 7: 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

4.2.2 RQ2: Effectiveness of Different Warnings

We show the effectiveness of various warning families of FindBugs, PMD, JLint, CheckStyle, and JCSC in flagging defects in Tables 8, 9, and 10 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. Correctness related warnings include a field that masks a superclass's field, invocation of toString to an array, etc.

Table 8: Percentages of Defects that are Missed, Partially Captured, and Fully Captured for Different Warning Families of FindBugs

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

PMD. For PMD, as shown in Table 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.

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

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

JLint. For JLint, as shown in Table 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.

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

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

CheckStyle. For CheckStyle, as shown in Table 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.

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

Warning Family	**Miss**	**Partial**	**Full**
Sizes	26%	23.5%	50.5%
Regular expression	39.5%	21%	39.5%
White space	34%	28.5%	37.5%
Java documentation	42.5%	32%	25.5%
Miscellaneous	54.5%	28%	17.5%

JCSC. For JCSC, as shown in Table 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.

Table 12: Percentages of Defects that are Missed, Partially Captured, and Fully Captured for Different Warning Families of JCSC

Warning Family	**Miss**	**Partial**	**Full**
General	76.5%	16.5%	7%
Java documentation	91%	7%	2%
Metrics	95%	4.5%	0.5%
Field	94.5%	5%	0.5%
Method	98%	2%	0%

4.2.3 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 categorize 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 13.

Table 13: 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

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

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. 6: Example of A Defect Missed By All Tools

4.2.4 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 14 describes the distribution of severe and non-severe defects among the 200 defects that we analyze in this study.

Table 14: Distribution of Defects of Various Severity Levels

Type	**Number of Defects**
Severe	45
Non-Severe	155

We investigate the number of false negatives for each category of defects. We show the results for severe and non-severe defects in Tables 15 and 16 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 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, where only 33.3% and 50% of the defects are missed respectively. For non-severe defects in Lucene, PMD and FindBugs are the top two tools that effectively capture non-severe bugs where only 62.5% of the defects are missed. 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 where only 16.67% and 11.11% of the defects are missed. For AspectJ, PMD and CheckStyle are the top two tools that could more effectively capture both severe (i.e., 3.2% and 29.03% of the defects are missed by PMD and CheckStyle respectively) and non-severe defects (i.e., 2.48% and 33.06% of the defects are missed by PMD and CheckStyle respectively). FindBugs could also effectively capture non-severe defects in AspectJ where only 33.06% of the defects are missed.

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

Table 15: Percentages of Severe Defects that are Missed, Partially Captured, and Fully Captured

Tools vs. Programs	**Lucene**			**Rhino**			**AspectJ**		
	Miss	**Partial**	**Full**	**Miss**	**Partial**	**Full**	**Miss**	**Partial**	**Full**
FindBugs	83.3%	0%	16.7%	0%	0%	100.0%	35.5%	25.8%	38.7%
JLint	83.3%	0%	16.7%	100.0%	0%	0%	96.8%	3.2%	0%
PMD	33.3%	8.3%	58.3%	0%	0%	100.0%	3.2%	35.5%	61.3%
CheckStyle	66.67%	8.33%	25%	50%	50%	0%	29.03%	51.61%	19.35%
JCSC	50%	25%	25%	100%	0%	0%	96.77%	3.23%	0%
All	16.67%	25%	58.33%	0%	0%	100.0%	0%	38.71%	61.29%

Table 16: Percentages of Non-Severe Defects that are Missed, Partially Captured, and Fully Captured

Tools vs. Programs	**Lucene**			**Rhino**			**AspectJ**		
	Miss	**Partial**	**Full**	**Miss**	**Partial**	**Full**	**Miss**	**Partial**	**Full**
FindBugs	62.5%	25%	12.5%	11.11%	0%	88.89%	33.06%	18.18%	48.76%
JLint	81.25%	18.75%	0%	83.33%	11.11%	5.56%	97.52%	0.83%	1.65%
PMD	62.5%	18.75%	18.75%	16.67%	5.56%	77.78%	2.48%	27.27%	70.25%
CheckStyle	75%	18.75%	6.25%	66.67%	33.33%	0%	33.06%	34.71%	32.23%
JCSC	75%	25%	0%	100%	0%	0%	84.3%	11.57%	4.13%
All	50%	31.25%	18.75%	5.56%	0%	94.44%	0.83%	25.62%	73.55%

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.

Summary. Overall, we find that among the five tools, PMD captures the most numbers of bugs for both severe and non-severe defects in most datasets. Also, all the five tools together could capture severe defects more effectively as compared to non-severe defects.

4.2.5 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.

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 Jr, 2006; Weiss et al, 2007; Hosseini et al, 2012). Following Thung et al (2012), we divide time to fix a defect into 2 classes: $\leq$30 days, and >30 days.

We show the results for $\leq$30 days and and >30 days defects in Tables 17 and 18 respectively. Generally, we find that >30 days defects can be better captured by the tools than $\leq$30 days defects. By using the five tools together (i.e., "All" rows in the tables), for defects that are fixed within 30 days, we could partially or fully capture all defects of AspectJ, and miss 14.9% and 40% defects of AspectJ and Lucene respectively. By using the five tools together, for defects that are fixed in more than 30 days, all defects of Lucene and Rhino can be captured, while only 0.79% of AspectJ defects are missed.

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: only 56% and 68% of defects fixed within 30 days could not be captured by PMD and JCSC respectively; 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 more than 30 days: only 28.57% of defects fixed within 30 days could not be identified by PMD and Findbugs; also, FindBugs could capture all Rhino's defects that are fixed in more than 30 days and only 7.69% of

Table 17: Percentages of Defects Fixed Within 30 Days that are Missed, Partially Captured, and Fully Captured

Tools vs. Programs	**Lucene**			**Rhino**			**AspectJ**		
	Miss	**Partial**	**Full**	**Miss**	**Partial**	**Full**	**Miss**	**Partial**	**Full**
FindBugs	76%	8%	16%	28.57%	0%	71.43%	11.54%	23.08%	65.38%
JLint	88%	4%	8%	85.71%	0%	14.29%	96.15%	0%	3.85%
PMD	56%	8%	36%	28.57%	0%	71.43%	0%	26.92%	73.08%
CheckStyle	76%	8%	16%	85.71%	14.29%	0%	38.46%	30.77%	30.77%
JCSC	68%	20%	12%	100%	0%	0%	96.15%	3.85%	0%
All	40%	24%	36%	14.29%	0%	85.71%	0%	23.08%	76.92%

Table 18: Percentages of Defects Fixed Above 30 Days that are Missed, Partially Captured, and Fully Captured

Tools vs. Programs	**Lucene**			**Rhino**			**AspectJ**		
	Miss	**Partial**	**Full**	**Miss**	**Partial**	**Full**	**Miss**	**Partial**	**Full**
FindBugs	33.33%	66.67%	0%	0%	0%	100%	38.1%	19.05%	42.86%
JLint	33.33%	66.67%	0%	84.62%	15.38%	0%	97.62%	1.59%	0.79%
PMD	0%	66.67%	33.33%	7.69%	7.69%	84.62%	2.38%	30.16%	67.46%
CheckStyle	33.33%	66.67%	0%	53.85%	46.15%	0%	30.95%	39.68%	29.37%
JCSC	33.33%	66.67%	0%	100%	0%	0%	84.92%	11.11%	3.97%
All	0%	66.67%	33.33%	0%	0%	100.0%	0.79%	29.37%	69.84%

the >30 days defects could not be captured by PMD. 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: only 38.46%, 11.54%, and 0% of defects fixed within 30 days could not be identified by CheckStyles, Findbugs, and PMD respectively; also, only 38.1%, 30.95%, and 2.38% of defects fixed more than 30 days could not be identified by FindBugs, CheckStyles, and PMD respectively.

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.

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: ≤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 ≤2 and >2 resolvers defects in Tables 19 and 20 respectively. When using the 5 tools together, we find that they are almost equally effective in capturing ≤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 (i.e., 35.71%) of ≤2 and >2 resolvers defects. For the AspectJ dataset, by using all the 5 tools, we miss 0% and 1.23% of ≤2 and >2 resolvers defects respectively. However, for the Rhino dataset, by using all the 5 tools together, we can more effectively capture > 2 resolvers defects than ≤2 resolvers defects, i.e., 0% of > 2 resolvers defects and 25% of ≤2 resolvers defects are missed.

Also, we find that among all the five tools, PMD could more effectively capture both ≤2 and >2 resolvers defects. The top two most effective tools in capturing ≤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 ≤2 and >2 resolvers defects: only 42.86% and 57.14% of ≤2 resolvers defects could not be captured by PMD and JCSC respectively; also, only 57.14% and 71.43% of >2 resolvers defects could not be identified by PMD and JCSC respectively. For Rhino's defects, PMD and FindBugs are the most effective tools to capture both ≤2 and >2 resolvers defects: only 25% and 50% of ≤2 resolvers defects could not be captured by PMD and Findbugs; also, FindBugs could capture all Rhino's >2 resolvers defects while only 12.5% of the defects could not be captured by PMD. For AspectJ's defects, PMD, FindBugs, and CheckStyles are the most effective tools to capture both ≤2 and >2 resolvers defects: only 35.21%, 33.8%, and 2.82% of ≤2 resolvers defects

Table 19: Percentages of Defects Fixed by at Most 2 Resolvers that are Missed, Partially Captured, and Fully Captured

Tools vs. Programs	**Lucene**			**Rhino**			**AspectJ**		
	Miss	**Partial**	**Full**	**Miss**	**Partial**	**Full**	**Miss**	**Partial**	**Full**
FindBugs	64.29%	21.43%	14.29%	50%	0%	50%	33.8%	18.31%	47.89%
JLint	64.29%	21.43%	14.29%	75%	0%	25%	97.18%	1.41%	1.41%
PMD	42.86%	14.29%	42.86%	25%	25%	50%	2.82%	26.76%	70.42%
CheckStyle	57.14%	14.29%	28.57%	75%	25%	0%	35.21%	35.21%	29.58%
JCSC	57.14%	21.43%	21.43%	100%	0%	0%	88.73%	8.45%	2.82%
All	35.71%	21.43%	42.86%	25%	0%	75%	0%	25.35%	74.65%

Table 20: Percentages of Defects Fixed by More than 2 Resolvers that are Missed, Partially Captured, and Fully Captured

Tools vs. Programs	**Lucene**			**Rhino**			**AspectJ**		
	Miss	**Partial**	**Full**	**Miss**	**Partial**	**Full**	**Miss**	**Partial**	**Full**
FindBugs	78.57%	7.14%	14.29%	0%	0%	100%	33.33%	20.99%	45.68%
JLint	100%	0%	0%	87.5%	12.5%	0%	97.53%	1.23%	1.23%
PMD	57.14%	14.29%	28.57%	12.5%	0%	87.5%	1.23%	32.1%	66.67%
CheckStyle	85.71%	14.29%	0%	62.5%	37.5%	0%	29.63%	40.74%	29.63%
JCSC	71.43%	28.57%	0%	100%	0%	0%	85.19%	11.11%	3.7%
All	35.71%	35.71%	28.57%	0%	0%	100.0%	1.23%	30.86%	67.9%

could not be captured by CheckStyles, Findbugs, and PMD respectively; also, only 33.33%, 29.63%, and 1.23% of AspectJ's >2 resolvers defects could not be captured by FindBugs, CheckStyles, and PMD respectively.

Summary. Overall, we find that by using the five tools together, we can capture >2 resolvers defects more effectively than ≤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 ≤2 and >2 resolvers defects.

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: ≤10 lines churned, and >10 lines churned.

We show the results for ≤10 and >10 lines churned defects in Tables 21 and 22. The performance of the 5 tools (used together) in capturing ≤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 ≤10 lines churned defects better than >10 lines churned defects (33.33% vs. 38.46% ≤10 and >10 lines churned defects are missed respectively). However, for Rhino, using the 5 tools together, we can capture >10 lines churned defects better than ≤10 lines churned defects (7.14% vs. 0% ≤10 and >10 lines churned defects are missed respectively). For AspectJ, using the 5 tools together, we can capture >10 lines churned defects and ≤10 lines churned defects equally well (0.78% vs. 0% ≤10 and >10 lines churned defects are missed respectively).

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 (i.e., only 40% and 40% of the defects are missed by PMD and CheckStyles respectively), while PMD and JCSC are the most effective tools to identify >10 lines churned defects (i.e., only 61.54% of the defects are missed by both tools). For Rhino's defects, PMD and FindBugs are the most effective tools to capture both ≤10 and >10 lines churned defects: both tools could capture all Rhino's >10 lines churned defects, and only 21.43% and 14.29% of Rhino's ≤10 lines churned defects could not be captured by the tools. For AspectJ's defects, PMD, FindBugs, and CheckStyles are the most effective tools to identify for both ≤10 and >10 lines churned defects: only 34.11%, 33.33%, and 2.33% of ≤10 lines churned defects could not be identified by CheckStyles, Findbugs, and PMD respectively; also, only 34.78%, 21.74%, and 0% of AspectJ's >10 lines churned defects could not be identified by FindBugs, CheckStyles, and PMD respectively.

Table 21: Percentages of Defects Fixed by Churning at Most 10 Lines that are Missed, Partially Captured, and Fully Captured

Tools vs. Programs	**Lucene**			**Rhino**			**AspectJ**		
	Miss	**Partial**	**Full**	**Miss**	**Partial**	**Full**	**Miss**	**Partial**	**Full**
FindBugs	66.67%	20%	13.33%	14.29%	0%	85.71%	33.33%	19.38%	47.29%
JLint	80%	13.33%	6.67%	92.86%	0%	7.14%	96.9%	1.55%	1.55%
PMD	40%	20%	40%	21.43%	0%	78.57%	2.33%	27.13%	70.54%
CheckStyle	60%	20%	20%	85.71%	14.29%	0%	34.11%	32.56%	33.33%
JCSC	66.67%	20%	13.33%	100%	0%	0%	89.15%	6.98%	3.88%
All	33.33%	26.67%	40%	7.14%	0%	92.86%	0.78%	25.58%	73.64%

Table 22: Percentages of Defects Fixed by Churning More than 10 Lines that are Missed, Partially Captured, and Fully Captured

Tools vs. Programs	**Lucene**			**Rhino**			**AspectJ**		
	Miss	**Partial**	**Full**	**Miss**	**Partial**	**Full**	**Miss**	**Partial**	**Full**
FindBugs	76.92%	7.69%	15.38%	0%	0%	100%	34.78%	21.74%	43.48%
JLint	84.62%	7.69%	7.69%	66.67%	33.33%	0%	100%	0%	0%
PMD	61.54%	7.69%	30.77%	0%	16.67%	83.33%	0%	43.48%	56.52%
CheckStyle	84.62%	7.69%	7.69%	16.67%	83.33%	0%	21.74%	69.57%	8.7%
JCSC	61.54%	30.77%	7.69%	100%	0%	0%	73.91%	26.09%	0%
All	38.46%	30.77%	30.77%	0%	0%	100.0%	0%	43.48%	56.52%

Table 23: 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

Summary. Overall, we find that the performance of the 5 tools (used together) in capturing ≤10 lines churned defects are similar to that in capturing >10 lines churned defects. Also among the five tools, PMD could more effectively identify ≤10 and >10 lines churned defects.

4.2.6 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 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 23 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 Table 24. 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. For Lucene, 33.33%, 66.66%, and 66.67% of its assignment related defects could not be captured by PMD, FindBugs, and CheckStyle. 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 Table 25. We find that using the five tools together, we could capture all conditional related defects in Rhino, and only 0.99% of AspectJ's and 38.46% of Lucene conditional related defects could not be captured by the tools. 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. For Lucene, PMD only misses 46.15% of conditional related defects, while the other tools miss more than 50% of

Table 24: Percentages of Assignment Related Defects that are Missed, Partially Captured, and Fully Captured

Tools vs. Programs	Lucene			Rhino			AspectJ		
	Miss	Partial	Full	Miss	Partial	Full	Miss	Partial	Full
FindBugs	66.66%	16.67%	16.67%	0%	0%	100.0%	11.11%	11.11%	77.78%
JLint	100%	0%	0%	100.0%	0%	0%	94.44%	5.56%	0%
PMD	33.33%	16.67%	50%	0%	0%	100.0%	11.11%	5.56%	83.33%
CheckStyle	66.67%	16.67%	16.67%	33.33%	66.67%	0%	38.89%	27.78%	33.33%
JCSC	83.33%	16.67%	0%	100%	0%	0%	88.89%	11.11%	0%
All	16.67%	33.33%	50%	0%	0%	100.0%	0%	5.56%	94.44%

Table 25: Percentages of Conditional Related Defects that are Missed, Partially Captured, and Fully Captured

Tools vs. Programs	Lucene			Rhino			AspectJ		
	Miss	Partial	Full	Miss	Partial	Full	Miss	Partial	Full
FindBugs	61.54%	15.38%	23.08%	7.69%	0%	92.31%	30.69%	23.76%	45.54%
JLint	76.92%	15.38%	7.69%	84.62%	15.38%	0%	98.02%	0.99%	0.99%
PMD	46.15%	15.38%	38.46%	15.38%	7.69%	76.92%	0.99%	34.65%	64.36%
CheckStyle	69.23%	15.38%	15.38%	69.23%	30.77%	0%	30.69%	39.6%	29.7%
JCSC	61.54%	23.08%	15.38%	100%	0%	0%	92.08%	6.93%	0.99%
All	38.46%	23.08%	38.46%	0%	0%	100.0%	0.99%	33.66%	65.35%

Table 26: Percentages of Return Related Defects that are Missed, Partially Captured, and Fully Captured

Tools vs. Programs	Lucene			Rhino			AspectJ		
	Miss	Partial	Full	Miss	Partial	Full	Miss	Partial	Full
FindBugs	0%	100%	0%	50%	0%	50%	75%	12.5%	12.5%
JLint	0%	100%	0%	50%	0%	50%	100%	0%	0%
PMD	0%	100%	0%	50%	0%	50%	0%	12.5%	87.5%
CheckStyle	0%	100%	0%	100%	0%	0%	37.5%	37.5%	25%
JCSC	0%	100%	0%	100%	0%	0%	62.5%	25%	12.5%
All	0%	100%	0%	50%	0%	50%	0%	12.5%	87.5%

the defects. PMD and Findbugs could effectively capture Rhino's conditional related defects with only 15.38% and 7.69% missed defects, and only 30.69% 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 Table 26. 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 (50% missed defects). 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, 50% of return related defects could not be captured by PMD, FindBugs, and JLint, while the other tools could not capture all the defects (100% missed defects). For AspectJ, PMD could effectively capture all return related defects without any missed defects, and CheckStyle only missed 37.5% of the defects. The other tools could not effectively capture AspectJ's return related defects as they miss more than 50% of the defects.

We show the results for method invocation related defects in Table 27. 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 defects in Lucene (50% missing defects). 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 66.67% of Lucene's method invocation 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 28.57% missed defects, 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 Table 28. 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 (50% missing defects). 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 50% 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 percent of the defects.

Table 27: Percentages of Method Invocation Related Defects that are Missed, Partially Captured, and Fully Captured

Tools vs. Programs	Lucene			Rhino			AspectJ		
	Miss	**Partial**	**Full**	**Miss**	**Partial**	**Full**	**Miss**	**Partial**	**Full**
FindBugs	100%	0%	0%	0%	0%	100.0%	42.86%	14.29%	42.86%
JLint	83.33%	0%	16.67%	100.0%	0%	0%	95.24%	0%	4.76%
PMD	66.67%	0%	33.33%	0%	0%	100%	0%	33.33%	66.67%
CheckStyle	83.33%	0%	16.67%	100%	0%	0%	28.57%	38.1%	33.33%
JCSC	66.67%	16.67%	16.67%	100%	0%	0%	76.19%	9.52%	14.29%
All	50%	16.67%	33.33%	0%	0%	100.0%	0%	28.57%	71.43%

Table 28: Percentages of Object Usage Related Defects that are Missed, Partially Captured, and Fully Captured

Tools vs. Programs	Lucene			Rhino			AspectJ		
	Miss	**Partial**	**Full**	**Miss**	**Partial**	**Full**	**Miss**	**Partial**	**Full**
FindBugs	100%	0%	0%	0%	0%	100.0%	75%	0%	25%
JLint	100%	0%	0%	100.0%	0%	0%	100%	0%	0%
PMD	100%	0%	0%	0%	0%	100.0%	0%	25%	75%
CheckStyle	100%	0%	0%	0%	100%	0%	50%	50%	0%
JCSC	50%	50%	0%	100%	0%	0%	50%	50%	0%
All	50%	50%	0%	0%	0%	100.0%	0%	25%	75%

Summary. Overall, we find that PMD could outperform other tools in capturing various types of 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. 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.

4.3 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. 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.

5 Related Work

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

5.1 Bug Finding

There are many bug-finding tools proposed in the literature (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 Section 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.

5.2 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.

5.3 Bug Triage

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

Cubranic 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.

5.4 Empirical Studies on Defects

Many studies investigate the nature of defects. Pan et al (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.

6 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.

7 Acknowledgement

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 GC (2006) Who should fix this bug? In: ICSE, pp 361–370

Artho C (2006) Jlint - find bugs in java programs. http://jlint.sourceforge.net/

Ayewah N, Pugh W (2010) The google findbugs fixit. In: ISSTA, pp 241–252

Ayewah N, Pugh W, Morgenthaler JD, Penix J, Zhou Y (2007) Evaluating static analysis defect warnings on production software. In: PASTE, pp 1–8

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

Ball T, Levin V, Rajamani SK (2011) A decade of software model checking with slam. Commun ACM 54(7):68–76

Beyer D, Henzinger TA, Jhala R, Majumdar R (2007) The software model checker blast. STTT 9(5-6):505–525

Brun Y, Ernst MD (2004) Finding latent code errors via machine learning over program executions. In: ICSE, pp 480–490

Burn O (2007) Checkstyle. http://checkstyle.sourceforge.net/

Cadar C, Ganesh V, Pawlowski PM, Dill DL, Engler DR (2008) EXE: Automatically generating inputs of death. ACM Trans Inf Syst Secur 12(2)

Chou A, Yang J, Chelf B, Hallem S, Engler DR (2001) An empirical study of operating system errors. In: SOSP, pp 73–88

Copeland T (2005) PMD Applied. Centennial Books

Corbett JC, Dwyer MB, Hatcliff J, Laubach S, Pasareanu CS, Robby, Zheng H (2000) Bandera: extracting finite-state models from java source code. In: ICSE, pp 439–448

Cousot P, Cousot R (2012) An abstract interpretation framework for termination. In: POPL, pp 245–258

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

Cubranic D, Murphy GC (2004) Automatic bug triage using text categorization. In: SEKE, pp 92–97

Dallmeier V, Zimmermann T (2007) Extraction of bug localization benchmarks from history. In: ASE, pp 433–436

Gabel M, Su Z (2010) Online inference and enforcement of temporal properties. In: ICSE, pp 15–24

Giger E, Pinzger M, Gall H (2011) Comparing fine-grained source code changes and code churn for bug prediction. In: MSR, pp 83–92

Godefroid P, Klarlund N, Sen K (2005) Dart: directed automated random testing. In: PLDI, pp 213–223

GrammaTech (2012) Codesonar. http://www.grammatech.com/products/codesonar/overview.html

Heckman SS (2007) Adaptively ranking alerts generated from automated static analysis. ACM Crossroads 14(1)

Heckman SS, Williams LA (2009) A model building process for identifying actionable static analysis alerts. In: ICST, pp 161–170

Heckman SS, Williams LA (2011) A systematic literature review of actionable alert identification techniques for automated static code analysis. Information & Software Technology 53(4):363–387

Holzmann GJ, Najm E, Serhrouchni A (2000) Spin model checking: An introduction. STTT 2(4):321–327

Hooimeijer P, Weimer W (2007) Modeling bug report quality. In: ASE, pp 34–43

Hosseini H, Nguyen R, Godfrey MW (2012) 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

Hovemeyer D, Pugh W (2004) Finding bugs is easy. In: OOPSLA

IBM (2012) T.J. Watson Libraries for Analysis (WALA). http://wala.sourceforge.net

Jocham R (2005) Jcsc - java coding standard checker. http://jcsc.sourceforge.net/

Kawrykow D, Robillard MP (2011) Non-essential changes in version histories. In: ICSE

Kim S, Ernst MD (2007) Prioritizing warning categories by analyzing software history. In: MSR

Kim S, Jr EJW (2006) How long did it take to fix bugs? In: Proceedings of the 2006 international workshop on Mining software repositories, ACM, pp 173–174

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

Kremenek T, Engler DR (2003) Z-ranking: Using statistical analysis to counter the impact of static analysis approximations. In: SAS

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

Nagappan N, Ball T (2005) Use of relative code churn measures to predict system defect density. In: ICSE, pp 284–292

Nagappan N, Ball T, Zeller A (2006) Mining metrics to predict component failures. In: ICSE, pp 452–461

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

Nethercote N, Seward J (2007) Valgrind: a framework for heavyweight dynamic binary instrumentation. In: PLDI, pp 89–100

Nielson F, Nielson HR, Hankin C (2005) Principles of Program Analysis. Springer

Ostrand TJ, Weyuker EJ, Bell RM (2005) Predicting the location and number of faults in large software systems. IEEE Trans Softw Eng 31:340–355

Pan K, Kim S, Jr EJW (2009) Toward an understanding of bug fix patterns. Empirical Software Engineering 14(3):286–315

Park JW, Lee MW, Kim J, won Hwang S, Kim S (2011) CosTriage: A cost-aware triage algorithm for bug reporting systems. In: AAAI

Rutar N, Almazan CB, Foster JS (2004) A comparison of bug finding tools for java. In: ISSRE, pp 245–256

Ruthruff JR, Penix J, Morgenthaler JD, Elbaum SG, Rothermel G (2008) Predicting accurate and actionable static analysis warnings: an experimental approach. In: ICSE, pp 341–350

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

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

Sun C, Lo D, Wang X, Jiang J, Khoo SC (2010) A discriminative model approach for accurate duplicate bug report retrieval. In: ICSE, pp 45–54

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

Thung F, Lo D, Jiang L, Lucia, Rahman F, Devanbu PT (2012) When would this bug get reported? In: ICSM, pp 420–429

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

Weeratunge D, Zhang X, Sumner WN, Jagannathan S (2010) Analyzing concurrency bugs using dual slicing. In: ISSTA, pp 253–264

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

Wu R, Zhang H, Kim S, Cheung SC (2011a) Relink: recovering links between bugs and changes. In: SIGSOFT FSE, pp 15–25

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

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

Xie X, Zhang W, Yang Y, Wang Q (2012) 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

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