How Many of All Bugs Do We Find? A Study of Static Bug Detectors

Andrew Habib
andrew.a.habib@gmail.com
Department of Computer Science
TU Darmstadt
Germany

Michael Pradel
michael@binaervarianz.de
Department of Computer Science
TU Darmstadt
Germany

ABSTRACT

Static bug detectors are becoming increasingly popular and are widely used by professional software developers. While most work on bug detectors focuses on whether they find bugs at all, and on how many false positives they report in addition to legitimate warnings, the inverse question is often neglected: How many of all real-world bugs do static bug detectors find? This paper addresses this question by studying the results of applying three widely used static bug detectors to an extended version of the Defects4J dataset that consists of 15 Java projects with 594 known bugs. To decide which of these bugs the tools detect, we use a novel methodology that combines an automatic analysis of warnings and bugs with a manual validation of each candidate of a detected bug. The results of the study show that: (i) static bug detectors find a non-negligible amount of all bugs, (ii) different tools are mostly complementary to each other, and (iii) current bug detectors miss the large majority of the studied bugs. A detailed analysis of bugs missed by the static detectors shows that some bugs could have been found by variants of the existing detectors, while others are domain-specific problems that do not match any existing bug pattern. These findings help potential users of such tools to assess their utility, motivate and outline directions for future work on static bug detection, and provide a basis for future comparisons of static bug detection with other bug finding techniques, such as manual and automated testing.

CCS CONCEPTS

• **Software and its engineering** → **Automated static analysis**; **Software testing and debugging**; • **General and reference** → *Empirical studies*;

KEYWORDS

static bug checkers, bug finding, static analysis, Defects4J

ACM Reference Format:
Andrew Habib and Michael Pradel. 2018. How Many of All Bugs Do We Find? A Study of Static Bug Detectors. In *Proceedings of the 2018 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE '18), September 3–7, 2018, Montpellier, France.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3238147.3238213

ASE '18, September 3–7, 2018, Montpellier, France

ACM ISBN 978-1-4503-5937-5/18/09…$15.00
https://doi.org/10.1145/3238147.3238213

1 INTRODUCTION

Finding software bugs is an important but difficult task. For average industry code, the number of bugs per 1,000 lines of code has been estimated to range between 0.5 and 25 [21]. Even after years of deployment, software still contains unnoticed bugs. For example, studies of the Linux kernel show that the average bug remains in the kernel for a surprisingly long period of 1.5 to 1.8 years [8, 24]. Unfortunately, a single bug can cause serious harm, even if it has been subsisting for a long time without doing so, as evidenced by examples of software bugs that have caused huge economic loses and even killed people [17, 28, 46].

Given the importance of finding software bugs, developers rely on several approaches to reveal programming mistakes. One approach is to identify bugs during the development process, e.g., through pair programming or code review. Another direction is testing, ranging from purely manual testing over semi-automated testing, e.g., via manually written but automatically executed unit tests, to fully automated testing, e.g., with UI-level testing tools. Once the software is deployed, runtime monitoring can reveal so far missed bugs, e.g., collect information about abnormal runtime behavior, crashes, and violations of safety properties, e.g., expressed through assertions. Finally, developers use static bug detection tools, which check the source code or parts of it for potential bugs.

In this paper, we focus on static bug detectors because they have become increasingly popular in recent years and are now widely used by major software companies. Popular tools include Google's Error Prone [1], Facebook's Infer [7], or SpotBugs, the successor to the widely used FindBugs tool [10]. These tools are typically designed as an analysis framework based on some form of static analysis that scales to complex programs, e.g., AST-based pattern matching or data-flow analysis. Based on the framework, the tools contain an extensible set of checkers that each addresses a specific bug pattern, i.e., a class of bugs that occurs across different code bases. Typically, a bug detector ships with dozens or even hundreds of patterns. The main benefit of static bug detectors compared to other bug finding approaches is that they find bugs early in the development process, possibly right after a developer introduces a bug. Furthermore, applying static bug detectors does not impose any special requirements, such as the availability of tests, and can be fully automated.

The popularity of static bug detectors and the growing set of bug patterns covered by them raise a question: *How many of all*

real-world bugs do these bug detectors find? Or in other words, what is the recall of static bug detectors? Answering this question is important for several reasons. First, it is an important part of assessing the current state-of-the-art in automatic bug finding. Most reported evaluations of bug finding techniques focus on showing that a technique detects bugs and how precise it is, i.e., how many of all reported warnings correspond to actual bugs rather than false positives. We do not consider these questions here. In contrast, practically no evaluation considers the above recall question. The reason for this omission is that the set of "all bugs" is unknown (otherwise, the bug detection problem would have been solved), making it practically impossible to completely answer the question. Second, understanding the strengths and weaknesses of existing static bug detectors will guide future work toward relevant challenges. For example, better understanding of which bugs are currently missed may enable future techniques to cover previously ignored classes of bugs. Third, studying the above question for multiple bug detectors allows us to compare the effectiveness of existing tools with each other: Are existing tools complementary to each other or does one tool subsume another one? Fourth and finally, studying the above question will provide an estimate of how close the current state-of-the-art is to the ultimate, but admittedly unrealistic, goal of finding all bugs.

To address the question of how many of all bugs do static bug detectors find, we perform an empirical study with 594 real-world bugs from 15 software projects, which we analyze with three widely used static bug detectors. The basic idea is to run each bug detector on a version of a program that contains a specific bug, and to check whether the bug detector finds this bug. While being conceptually simple, realizing this idea is non-trivial for real-world bugs and bug detectors. The main challenge is to decide whether the set of warnings reported by a bug detector captures the bug in question. To address this challenge, we present a novel methodology that combines automatic line-level matching, based on the lines involved in the bug fix, and a manual analysis of the matched lines. The manual analysis is crucial because a bug detector may coincidentally flag a line as buggy due to a reason different from the actual bug, such as an unrelated problem or a false positive. Since our study focuses on a finite set of bugs, we cannot really answer how many of *all* bugs are found. Instead, we approximate the answer by considering a large and diverse set of bugs from various real-world projects.

Our study relates to but significantly differs from a previous study by Thung et al. [39, 40]. Their work also addresses the question of how many of all real-world bugs are found by static checkers. Our work differs in the methodology used to answer this question: We manually validate whether the warnings reported by a tool correspond to a specific bug in the code, instead of checking whether the lines flagged by a tool include the faulty lines. This manual validation leads to significantly different results than the previous study because many warnings coincidentally match a faulty line but are actually unrelated to the specific bug. Another difference is that our study focuses on a more recent and improved generation of static bug detectors. Thung et al.'s study considers what might be called the first generation of static bug detectors for Java, e.g., PMD and CheckStyle. While these tools contributed significantly to the state-of-the-art when they were initially presented, it has also been shown that they suffer from severe limitations, in particular, large numbers of false positives. Huge advances in static bug detection have been made since then. Our study focuses on a novel generation of static bug detectors, including tools that have been adopted by major industry players and that are in wide use.

The main findings of our study are the following:

- The three bug detectors together reveal 27 of the 594 studied bugs (4.5%). This non-negligible number is encouraging and shows that static bug detectors can be beneficial.
- The percentage of detected among all bugs ranges between 0.84% and 3%, depending on the bug detector. This result points out a significant potential for improvement, e.g., by considering additional bug patterns. It also shows that checkers are mostly complementary to each other.
- The majority of missed bugs are domain-specific problems not covered by any existing bug pattern. At the same time, several bugs could have been found by minor variants of the existing bug detectors.

2 METHODOLOGY

This section presents our methodology for studying which bugs are detected by static bug detectors. At first, we describe the bugs (§ 2.1) and bug detection tools (§ 2.2) that we study. Then, we present our experimental procedure for identifying and validating matches between the warnings reported by the bug detectors and the real-world bugs (§ 2.3). Finally, we discuss threats to validity in § 2.4.

2.1 Real-World Bugs

Our study builds on an extended version of the Defects4J data set, a collection of bugs from popular Java projects. In total, the data set consists of 597 bugs that are gathered from different versions of 15 projects. We use Defects4J for this study for three reasons. First, it provides a representative set of real-world bugs that has been gathered independently of our work. The bugs cover a wide spectrum of application domains and have been sampled in a way that does not bias the data set in any relevant way. Second, the data set is widely used for other bug-related studies, e.g., on test generation [38], mutation testing [13], fault localization [26], and bug repair [20], showing that is has been accepted as a representative set of bugs. Third, Defects4J provides not only bugs but also the corresponding bug fixes, as applied by the actual developers. Each bug is associated with two versions of the project that contains the bug: a buggy version, just before applying the bug fix, and a fixed version, just after applying the bug fix. The bug fixes have been isolated by removing any irrelevant code changes, such as new features or refactorings. As a result, each bug is associated with one or more Java classes, i.e., source code files that have been modified to fix the bug. The availability of fixes is important not only to validate that the developers considered a bug as relevant, but also to understand its root cause.

The current official version of Defects4J (version 1.1.0) consists of 395 bugs collected from 6 Java projects. A recent addition to these bugs extends the official release with 202 additional bugs from 9 additional projects.[1] In our work, we use the extended version of the data set and refer to it as "Defects4J". Table 1 lists the projects

[1] https://github.com/rjust/defects4j/pull/112

Table 1: Projects and bugs of Defects4J.

ID	Project	Bugs	
Original Defects4J:			
Chart	JFreeChart	26	
Closure	Google Closure	133	
Lang	Apache commons-lang	64	(65)
Math	Apache commons-math	106	
Mockito	Mockito framwork	38	
Time	Joda-Time	27	
Total of 6 projects		**394**	(395)
Extended Defects4J:			
Codec	Apache commons-codec	21	(22)
Cli	Apache commons-cli	24	
Csv	Apache commons-csv	12	
JXPath	Apache commons-JXPath	14	
Guava	Guava library	9	
JCore	Jackson core module	13	
JDatabind	Jackson data binding	39	
JXml	Jackson XML utilities	5	
Jsoup	Jsoup HTML parser	63	(64)
Total of 9 projects		**200**	(202)
Total of 15 projects		**594**	(597)

and bugs in the data set.[2] We exclude three of the 597 bugs for technical reasons: Lang-48 because Error Prone does not support Java 1.3, and Codec-5 and Jsoup-4 because they introduce a new class in the bug fix, which does not match our methodology that relies on analyzing changes to existing files.

2.2 Static Bug Detectors

We study three static bug detectors for Java: (i) Error Prone [1], a tool developed by Google and is integrated into their Tricorder static analysis ecosystem [36]; (ii) Infer [7], a tool developed and used internally by Facebook; and (iii) SpotBugs, the successor of the pioneering FindBugs [10] tool. These tools are used by professional software developers. For example, Error Prone and Infer are automatically applied to code changes to support manual code review at Google and Facebook, respectively. All three tools are available as open-source. We use the tools with their default configuration.

2.3 Experimental Procedure

Given a set of bugs and a set of static bug detectors, the overall goal of the methodology is to identify those bugs among the set B of provided bugs that are detected by the given tools. We represent a *detected bug* as a tuple (b, w), where $b \in B$ is a bug and w is a warning that points to the buggy code. A single bug b may be detected by multiple warnings, e.g., (b, w_1) and (b, w_2), and a single warning may point to multiple bugs, e.g., (b_1, w) and (b_2, w).

A naive approach to assess whether a tool finds a particular bug would be to apply the tool to the buggy version of the code and to manually inspect each reported warning. Unfortunately, static bug detectors may produce many warnings and manually inspecting each warning for each buggy version of a program does not scale to the number of bugs we consider. Another possible approach is to fully automatically match warnings and bugs, e.g., by assuming that every warning at a line involved in a bug fix points to the respective bug. While this approach solves the scalability problem, it risks to overapproximate the number of detected bugs. The reason is that some warnings may coincidentally match a code location involved in a bug, but nevertheless do not point to the actual bug.

[2]We refer to bugs using the notation ProjectID-N, where N is a unique number.

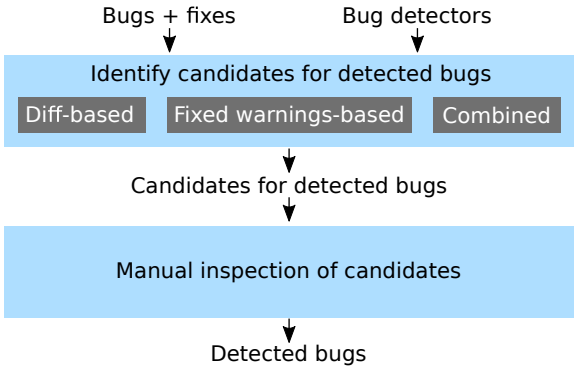


Figure 1: Overview of our methodology.

Our approach to identify detected bugs is a combination of automatic and manual analysis, which reduces the manual effort compared to inspecting all warnings while avoiding the overapproximation problem of a fully automatic matching. To identify the detected bugs, we proceed in two main steps, as summarized in Figure 1. The first step automatically identifies candidates for detected bugs, i.e., pairs of bugs and warnings that are likely to match each other. We apply three variants of the methodology that differ in how to identify such candidates:

- an approach based on differences between the code before and after fixing the bug,
- an approach based on warnings reported for the code before and after fixing the bug, and
- the combination of the two previous approaches.

The second step is to manually inspect all candidates to decide which bugs are indeed found by the bug detectors. This step is important to avoid counting coincidental matches as detected bugs.

2.3.1 *Identifying Candidates for Detected Bugs.*

Common Definitions. We explain some terms and assumptions used throughout this section. Given a bug b, we are interested in the set L_b of *changed lines*, i.e., lines that were changed when fixing the bug. We assume that these lines are the locations where developers expect a static bug detector to report a warning. In principle, this assumption may not hold because the bug location and the fix location may differ. We further discuss this potential threat to validity in § 2.4. We compute the changed lines based on the differences, or short, the diff, between the code just before and just after applying the bug fix. The diff may involve multiple source code files. We compute the changed lines as lines that are modified

or deleted, as these are supposed to directly correspond to the bug. In addition, we consider a configurable window of lines around the location of newly added lines. As a default value, we use a window size of [-1,1].

Applying a bug detector to a program yields a set of warnings. We refer to the sets of warnings for the program just before and just after fixing a bug b as $W_{before}(b)$ and $W_{after}(b)$, or simply W_{before} and W_{after} if the bug is clear from the context. The bug detectors we use can analyze entire Java projects. Since the purpose of our study is to determine whether specific bugs are found, we apply the analyzers only to the files involved in the bug fix, i.e., files that contain at least one changed line $l \in L_b$. We also provide each bug detector the full compile path along with all third-party dependencies of each buggy or fixed program so that inter-project and third-party dependencies are resolved. The warnings reported when applying a bug detector to a file are typically associated with specific line numbers. We refer to the lines that are flagged by a warning w as $lines(w)$.

Diff-based Methodology. One approach to compute a set of candidates for detected bugs is to rely on the diff between the buggy and the fixed versions of the program. The intuition is that a relevant warning should pinpoint one of the lines changed to fix the bug. In this approach, we perform the following for each bug and bug detector:

(1) Compute the lines that are flagged with at least one warning in the code just before the bug fix:

$$L_{warnings} = \bigcup_{w \in W_{before}} lines(w)$$

(2) Compute the candidates of detected bugs as all pairs of a bug and a warning where the changed lines of the bug overlap with the lines that have a warning:

$$B^{diff}_{cand} = \{(b, w) \mid L_b \cap L_{warnings} \neq \emptyset\}$$

For example, the bug in Figure 2a is a candidate for a bug detected by SpotBugs because the tool flagged line 55, which is also in the set of changed lines.

Fixed Warnings-based Methodology. As an alternative approach for identifying a set of candidates for detected bugs, we compare the warnings reported for the code just before and just after fixing a bug. The intuition is that a warning caused by a specific bug should disappear when fixing the bug. In this approach, we perform the following for each bug and bug detector:

(1) Compute the set of fixed warnings, i.e., warnings that disappear after applying the bug fix:

$$W_{fixed} = W_{before} \setminus W_{after}$$

(2) Compute the candidates for detected bugs as all pairs of a bug and a warning where the warning belongs to the fixed warnings set:

$$B^{fixed}_{cand} = \{(b, w) \mid w \in W_{fixed}\}$$

In this step, we do not match warning messages based on line numbers because line numbers may not match across the buggy and fixed files due to added and deleted code. Instead, we compare the messages based on the warning type, category, severity, rank, and code entity, e.g., class, method, and field.

For example, Figure 2c shows a bug that the fixed warnings-based approach finds as a candidate for a detected bug by Error Prone because the warning message reported at line 175 disappears in the fixed version. In contrast, the approach misses the candidate bug in Figure 2a because the developer re-introduced the same kind of bug in line 62, and hence, the same warning is reported in the fixed code.

Combined Methodology. The diff-based approach and the fixed warnings-based approach may yield different candidates for detected bugs. For instance, both approaches identify the bugs in Figure 2c and Figure 2d as candidates, whereas only the diff-based approach identifies the bugs in Figure 2a and Figure 2b. Therefore, we consider as a third variant of our methodology the combination of the fixed warnings and the diff-based approach:

$$B^{combine}_{cand} = B^{diff}_{cand} \cup B^{fixed}_{cand}$$

Unless otherwise mentioned, the combined methodology is the default in the remainder of the paper.

2.3.2 *Manual Inspection and Classification of Candidates.*

The automatically identified candidates for detected bugs may contain coincidental matches of a bug and warning. For example, suppose that a bug detector warns about a potential null dreference at a specific line and that this line gets modified as part of a bug fix. If the fixed bug is completely unrelated to dereferncing a null object, then the warning would not have helped a developer in spotting the bug.

To remove such coincidental matches, we manually inspect all candidates for detected bugs and compare the warning messages against the buggy and fixed versions of the code. We classify each candidate into one of three categories: (i) If the warning matches the fixed bug and the fix modifies lines that affect the flagged bug only, then this is a *full match*. (ii) If the fix targets the warning but also changes other lines of code not relevant to the warning, then it is a *partial match*. (iii) If the fix does not relate to the warning message at all, then it is a *mismatch*.

For example, the bug in Figure 2d is classified as a full match since the bug fix exactly matches the warning message: to prevent a `NullPointerException` on the value returned by `ownerDocument()`, a check for nullness has been added in the helper method `getOutputSettings()`, which creates an empty `Document("")` object when `ownerDocument()` returns null.

As an example of a partial match, consider the bug in Figure 2a. As we discussed earlier in § 2.3.1, the developer attempted a fix by applying proper check and cast in lines 58-63 of the fixed version. We consider this candidate bug a partial match because the fixed version also modifies line 60 in the buggy file by changing the return value of the method `hashCode()`. This change is not related to the warning reported by SpotBugs. It is worth noting that the fact that the developer unfortunately re-introduced the same bug in line 62 of the fixed version does not contribute to the partial matching decision.

Finally, the bug in Figure 2b is an example of a mismatch because the warning reported by Error Prone is not related to the bug fix.

Buggy code:

```
53 @Override
54 public boolean equals(Object o) {
55   return method.equals(o);
56 }
57
58 @Override
59 public int hashCode() {
60   return 1;
61 }
```

Bug fix →

Fixed code:

```
53 @Override
54 public boolean equals(Object o) {
55   if (this == o) {
56     return true;
57   }
58   if (o instanceof DelegatingMethod) {
59     DelegatingMethod that = (DelegatingMethod) o;
60     return method.equals(that.method);
61   } else {
62     return method.equals(o);
63   }
64 }
65
66 @Override
67 public int hashCode() {
68   return method.hashCode();
69 }
```

(a) Bug Mockito-11. Warning by SpotBugs at line 55: Equality check for operand not compatible with `this`. L_b = { 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 67, 68, 69 }. Found by diff-based methodology. Classification: Partial match.

Buggy code:

```
1602 public Dfp multiply(final int x) {
1603   return multiplyFast(x);
1604 }
```

Bug fix →

Fixed code:

```
1602 public Dfp multiply(final int x) {
1603   if (x >= 0 && x < RADIX) {
1604     return multiplyFast(x);
1605   } else {
1606     return multiply(newInstance(x));
1607   }
1608 }
```

(b) Bug Math-17. Warning by Error Prone at line 1602: Missing `@Override`. L_b = { 1602, 1603, 1604, 1605, 1606, 1607, 1608 }. Found by diff-based methodology. Classification: Mismatch.

Buggy code:

```
173 public Week(Date time, TimeZone zone) {
174   // defer argument checking...
175   this(time, RegularTimePeriod.DEFAULT_TIME_ZONE,
          Locale.getDefault());
176 }
```

Bug fix →

Fixed code:

```
173 public Week(Date time, TimeZone zone) {
174   // defer argument checking...
175   this(time, zone, Locale.getDefault());
176 }
```

(c) Bug Chart-8. Warning by Error Prone at line 175: Chaining constructor ignores parameter. L_b = { 175 }. Found by: Diff-based methodology and fixed warnings-based methodology. Classification: Full match.

Buggy code:

```
214 public Document ownerDocument() {
215   if (this instanceof Document)
216     return (Document) this;
217   else if (parentNode == null)
218     return null;
219   else
220     return parentNode.ownerDocument();
221 }
 :
362 protected void outerHtml(StringBuilder accum) {
363       new NodeTraversor(new
            OuterHtmlVisitor(accum, ownerDocument()
            .outputSettings()))
            .traverse(this);
364 }
```

Bug fix →

Fixed code:

```
362 protected void outerHtml(StringBuilder accum) {
363   new NodeTraversor(new OuterHtmlVisitor(accum,
          getOutputSettings())).traverse(this);
364 }
365
366 // if this node has no document (or parent),
       retrieve the default output settings
367 private Document.OutputSettings
       getOutputSettings() {
368   return ownerDocument() != null ?
          ownerDocument().outputSettings() : (new
          Document("")).outputSettings();
369 }
```

(d) Bug Jsoup-59. Warning by Infer at line 363: `null` dereference. L_b = { 363, 364, 365, 366, 367, 368, 369 }. Found by: Diff-based methodology and fixed warnings-based methodology. Classification: Full match.

Figure 2: Candidates for detected bugs and their manual classification.

2.3.3 Error Rate. Beyond the question of how many of all bugs are detected, we also consider the *error rate* of a bug detector. Intuitively, it indicates how many warnings the bug detector reports. We compute the error rate by normalizing the number of reported warnings to the number of analyzed lines of code:

$$ER = \frac{\sum_{b \in B} |W_{before}(b)|}{\sum_{b \in B} \sum_{f \in files(b)} LoC(f)}$$

where $files(b)$ are the files involved in fixing bug b and $LoC(f)$ yields the number of lines of code of a file.

2.4 Threats to Validity

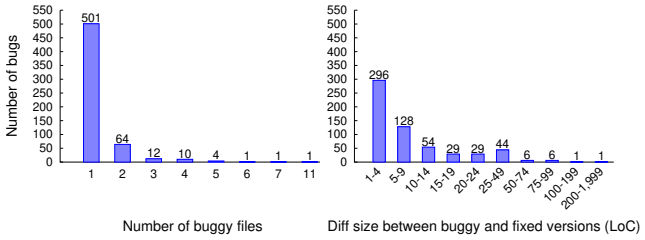
As for all empirical studies, there are some threats to the validity of the conclusions drawn from our results. One limitation is the selection of bugs and bug detectors, both of which may or may not be representative for a larger population. To mitigate this threat, we use a large set of real-world bugs from a diverse set of popular open-source projects. Moreover, the bugs have been gathered independently of our work and have been used in previous bug-related studies [13, 20, 26, 38]. For the bug detectors, we study tools that are widely used in industry and which we believe to be representative for the current state-of-the-art. Despite these efforts, we cannot claim that our results generalize beyond the studied artifacts.

Another threat to validity is that our methodology for identifying detected bugs could, in principle, both miss some detected bugs and misclassify coincidental matches as detected bugs. A reason for potentially missing detected bugs is our assumption that the lines involved in a bug fix correspond to the lines where a developer expects a warning to be placed. In principle, a warning reported at some other line might help a developer to find the bug, e.g., because the warning eventually leads the developer to the buggy code location. Since we could only speculate about such causal effects, we instead use the described methodology. The final decision whether a warning corresponds to a bug is taken by a human and therefore subjective. To address this threat, both authors discussed every candidate for a detected bug where the decision is not obvious.

A final threat to validity results from the fact that static bug detectors may have been used during the development process of the studied projects. If some of the developers of the studied projects use static bug detectors before checking in their code, they may have found some bugs that we miss in this study. As a result, our results should be understood as an assessment of how many of those real-world bugs that are committed to the version control systems can be detected by static bug detectors.

3 EXPERIMENTAL RESULTS

This section presents the results of applying our methodology to 594 bugs and three bug detectors. We start by describing some properties of the studied bugs (§ 3.1) and the warnings reported by the bug detectors (§ 3.2). Next, we report on the candidates for detected bugs (§ 3.3) and how many of them could be manually validated and their kinds (§ 3.4), followed by a comparison of the studied bug detectors (§ 3.5). To better understand the weaknesses of current bug detectors, § 3.6 discusses why the detectors miss many bugs. Finally, § 3.7 empirically compares the three variants of our methodology.



(a) Number of buggy files. **(b) Total size of diffs between buggy and fixed files.**

Figure 3: Properties of the studied bugs.

Table 2: Warnings generated by each tool. The minimum, maximum, and average numbers of warnings are per bug and consider all files involved in the bug fix.

	Warnings				
	Per bug				
Tool	Min	Max	Avg	Total	Error rate
Error Prone	0	148	7.58	4,402	0.01225
Infer	0	36	0.33	198	0.00055
SpotBugs	0	47	1.1	647	0.0018
Total				**5,247**	

3.1 Properties of the Studied Bugs

To better understand the setup of our study, we measure several properties of the 594 studied bugs. Figure 3a shows how many files are involved in fixing a bug. For around 85% of the bugs, the fix involves changing a single source code file. Figure 3b shows the number of lines of code in the diff between the buggy and the fixed versions. This measure gives an idea of how complex the bugs and their fixes are. The results show that most bugs involve a small number of lines: For 424 bugs, the diff size is between one and nine lines of code. Two bugs have been fixed by modifying, deleting, or inserting more than 100 lines.

3.2 Warnings Reported by the Bug Detectors

The first step in our methodology is running each tool on all files involved in each of the bugs. Table 2 shows the minimum, maximum, and average number of warnings per bug, i.e., in the files involved in fixing the bug, the total number of warnings reported by each tool, and the error rate as defined in § 2.3.3. We find that Error Prone reports the highest number of warnings, with a maximum of 148 warnings and an average of 7.58 warnings per bug. This is also reflected by an error rate of 0.01225.

The studied bug detectors label each warning with a description of the potential bug. Table 3 shows the top 5 kinds of warnings reported by each tool. The most frequent kind of warning by Error Prone is about missing the `@Override` annotation when a method

Table 3: Top 5 warnings reported by each static checker.

Warning	Count
Error Prone	
Missing `@Override`	3211
Comparison using reference equality	398
Boxed primitive constructor	234
Operator precedence	164
Type parameter unused in formals	64
Infer	
`null` dereference	90
Thread safety violation	43
Unsafe `@GuardedBy` access	30
Resource leak	29
Method with mutable return type returns immutable collection	1
SpotBugs	
`switch` without `default`	109
Inefficient `Number` constructor	79
Read of unwritten field	45
Method naming convention	37
Reference to mutable object	31

overrides a method with the same signature in its parent class. Infer's most reported kind of warning complains about a potential null dereference. Finally, the most frequent kind of warning by SpotBugs is related to missing the `default` case in a `switch` statement. The question how many of these warnings point to a valid problem (i.e., true positives) is outside of the scope of this paper.

3.3 Candidates for Detected Bugs

Given the number of reported warnings, which totals to 5,247 (Table 2), it would be very time-consuming to manually inspect each warning. The automated filtering of candidates for detected bugs yields a total of 153 warnings and 89 candidates (Table 4), which significantly reduces the number of warnings and bugs to inspect. Compared to all reported warnings, the selection of candidates reduces the number of warnings by 97%.

The number of warnings is greater than the number of candidates because we count warnings and candidates obtained from all tools together and each tool could produce multiple warnings per line(s).

3.4 Validated Detected Bugs

To validate the candidates for detected bugs, we inspect each of them manually. Based on the inspection, we classify each candidate as a full match, a partial match, or a mismatch, as described in § 2.3.2. Overall, the three tools found 31 bugs, as detailed in the table in Figure 4. After removing duplicates, i.e., bugs found by more than one tool, there are 27 unique validated detected bugs.

We draw two conclusions from these results. First, the fact that 27 unique bugs are detected by the three studied bug detectors shows that these tools would have had a non-negligible impact, if they would have been used during the development of the studied

Tool	Bugs
Error Prone	8
Infer	5
SpotBugs	18
Total:	**31**
Total of 27 unique bugs	

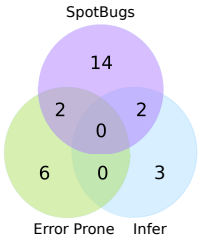


Figure 4: Total number of bugs found by all three static checkers and their overlap.

programs. This result is encouraging for future work on static bug detectors and explains why several static bug detection tools have been adopted in industry. Second, even when counting both partial and full matches, the overall bug detection rate of all three bug detectors together is only 4.5%. While reaching a detection anywhere close to 100% is certainly unrealistic, e.g., because some bugs require a deep understanding of the specific application domain, we believe that the current state-of-the-art leaves room for improvement.

To get an idea of the kinds of bugs the checkers find, we describe the most common patterns that contribute to finding bugs. Out of the eight bugs found by Error Prone, three are due to missing an `@Override` annotation, and two bugs because the execution may fall through a `switch` statement. For the five bugs found by Infer, four bugs are potential `null` deferences. Out of the 18 bugs detected by SpotBugs, three are discovered by pointing to dead local stores (i.e., unnecessarily computed values), and two bugs are potential `null` deferences. Finally, the two bugs found by both Infer and SpotBugs are `null` deferences, whereas the two bugs found by both Error Prone and SpotBugs are a string format error and an execution that may fall through a `switch` statement.

3.5 Comparison of Bug Detectors

The right-hand side of Figure 4 shows to what extent the bug detectors complement each other. SpotBugs finds most of the bugs, 18 of all 27, of which 14 are found only by SpotBugs. Error Prone finds 6 bugs that are not found by any other tool, and Infer finds 3 bugs missed by the other tools. We conclude that the studied tools complement each other to a large extent, suggesting that developers may want to combine multiple tools and that researchers could address the problem of how to reconcile warnings reported by different tools.

3.6 Reasons for Missed Bugs

To better understand why the vast majority of bugs are not detected by the studied bug detectors, we manually inspect and categorize some of the missed bugs. We inspect a random sample of 20 of all bugs that are not detected by any bug detector. For each sampled bug, we try to understand the root cause of the problem by inspecting the diff and by searching for any issue reports associated with the bug. Next, we carefully search the list of bug patterns supported by the bug detectors to determine whether any of the detectors could have matched the bug. If there is a bug detector that relates to the bug, e.g., by addressing a similar bug pattern, then we experiment with variants of the buggy code to understand

why the detector has not triggered an alarm. Based on this process, we have the two interesting findings.

3.6.1 Domain-specific Bugs. First, the majority of the missed bugs (14 out of 20) are domain-specific problems not related to any of the patterns supported by the bug checkers. The root causes of these bugs are mistakes in the implementation of application-specific algorithms, typically because the developer forgot to handle a specific case. Moreover, these bugs manifest in ways that are difficult to identify as unintended without domain knowledge, e.g., by causing an incorrect string to be printed or an incorrect number to be computed. For example, Math-67 is a bug in the implementation of a mathematical optimization algorithm that returns the last computed candidate value instead of the best value found so far. Another example is Closure-110, a bug in a JavaScript compiler that fails to properly handle some kinds of function declarations. Finally, Time-14 is due to code that handles dates but forgot to consider leap years and the consequences of February 29.

3.6.2 Near Misses. Second, some of the bugs (6 out of 20) could be detected with a more powerful variant of an existing bug detector. We distinguish two subcategories of these bugs. On the one hand, the root causes of some bugs are problems targeted by at least one existing bug detector, but the current implementation of the detector misses the bug. These bugs manifest through a behavior that is typically considered unintended, such as infinite recursion or out-of-bounds array accesses. For example, Commons-Csv-7 is caused by accessing an out-of-bounds index of an array, which is one of the bug patterns searched for by SpotBugs. Unfortunately, the SpotBugs checker is intra-procedural, while the actual bug computes the array index in one method and then accesses the array in another method. Another example is Lang-49, which causes an infinite loop because multiple methods call each other recursively, and the conditions for stopping the recursion miss a specific input. Both Error Prone and SpotBugs have checkers for infinite loops caused by missing conditions that would stop recursion. However, these checkers target cases that are easier to identify than Lang-49, which would require inter-procedural reasoning about integer values. A third example in this subcategory is Chart-5, which causes an `IndexOutOfBoundsException` when calling `ArrayList.add`. The existing checker for out-of-bounds accesses to arrays might have caught this bug, but it does not consider `ArrayLists`.

On the other hand, the root causes of some bugs are problems that are similar to but not the same as problems targeted by an existing checker. For example, Commons-Codec-8 is about forgetting to override some methods of the JDK class `FilterInputStream`. While SpotBugs and Error Prone have checkers related to streams, including some that warn about missing overrides, none of the existing checkers targets the methods relevant in this bug.

3.6.3 Implications for Future Work. We draw several conclusions from our inspection of missed bugs. The first and perhaps most important is that there is a huge need for bug detection techniques that can detect domain-specific problems. Most of the existing checkers focus on generic bug patterns that occur across projects and often even across domains. However, as most of the missed bugs are domain-specific, future work should complement the existing detectors with techniques beyond checking generic bug patterns. One promising direction could be to consider informal specifications, such as natural language information embedded in code or available in addition to code.

We also conclude that further work on sophisticated yet practical static analysis is required. Given that several currently missed bugs could have been found by inter-procedural variants of existing intra-procedural analyses suggests room for improvement. The challenge here is to balance precision and recall: Because switching to inter-procedural analysis needs to approximate, e.g., call relationships, this step risks to cause additional false positives. Another promising direction suggested by our results is to generalize bug detectors that have been developed for a specific kind of problem to related problems, e.g., `ArrayLists` versus arrays.

Finally, our findings suggest that some bugs are probably easier to detect with techniques other than static checkers. For example, the missed bugs that manifest through clear signs of misbehavior, such as an infinite loop, are good candidates for fuzz-testing with automated test generators.

Table 4: Candidate warnings (W) and bugs (B) obtained from the automatic matching.

	Approach					
	Diff-based		Fixed warnings		Combined	
Tool	W	B	W	B	W	B
Error Prone	51	33	18	14	53	35
Infer	30	9	14	6	32	11
SpotBugs	51	32	29	22	68	43
Total:	132	74	61	42	**153**	**89**

3.7 Assessment of Methodologies

We compare the three variants of our methodology and validate that the manual inspection of candidates of detected bugs is crucial.

3.7.1 Candidates of Detected Bugs. Our methodology for identifying candidates for detected bugs has three variants (§ 2.3.1). Table 4 compares them by showing for each variant how many warnings and bugs it identifies as candidates. The number of warnings is larger than the number of bugs because the lines involved in a single bug may match multiple warnings. Overall, identifying candidates based on diffs yields many more warnings, 132 in total, than by considering which warnings are fixed by a bug fix, which yields 61 warnings. Combining the two methodologies by considering the union of candidates gives a total of 153 warnings corresponding to 89 bugs. Since more than one static checker could point to the same bug, the total number of unique candidates for detected bugs by all tools together boils down to 79 bugs.

Figure 5 visualizes how the variants of the methodology complement each other. For example, for Error Prone, the fixed warnings-based approach finds 14 candidates, 2 of which are only found by this approach. The diff-based technique finds 21 candidates not found by the fixed warnings approach. Overall, the diff-based and the fixed warnings-based approaches are at least partially complementary, making it worthwhile to study and compare both.

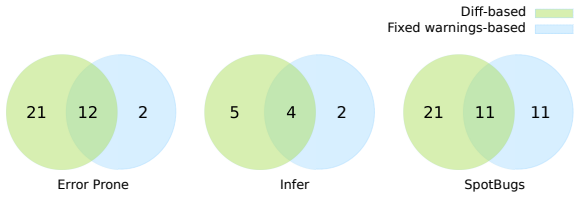


Figure 5: Candidate detected bugs using the two different automatic matching techniques.

3.7.2 Validated Detected Bugs. Figure 6 shows how many of the candidates obtained with the diff-based and the fixed warnings-based approach we could validate during the manual inspection. The left chart of Figure 6a shows the results of manually inspecting each warning matched by the diff-based approach. For instance, out of the 51 matched warnings reported by Error Prone, 6 are full matches and 2 are partial matches, whereas the remaining 43 do not correspond to any of the studied bugs. The right chart in Figure 6a shows how many of the candidate bugs are actually detected by the reported warnings. For example, out of 9 bugs that are possibly detected by Infer, we have validated 3. Figure 6b and Figure 6c show the same charts for the fixed warnings-based approach and the combined approach.

The comparison shows that the diff-based approach yields many more mismatches than the fixed warnings-based approach. Given this result, one may wonder whether searching for candidates only based on fixed warnings would yield all detected bugs. In Figure 7, we see for each bug detector, how many unique bugs are found by the two automatic matching approaches. For both Error Prone and Infer, although the diff-based approach yields a large number of candidates, the fixed warnings-based methodology is sufficient to identify all detected bugs. For SpotBugs, though, one detected bug would be missed when inspecting only the warnings that have been fixed when fixing the bug. The reason is bug Mockito-11 in Figure 2a. The fixed warnings-based methodology misses this bug because the bug fix accidentally re-introduces another warning of the same kind, at line 62 of the fixed code.

In summary, we find that the fixed warnings-based approach requires less manual effort while revealing almost all detected bugs. This result suggests that future work could focus on the fixed warnings-based methodology, allowing such work to manually inspect even more warnings than we did.

3.7.3 Manual Inspection. Table 4 shows that the combined approach yields 153 candidate warnings corresponding to 89 (79 unique) bugs. However, the manual validation reveals that only 34 of those warnings and a corresponding number of 31 (27 unique) bugs correspond to actual bugs, whereas the remaining matches are coincidental. Out of the 34 validated candidates, 22 are full matches and 12 are partial matches (Figure 6c). In other words, 78% of the candidate warnings and 66% of the candidate bugs are spurious matches, i.e., the warning is about something unrelated to the specific bug and only happens to be on the same line.

These results confirm that the manual step in our methodology is important to remove coincidental matches. Omitting the manual inspection would skew the results and mislead the reader to believe that more bugs are detected. This skewing of results would be even stronger for bug detectors that report more warnings per line of code, as evidenced in an earlier study [39].

To ease reproducibility and to enable others to build on our results, full details of all results are available online.[3]

4 RELATED WORK

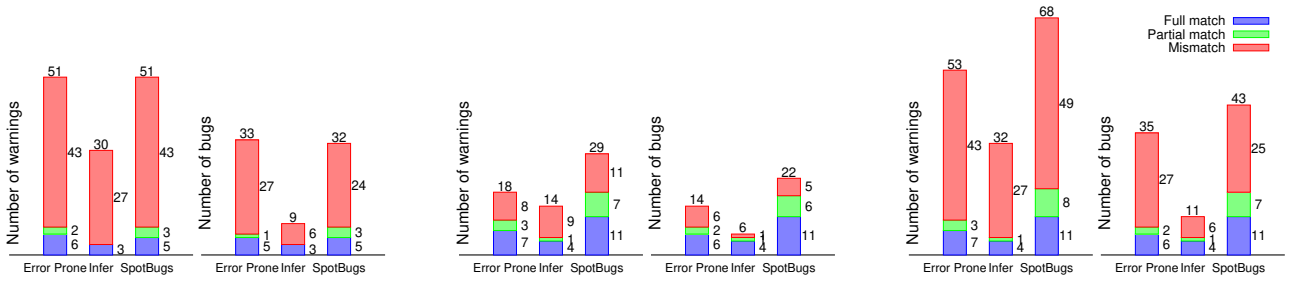
Studies of Static Bug Detectors. Most existing studies of static bug detectors focus on precision, i.e., how many of all warnings reported by a tool point to actual bugs [34, 41, 45]. In contrast, our study asks the opposite question: What is the recall of static bug detectors, i.e., how many of all (known) bugs are found? Another difference to existing studies is our choice of static bug detectors: To the best of our knowledge, this is the first paper to study the effectiveness of Error Prone, Infer, and SpotBugs.

The perhaps most related existing work is a study by Thung et al.[39, 40] that also focuses on the recall of static bug detectors. Our work differs in the methodology, because we manually validate each detected bug, and in the selection of bugs and bug detectors, because we focus on more recent, industrially used tools. As a result of our improved methodology, our results differ significantly: While they conclude that between 64% and 99% of all bugs are partially or fully detected, we find that only 4.5% of all studied bugs are found. The main reason for this difference is that some of the bug detectors used by Thung et al. report a large number of warnings. For example, a single tool alone reports over 39,000 warnings for the Lucene benchmark (265,821 LoC), causing many lines to be flagged with at least one warning with error rate 0.15 (§ 2.3.3). Since their methodology fully automatically matches source code lines and lines with warnings, most bugs appear to be found. Instead, we manually check whether a warning indeed corresponds to a particular bug to remove false matches.

To facilitate evaluating bug detection techniques, several benchmarks of bugs have been proposed. BugBench [18] consists of 17 bugs in C programs. Cifuentes et al. significantly extend this benchmark, resulting in 181 bugs that are sampled from four categories, e.g., buffer overflows. They use the benchmark to compare 4 bug detectors using an automatic, line-based matching to measure recall. Future work could apply our semi-manual methodology to their bug collection to study whether our results generalize to C programs. Rahman et al. compare the benefits of static bug detectors and statistical bug prediction [31]. To evaluate whether an approach would have detected a particular bug, their study compares the lines flagged with warnings and the lines changed to fix a bug, which roughly corresponds to the first step of our methodology and lacks a manual validation whether a warning indeed points to the bug. Johnson et al. conducted interviews with developers to understand why static bug detectors are (not) used [11]. The study suggests that better ways of presenting warnings to developers and integrating bug detectors into the development workflow would increase the usage of these tools.

Studies of Other Bug Detection Techniques. Other studies consider bug detection techniques beyond static bug detectors, test generation [2, 38]. One of these studies [38] also considers bugs

[3] https://github.com/sola-da/StaticBugCheckers



(a) Diff-based approach. (b) Fixed warnings-based approach. (c) Combined approach.

Figure 6: Manual inspection of candidate warnings and bugs from the two automatic matching approaches.

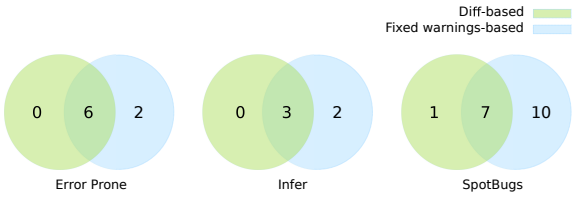


Figure 7: Actual bugs found using the two different automatic matching techniques.

in Defects4J and finds that most test generators detect less than 20% of these bugs. Another study focuses on manual bug detection and compares the effectiveness of code reading, functional testing, and structural testing [44]. Finally, Legunsen et al. study to what extent checking API specifications via runtime monitoring reveals bugs. All these studies are complementary to ours, as we focus on static bug detectors. Future work could study how well different bug finding techniques complement each other.

Studies of Bugs and Bug Fixes. An important step toward improving bug detection is to understand real-world bugs. To this end, studies have considered several kinds of bugs, including bugs in the Linux kernel [8], concurrency bugs [19], and correctness [23] and performance bugs [37] in JavaScript. Pan et al. study bug fixes and identify recurring, syntactical patterns [25]. Work by Ray et al. reports that statistical language models trained on source code show buggy code to have higher entropy than fixed code [32], which can help static bug detectors to prioritize their warnings.

Static Bug Detectors and Real-World Deployments. The lint tool [12], originally presented in 1978, is one of the pioneers on static bug detection. Since then, static bug detection has received significant attention by researchers, including work on finding API misuses [22, 30, 43], name-based bug detection [29], security bugs [6], finding violations of inferred programmer beliefs [9] and other kinds of anomaly detection [16], bug detection based on statistical language models [42], and on detecting performance bugs [27]. Several static bug detection approaches have been adopted by industry. Bessey et al. report their experiences from commercializing static bug detectors [5]. Ayewah et al. share lessons learned from applying FindBugs, the predecessor of the SpotBugs tool considered in our study, at Google [3, 4]. A recent paper describes the success of deploying a name-based static checker [33]. Since many bug detectors suffer from a large number of warnings, some of which are false positives, an important question is which warnings to inspect first. Work on prioritizing analysis warnings addresses this question based on the frequency of true and false positives [15], the version history of a program [14], and statistical models based on features of warnings and code [35].

5 CONCLUSION

This paper investigates how many of all bugs can be found by current static bug detectors. To address this question, we study a set of 594 real-world Java bugs and three widely used bug detection tools. The core of our study is a novel methodology to assess the recall of bug detectors, which identifies detected bugs through a combination of automatic, line-based matching and manual validation of candidates for detected bugs. The main findings of the study include the following: (i) Static bug detectors find a non-negligible number, 27, of real-world bugs, showing that static bug detectors are certainly worthwhile and developers should use them during development. (ii) Different bug detectors complement each other in the sense that they detect different subsets of the studied bugs. Users of static bug detectors may want to run more than one tool. (iii) The large majority (95.5%) of the studied bugs are not detected by the studied tools, showing that there is ample of room for improving the current state-of-the-art. (iv) Many of the missed bugs are due to domain-specific problems instead of generic bug patterns, while some currently missed bugs could be found with more powerful variants of existing checks.

Overall, we conclude that future work should focus not only on reducing false positives, as highlighted by previous studies, but also on detecting a larger fraction of all real-world bugs, e.g., by considering a larger variety of bug patterns and by searching domain-specific bugs. Beyond these findings, which are relevant to users and creators of static bug detectors, our results can serve as a basis for a future study on comparing static analysis with other bug detection techniques, such as manual and automated testing.

ACKNOWLEDGMENTS

We thank the anonymous reviewers and Julia Lawall for their valuable comments. This work was supported in part by the German Research Foundation within the Emmy Noether project ConcSys and the Perf4JS project, by the German Federal Ministry of Education and Research and by the Hessian Ministry of Science and the Arts within CRISP, and by the Hessian LOEWE initiative within the Software-Factory 4.0 project.

REFERENCES

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

[2] Mohammad Moein Almasi, Hadi Hemmati, Gordon Fraser, Andrea Arcuri, and Janis Benefelds. 2017. An Industrial Evaluation of Unit Test Generation: Finding Real Faults in a Financial Application. In *39th IEEE/ACM International Conference on Software Engineering: Software Engineering in Practice Track, ICSE-SEIP 2017, Buenos Aires, Argentina, May 20-28, 2017*. 263–272.

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

[4] Nathaniel Ayewah and William Pugh. 2010. The Google FindBugs fixit. In *Proceedings of the Nineteenth International Symposium on Software Testing and Analysis, ISSTA 2010, Trento, Italy, July 12-16, 2010*. 241–252.

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

[6] Fraser Brown, Shravan Narayan, Riad S. Wahby, Dawson R. Engler, Ranjit Jhala, and Deian Stefan. 2017. Finding and Preventing Bugs in JavaScript Bindings. In *2017 IEEE Symposium on Security and Privacy, SP 2017, San Jose, CA, USA, May 22-26, 2017*. 559–578.

[7] Cristiano Calcagno, Dino Distefano, Jérémy Dubreil, Dominik Gabi, Pieter Hooimeijer, Martino Luca, Peter O'Hearn, Irene Papakonstantinou, Jim Purbrick, and Dulma Rodriguez. 2015. Moving fast with software verification. In *NASA Formal Methods Symposium*. Springer, 3–11.

[8] Andy Chou, Junfeng Yang, Benjamin Chelf, Seth Hallem, and Dawson R. Engler. 2001. An Empirical Study of Operating System Errors. In *Symposium on Operating Systems Principles (SOSP)*. 73–88. http://doi.acm.org/10.1145/502034.502042

[9] Dawson Engler, David Yu Chen, Seth Hallem, Andy Chou, and Benjamin Chelf. 2001. Bugs as Deviant Behavior: A General Approach to Inferring Errors in Systems Code. In *Symposium on Operating Systems Principles (SOSP)*. ACM, 57–72.

[10] David Hovemeyer and William Pugh. 2004. Finding bugs is easy. In *Companion to the Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. ACM, 132–136.

[11] Brittany Johnson, Yoonki Song, Emerson Murphy-Hill, and Robert Bowdidge. 2013. Why don't software developers use static analysis tools to find bugs?. In *Proceedings of the 2013 International Conference on Software Engineering*. IEEE Press, 672–681.

[12] S. C. Johnson. 1978. Lint, a C Program Checker.

[13] René Just, Darioush Jalali, Laura Inozemtseva, Michael D Ernst, Reid Holmes, and Gordon Fraser. 2014. Are mutants a valid substitute for real faults in software testing?. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 654–665.

[14] Sunghun Kim and Michael D. Ernst. 2007. Which warnings should I fix first?. In *European Software Engineering Conference and Symposium on Foundations of Software Engineering (ESEC/FSE)*. ACM, 45–54.

[15] Ted Kremenek and Dawson R. Engler. 2003. Z-Ranking: Using Statistical Analysis to Counter the Impact of Static Analysis Approximations. In *International Symposium on Static Analysis (SAS)*. Springer, 295–315.

[16] Bin Liang, Pan Bian, Yan Zhang, Wenchang Shi, Wei You, and Yan Cai. 2016. AntMiner: Mining More Bugs by Reducing Noise Interference. In *ICSE*.

[17] J. L. Lions. 1996. ARIANE 5 Flight 501 Failure. Report by the Inquiry Board. European Space Agency.

[18] Shan Lu, Zhenmin Li, Feng Qin, Lin Tan, Pin Zhou, and Yuanyuan Zhou. 2005. Bugbench: Benchmarks for evaluating bug detection tools. In *Workshop on the Evaluation of Software Defect Detection Tools*.

[19] Shan Lu, Soyeon Park, Eunsoo Seo, and Yuanyuan Zhou. 2008. Learning from mistakes: a comprehensive study on real world concurrency bug characteristics. In *Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. ACM, 329–339.

[20] Matias Martinez, Thomas Durieux, Romain Sommerard, Jifeng Xuan, and Martin Monperrus. 2017. Automatic repair of real bugs in java: A large-scale experiment on the defects4j dataset. *Empirical Software Engineering* 22, 4 (2017), 1936–1964.

[21] Steve McConnell. 2004. *Code Complete: A Practical Handbook of Software Construction, Second Edition*. Microsoft Press.

[22] Tung Thanh Nguyen, Hoan Anh Nguyen, Nam H. Pham, Jafar M. Al-Kofahi, and Tien N. Nguyen. 2009. Graph-based mining of multiple object usage patterns. In *European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. ACM, 383–392.

[23] Frolin S. Ocariza Jr., Kartik Bajaj, Karthik Pattabiraman, and Ali Mesbah. 2013. An Empirical Study of Client-Side JavaScript Bugs. In *Symposium on Empirical Software Engineering and Measurement (ESEM)*. 55–64.

[24] Nicolas Palix, Gaël Thomas 0001, Suman Saha, Christophe Calvès, Julia L. Lawall, and Gilles Muller. 2011. Faults in linux: ten years later. In *Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. ACM, 305–318.

[25] Kai Pan, Sunghun Kim, and E. James Whitehead Jr. 2009. Toward an understanding of bug fix patterns. *Empirical Software Engineering* 14, 3 (2009), 286–315.

[26] Spencer Pearson, José Campos, René Just, Gordon Fraser, Rui Abreu, Michael D Ernst, Deric Pang, and Benjamin Keller. 2017. Evaluating and improving fault localization. In *Software Engineering (ICSE), 2017 IEEE/ACM 39th International Conference on*. IEEE, 609–620.

[27] Jacques A. Pienaar and Robert Hundt. 2013. JSWhiz: Static analysis for JavaScript memory leaks. In *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization, CGO 2013, Shenzhen, China, February 23-27, 2013*. 11:1–11:11.

[28] Kevin Poulsen. 2004. Software Bug Contributed to Blackout. SecurityFocus.

[29] Michael Pradel and Thomas R. Gross. 2011. Detecting anomalies in the order of equally-typed method arguments. In *International Symposium on Software Testing and Analysis (ISSTA)*. 232–242.

[30] Michael Pradel, Ciera Jaspan, Jonathan Aldrich, and Thomas R. Gross. 2012. Statically Checking API Protocol Conformance with Mined Multi-Object Specifications. In *International Conference on Software Engineering (ICSE)*. 925–935.

[31] Foyzur Rahman, Sameer Khatri, Earl T Barr, and Premkumar Devanbu. 2014. Comparing static bug finders and statistical prediction. In *Proceedings of the 36th International Conference on Software Engineering*. ACM, 424–434.

[32] Baishakhi Ray, Vincent Hellendoorn, Saheel Godhane, Zhaopeng Tu, Alberto Bacchelli, and Premkumar T. Devanbu. 2016. On the "naturalness" of buggy code. In *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016*. 428–439.

[33] Andrew Rice, Edward Aftandilian, Ciera Jaspan, Emily Johnston, Michael Pradel, and Yulissa Arroyo-Paredes. 2017. Detecting Argument Selection Defects. In *OOPSLA*.

[34] Nick Rutar, Christian B. Almazan, and Jeffrey S. Foster. 2004. A Comparison of Bug Finding Tools for Java. In *International Symposium on Software Reliability Engineering (ISSRE)*. IEEE Computer Society, 245–256.

[35] Joseph R. Ruthruff, John Penix, J. David Morgenthaler, Sebastian Elbaum, and Gregg Rothermel. 2008. Predicting accurate and actionable static analysis warnings: an experimental approach. In *International Conference on Software Engineering (ICSE)*. 341–350.

[36] Caitlin Sadowski, Jeffrey van Gogh, Ciera Jaspan, Emma Söderberg, and Collin Winter. 2015. Tricorder: Building a Program Analysis Ecosystem. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1 (ICSE '15)*. IEEE Press, Piscataway, NJ, USA, 598–608. http://dl.acm.org/citation.cfm?id=2818754.2818828

[37] Marija Selakovic and Michael Pradel. 2016. Performance Issues and Optimizations in JavaScript: An Empirical Study. In *International Conference on Software Engineering (ICSE)*. 61–72.

[38] Sina Shamshiri, René Just, José Miguel Rojas, Gordon Fraser, Phil McMinn, and Andrea Arcuri. 2015. Do Automatically Generated Unit Tests Find Real Faults? An Empirical Study of Effectiveness and Challenges (T). In *30th IEEE/ACM International Conference on Automated Software Engineering, ASE 2015, Lincoln, NE, USA, November 9-13, 2015*. 201–211.

[39] Ferdian Thung, Lucia, David Lo, Lingxiao Jiang, Foyzur Rahman, and Premkumar T. Devanbu. 2012. To what extent could we detect field defects? an empirical study of false negatives in static bug finding tools. In *Conference on Automated Software Engineering (ASE)*. ACM, 50–59.

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

[41] Stefan Wagner, Jan Jürjens, Claudia Koller, and Peter Trischberger. 2005. Comparing Bug Finding Tools with Reviews and Tests. In *International Conference on Testing of Communicating Systems (TestCom)*. Springer, 40–55.

[42] Song Wang, Devin Chollak, Dana Movshovitz-Attias, and Lin Tan. 2016. Bugram: bug detection with n-gram language models. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, ASE 2016, Singapore, September 3-7, 2016*. 708–719.

[43] Andrzej Wasylkowski and Andreas Zeller. 2009. Mining Temporal Specifications from Object Usage. In *International Conference on Automated Software Engineering (ASE)*. IEEE, 295–306.

[44] Murray Wood, Marc Roper, Andrew Brooks, and James Miller. 1997. Comparing and Combining Software Defect Detection Techniques: A Replicated Empirical Study. In *Software Engineering - ESEC/FSE '97, 6th European Software Engineering Conference Held Jointly with the 5th ACM SIGSOFT Symposium on Foundations of Software Engineering, Zurich, Switzerland, September 22-25, 1997, Proceedings*. 262–277.

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

[46] Michael Zhivich and Robert K. Cunningham. 2009. The Real Cost of Software Errors. *IEEE Security & Privacy* 7, 2 (2009), 87–90.