FSE: U: Bugs in the Wild: Examining the Effectiveness of Static Analyzers at Finding Real-World Bugs

David A. Tomassi
University of California, Davis
United States
datomassi@ucdavis.edu

ABSTRACT

Static analysis is a powerful technique to find software bugs. In past years, a few static analysis tools have become available for developers to find certain kinds of bugs in their programs. However, there is little evidence on how effective the tools are in finding bugs in real-world software. In this paper, we present a preliminary study on the popular static analyzers ErrorProne, SpotBugs, and Infer. Specifically, we consider 320 real Java bugs from the BugSwarm dataset, and determine which of these bugs can potentially be found by the analyzers, and how many are indeed detected. We find that 40.3% of the bugs are candidates for detection by SpotBugs and Infer while only 30.3% are candidates for ErrorProne. Our evaluation shows that the analyzers are relatively easy to incorporate into the tool chain of diverse projects that use the Maven build system. However, the analyzers are not as effective detecting the bugs under study, with only one bug successfully detected by SpotBugs.

CCS CONCEPTS

• **Software and its engineering → Software testing and debugging**;

KEYWORDS

bug finding tools, static analysis, BugSwarm

1 RESEARCH PROBLEM

Static analysis tools (e.g., [1, 3, 4, 7–9]) for bug finding are useful in software development. These tools analyze the program to find common patterns that lead to troublesome code. However, there is a question of their effectiveness for finding real bugs in a diverse world of software. Such information could be valuable for developers when determining which tool(s) to employ, and for the tool developers themselves when identifying opportunities to further improve their tools.

In this paper, we evaluate the popular static analyzers ErrorProne [1], SpotBugs [9], and Infer [4]. ErrorProne and SpotBugs are static analysis tools available as plugins for a variety of Java build systems (Maven, Gradle, etc), while Infer is available via a standalone tool. ErrorProne applies patterns to find bugs ranging from infinite recursion to incompatible argument types in Java programs. Similarly, SpotBugs is a pattern-based tool that finds bugs that include null pointer dereferences, casting errors, and infinite loops. Infer has a collection of different analyses ranging from abstract interpretation to linters, and finds bugs related to thread safety, resource leaks, and null pointer dereferences. We study 320 real-world Java bugs from 48 distinct Java projects found in the BugSwarm dataset [11]. Our goal is to answer the following research questions: (1) How many of the bugs could potentially be found by the analyzers?, and (2) How many of these bugs are successfully detected?

The first challenge in this study is to manually examine the 320 bugs (and their respective fixes) to determine whether these bugs are within the scope of the kinds of bugs found by ErrorProne, SpotBugs, and Infer. The second challenge is to run the analyzers on each of the program versions (320×2), to produce the corresponding bug reports. Third, we analyze the bug reports to determine whether the analyzers successfully detect a given bug.

We examined the documentation of the tools to determine the kinds of bugs that the tools can detect. We manually inspected each bug in our dataset, and found that 30.3%, 40.3%, and 40.3% of the bugs under study fall into the categories of bugs reported to be found by ErrorProne, SpotBugs, and Infer, respectively. It is important to note that SpotBugs and Infer find bugs in the same categories according to our classification. The tools are relatively easy to incorporate into the tool chain of the projects, and successfully analyze on average 76% of the relevant buggy programs. However, the tools were unable to detect any bugs, except for one bug that is detected by SpotBugs.

2 RELATED WORK

Previous studies have compared static-analysis tools based on various metrics. Ayewah et al. [10] conducted a user-centric study of the static analyzer FindBugs in which the tool is run against the Sun's JDK implementation to determine what bugs are reported.

Rutar et al. [15] performed a comparison of Bandera, ESC/Java 2, FindBugs, JLint, and PMD on five mid-sized programs to determine the strengths and weaknesses of each tool. Their analysis included a taxonomy of detected bugs, time to analysis completion, and number of warnings and bugs found. Similarly, Wagner et al. [17] compared FindBugs, PMD, and QJ Pro on five projects, four of which are unnamed industrial projects. The authors also categorized the types of bugs found by each tool and compared their bug detection rates. Unlike the above studies, we do not compare tools based on metrics such as run-time or number of bugs reported. Instead, we examine tools independently to determine their effectiveness at finding real bugs within their scope.

Thung et al. [16] looks at three static analyzers FindBugs, JLint, and PMD using three different programs as their benchmark. They measure the false negatives rates of each tool. Their methodology of determining if a bug was found by a tool is through an overlap of reported lines by the tool and the faulty lines extracted from a diff. They report missed, partially and fully captured bugs. Our study differs with respect to the tools, benchmark, and how we determine if a bug was found.

Table 1: Characteristics of Top Five Projects

Project Repo	# Bugs	LOC	# Tests
raphw/byte-buddy	105	170,885	4,533
checkstyle/checkstyle	50	135,950	1,662
tananaev/traccar	33	59,803	237
junkdog/artemis-odb	21	70,916	430
yamcs/yamcs	12	286,251	336

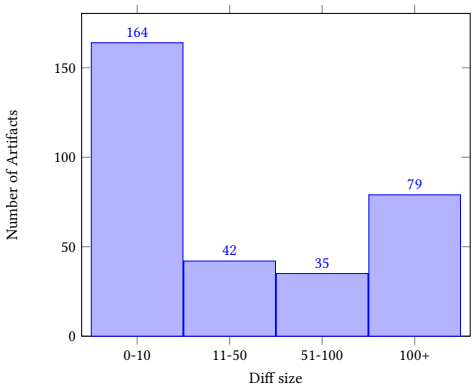

Figure 1: Diff size for the artifacts.

Nanda et al. [14] present a portal for running multiple static analysis tools and aggregating their reports to present a comprehensive bug report. The authors run FindBugs, Safe, and XYLEM on two projects Apache Ant and an unnamed project from IBM. They compare the different types of interprocedural analysis, shallow and deep, in a small study, looking at FindBugs and XYLEM measuring precision and recall. We compare different tools with a more comprehensive dataset and measure different metrics.

Most recently, and concurrent to our work, Habib and Pradel [12] conducted an evaluation of ErrorProne, Infer, and SpotBugs on the Defects4J dataset [13]. The authors ran the tools on 594 bugs from 15 Java projects, and reported that 95.5% of the bugs could not be found by any of the tools. Our study is conducted on a different set of bugs from 48 Java projects. Furthermore, we first examine the bugs to identify candidates for the tools under study, and then report how many of the candidate bugs are detected.

3 TECHNICAL APPROACH

ErrorProne, SpotBugs, and Infer are not designed to find *all* possible bugs. This section answers the following questions: (1) How many of a sample of real bugs could potentially be found by the analyzers?, and (2) How many of these bugs are indeed detected?

Bug Selection. We consider 320 real Java bugs from 48 distinct projects that use the Maven build system. We randomly sampled these bugs from 1,768 Java bugs found in the BugSwarm dataset. For each bug, BugSwarm provides a Docker image that includes buggy and fixed versions of the source code, regression tests, and scripts to build and run the tests. Each bug has at least one failing test that exposes the bug. Table 1 lists the five projects with the most bugs in the sample. The size of the projects and the number of regression tests highlights that the projects are not toy programs and have a significant number of tests. Figure 1 shows the number of artifacts with a given diff size[1]. A majority of the artifacts in our sample have a small diff, which means that the fix is small. This is beneficial when inspecting the code to classify the bugs, and when inspecting bug reports to determine the effectiveness of the tools.

Manual Bug Inspection. In conjunction with the BugSwarm developers, we carefully examined the 320 bugs under study. Table 2 shows the identified bug categories. We referred to [2, 5, 6] to determine which of these categories the tools can detect. We found that bugs classified as Logic Error, Casting Error, and Resource Leak are good candidates for all three tools. Additionally, NullPointerException bugs are also handled by SpotBugs and Infer. Categories such as Test, Configuration, and Dependency Errors were found to fall outside the scope of the tools.

Bug Report Generation. We ran ErrorProne, SpotBugs, and Infer on *all* bugs in our sample, but focused on the bugs that fit within the selected categories to determine *tool effectiveness*. ErrorProne and SpotBugs are available as a build system plugin, so we modified the Maven `pom` files. Infer is a standalone tool, so there was no need to modify any of the projects files. After incorporating the tools into each program's build process, the rest was significantly easier as BugSwarm provides Docker images for each bug with scripts to build the code. We measured effectiveness based on how many of the bugs within the criteria were detected. We ran the analyzers twice for each bug (the buggy and fixed versions). The tools produce a report for each run.

Bug Report Inspection. We manually inspected all reports for each artifact. Reports include the description and line number(s) of each bug detected. We examine these reports to find whether there was any reference to the diff between the buggy and fixed version of the code, the context of the bug, and a diff of the buggy and fixed reports. Our methodology of determining if a bug was found is establishing that three conditions are held, this is done as follows. (1) We will first check to see if the lines warned about by the tool overlap with the lines in the diff. (2) We check the context of the bug warning. If the bug is a null pointer dereference then we would expect the bug warning message and type to be related to that. (3) Additionally, we compare the reports for both versions; if a bug report were related to the actual bug, we would expect it to not be present in the report produced for the fixed version of the code. If all of (1), (2), and (3) conditions are met then we say the tool found the bug, otherwise it did not.

4 PRELIMINARY RESULTS

How many bugs could potentially be detected by the tools? After our manual bug classification, and studying the bug finding capabilities of each tool, we found that 97 (30.3%), 129 (40.3%), and 129 (40.3%) out of 320 bugs could potentially be detected by ErrorProne, SpotBugs, and Infer, respectively. This can be seen in Table 3.

[1]The diff size refers to the number of lines of code that differ between the buggy version of the program and the fixed version.

Table 2: Manual Bug Classification

Bug Category	Description	Number of artifacts
Logic Error	The bug is logical in nature. Examples: off-by-one bugs, or changing if-statement conditions/logical operators.	94
Test Error	The test uses a source code entity incorrectly, even though the source entity's behavior was apparently correct.	50
Assertion Error	The test expects X but got Y because the assertion was written incorrectly.	38
NullPointerException	The bug causes a NullPointerException	32
Configuration Error	The bug is caused by an incorrect configuration file.	28
Dependency Error	The bug is caused by an incorrect dependency or dependency version.	14
Identifier Error	The bug is caused by an unresolvable identifier.	9
Visibility Error	The bug is caused by incorrect access modifiers on any entity.	7
Casting Error	The bug is caused by an invalid cast.	2
Resource leak	The bug is caused by a failure to close or release a resource (file, stream, etc.).	1

```
HttpRequest setupRequest(URI uri, HttpMthd httpMthd, byte[] body, ...) {
  ByteBuf content = (body==null)? Unpld.EMPTY_BUFFER:Unpld.cpdBuff(body);
  HttpRequest request = new DefaultFullHttpRq(HttpVersion.HTTP_1_1, httpMthd, getPathWithQuery(uri), content);
  - HttpUtil.setContentLength(request, body.length);
  + int length = body==null?0:body.length;
  + HttpUtil.setContentLength(request, length);
...
}
```

(a) GitHub diff.

```
<BugInstance rank="6" abbrev="NP" category="CORRECTNESS" priority="1" type="NP_NULL_ON_SOME_PATH" >
  <Method classname="org.yamcs.api.rest.HttpClient" name="setupRequest">
    <SourceLine role="SOURCE_LINE_DEREF" start="4" end="4" sourcefile="HttpClient.java">
    <SourceLine role="SOURCE_LINE_KNOWN_NULL" start="2" end="2" sourcefile="HttpClient.java">
```

(b) SpotBugs XML report.

Figure 2: GitHub Diff and SpotBugs XML report for an artifact that SpotBugs found.

```
-public class UppercaseSuffixes__CheckTest {
+public class UppercaseSuffixesCheckTest {
...
}
```

(a) Github Diff.

```
public void visitNode(Tree tree) {
  MethodTree mthd = (MethodTree) tree;
  boolean isPblc = mthd.mdfrs().contains(Modifier.PUBLIC);
  if (hasSemantic()) {
    isPublic = method.symbol().isPublic();
  }
...
}
```

(b) Source code of method flagged by ErrorProne.

```
.../Transactional MethodVisibilityCheck.java:[3,51] [CollectionIncompatibleType]
Argument 'Modifier.PUBLIC' should not be passed to this method;
its type Modifier is not compatible with its collection's type argument ModifierTree.
```

(c) ErrorProne bug warning.

Figure 3: Source code of an overlap of buggy code and a potential bug ErrorProne warns about.

The manual classification yielded ten different categories as shown in Table 2. The classification was done by looking at the GitHub diff, Travis-CI build log, the commit message, the source code, and the tool's documentation (bug patterns). We can see that the top category is Logic and the following two are related to test code. This latter can not be found by tools as it requires domain knowledge about the project. Since no tool can, or claims to, find all bugs we need too account for this. This fact highlights the need for the classification as it is an integral part of the evaluation of any static analyzer tool. The categories in the classification that each tool can find bugs in are Logic Error, Casting Error, and Resource Leak. Note that the NullPointerException bug category can be found by both SpotBugs and Infer.

How many bugs were detected by the tools? We ran the analyzers on all 320 pairs of buggy/fixed programs. We found that 59 for

Table 3: Results

Description	ErrorProne	SpotBugs	Infer
Bug Candidates	97	129	129
Total Compilation Issues	27 (47)	33 (88)	17 (86)
Total Tool Crashes	0 (12)	0	0
Bug Reports	35	60	80
Bug Candidates Found	0	1	0

```
1public void getByAlpha2(){
2  Country country =
3  CountryService.getInstance().getByAlpha("CO");
4  Assert.assertNotNull(country);
5  Assert.assertEquals("CO",country.getAlpha2Code());
6}
7...
8public Country getByAlpha(String alpha) {
9  int alphaLength = alpha.length();
10  for(Country country : countries) {
11    if (alphaLength == 2) {
12      if (country.getAlpha2Code().
13            equals(alpha.toLowerCase())) {
14        return country;
15      }
16    ...
17  }
18}
```

(a) Source code of methods with NP bug.

```
1{"bug_class":"PROVER",
2"kind":"ERROR",
3"bug_type":"NULL_DEREFERENCE",
4"qualifier":"object `country` last assigned on line
      2 could be null, dereferenced at line 5.",
5"severity":"HIGH",
6"procedure":"void CountryServiceTest.getByAlpha2()",
7"line":5,
8...}
```

(b) Infer report snippet of a NP bug.

Figure 4: Source code of an overlap of buggy code and a potential bug Infer warns about.

```
1protected Swagger read(Class<?> cls, ...) {
2  Api api = (Api) cls.getAnnotation(Api.class);
3  ...
4  if (api == null || readable) {
5  ...
6  }
7+ if( api != null ) {
8  Extension[] extensions = api.infoExtensions();
9  }
10}
```

Figure 5: GitHub diff of a bug SpotBugs did not find.

ErrorProne, 88 for SpotBugs, and 86 for Infer did not run because of either a compilation issue or a tool crash. These numbers are shown within parenthesis in Table 3. We then focused on examining the reports produced for the 97, 129, and 129 pairs of buggy and fixed programs within the selected bug categories. For this subset, 27, 33, and 17 programs resulted in a compilation error. Table 3 shows the results for this subset. ErrorProne reported potential bugs in 35 out of 97 buggy programs. However, the reports did not include the bugs under study. SpotBugs reported potential bugs in 60 out of 129 buggy programs, but only one of the bugs matched the actual bug (a null pointer dereference). Infer reported potential bugs in 80 of the 129 with zero being the actual bug.

The reports for the buggy and fixed versions were identical in all but 4, 1, 218 instances for ErrorProne, SpotBugs, and Infer, respectively. This means that the bug fix did not make a difference in the bugs reported by the tools before and after the fix. For the rest of the reports, the bug for the fixed program included a new bug apparently introduced by the fix. Infer seems to be more sensitive than the other tools in this respect.

The bug that SpotBugs found is shown in Figure 2. The SpotBugs report is shown in Figure 2b. It reports that there is a known null value at line 2, which is dereferenced on line 4. The GitHub diff of the fix is shown in Figure 2a. We observe on line 2 that the function checks body for null. Then on line 4 the variable body is dereferenced to get its length. This causes a null pointer exception, which SpotBugs correctly detected. The fix adds another null check for body, and sets `length` to 0 if the body is null.

An example of a reported potential bug found by ErrorProne and the line in question are shown in Figure 3. ErrorProne warns of a bug about an incorrect argument type being passed to a method in Figure 3c. The source code is shown in Figure 3b. The actual bug that caused this build to fail and its fix are shown in Figure 3a. The problem is an incorrect use of an identifer, which does not exactly match the bug reported.

An example of a false positive reported by Infer is shown in Figure 4. The report in Figure 4b refers to line 2 in the source code shown in Figure 4a. More specifically, the report warns about an object, `country`, that is being assigned a value from a return value from a method could be null. Infer asserts that `country` is dereferenced on line 5, which can cause a null dereference. This actually cannot happen because on line 4 there is an assertion that `country` is not null, so this path is not feasible.

An example of a bug SpotBugs did not find is shown in Figure 5. The object that is being dereferenced is `api`, which is assigned a value from the return value of a method. However, we can observe that there is a null check on line 4, and `api` is never assigned another value later on, so it is still null. The null dereference is reported on line 8, where `api` is being dereferenced. So, there is a path where `api` could be null and is being dereferenced, which causes the null pointer exception to occur for the artifact causing the build to fail. The fix is added on line 7 where a null check is added for the variable `api` before the dereference.

5 THREATS TO VALIDITY

Our study cannot be generalized to all tools or bug types. We chose a random sample from a dataset that has bugs and their fixes from real open source project, the size of our sample in not large enough to make a claim about these tools beyond our sample. Our methodology required extensive manual inspection, which could be error prone. We tried to minimize this by having multiple people involved with the classification and giving the tools the benefit of the doubt

in cases of uncertainty. The tools reported warnings about many other unknown bugs that could possibly be true, but this study focused on specific bugs to measure effectiveness. We did not investigate the nature of the additional bugs as they did not fall within the effectiveness methodology we described in the approach.

6 CONCLUSIONS AND FUTURE WORK

We ran 320 from the BugSwarm dataset on ErrorProne, SpotBugs, and Infer. To measure the effectiveness of the static analyzers, we determined whether the tools can find the exact bug that caused the build to fail. Since each tool finds certain kinds of bugs we performed a classification of our data to only consider bugs that fall within such categories. After manual inspection of the tool reports we found that only SpotBugs found one of the bugs, while ErrorProne and Infer failed to find any of the bugs.

There are many avenues to extend this study. The first is that there are many other bug-finding tools (e.g., lgtm [7]) that could be included in this study. Second, in the future we would like to look in more depth on why the tools failed to find bugs in a given criteria. Thirdly, we are have focused on whether the tools find the specific bugs in our dataset. An interesting future avenue would be to see how many of the additional reported bugs from each tool are actually unknown true bugs. Lastly, BugSwarm is continuously growing, thus we expect to have access to an order of magnitude more bugs in the near future that we could use to expand this study in the future.

ACKNOWLEDGMENTS

Special thanks to Naji Dmeiri and Yichen Wang for their contribution to the classification of BugSwarm bugs and Cindy Rubio González for guidance of this study. This work was supported in part by NSF grants CNS-1629976 and CCF-1464439, and a Microsoft Azure Award.

REFERENCES

[1] 2018. Error Prone. https://github.com/google/error-prone. (2018).
[2] 2018. Error Prone Bug Patterns. https://errorprone.info/bugpatterns. (2018).
[3] 2018. Find Bugs. http://findbugs.sourceforge.net/. (2018).
[4] 2018. Infer. http://fbinfer.com/. (2018).
[5] 2018. Infer Bug Descriptions. https://fbinfer.com/docs/checkers-bug-types.html. (2018).
[6] 2018. SpotBugs Bug Descriptions. https://spotbugs.readthedocs.io/en/latest/bugDescriptions.html. (2018).
[7] 2019. lgtm. https://lgtm.com/. (2019).
[8] 2019. PMD. https://pmd.github.io/. (2019).
[9] 2019. Spot Bugs. https://spotbugs.github.io/. (2019).
[10] N. Ayewah, D. Hovemeyer, J. D. Morgenthaler, J. Penix, and W. Pugh. 2008. Using Static Analysis to Find Bugs. *IEEE Software* 25, 5 (Sept 2008), 22–29. https://doi.org/10.1109/MS.2008.130
[11] Naji Dmeiri, David A. Tomassi, Yichen Wang, Antara Bhowmick, Yen-Chuan Liu, Premkumar Devanbu, Bogdan Vasilescu, and Cindy Rubio-González. 2019. BugSwarm: Mining and Continuously Growing a Dataset of Reproducible Failures and Fixes. (2019). To appear in Proceedings of the 41st ACM/IEEE International Conference on Software Engineering ICSE 2019, Montreal, Canada, May 25 - 31, 2019.
[12] Andrew Habib and Michael Pradel. 2018. How many of all bugs do we find? a study of static bug detectors. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE 2018, Montpellier, France, September 3-7, 2018*. 317–328. https://doi.org/10.1145/3238147.3238213
[13] René Just, Darioush Jalali, and Michael D. Ernst. 2014. Defects4J: A Database of Existing Faults to Enable Controlled Testing Studies for Java Programs. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis (ISSTA 2014)*. ACM, New York, NY, USA, 437–440. https://doi.org/10.1145/2610384.2628055
[14] Mangala Gowri Nanda, Monika Gupta, Saurabh Sinha, Satish Chandra, David Schmidt, and Pradeep Balachandran. 2010. Making defect-finding tools work for you. In *2010 ACM/IEEE 32nd International Conference on Software Engineering*, Vol. 2. IEEE, 99–108.
[15] Nick Rutar, Christian B. Almazan, and Jeffrey S. Foster. 2004. A Comparison of Bug Finding Tools for Java. In *Proceedings of the 15th International Symposium on Software Reliability Engineering (ISSRE '04)*. IEEE Computer Society, Washington, DC, USA, 245–256. https://doi.org/10.1109/ISSRE.2004.1
[16] 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. *Automated Software Engg.* 22, 4 (Dec. 2015), 561–602. https://doi.org/10.1007/s10515-014-0169-8
[17] Stefan Wagner, Jan Jürjens, Claudia Koller, and Peter Trischberger. 2005. Comparing Bug Finding Tools with Reviews and Tests. In *Testing of Communicating Systems*, Ferhat Khendek and Rachida Dssouli (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 40–55.