On the Real-World Effectiveness of Static Bug Detectors at Finding Null Pointer Exceptions

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

Cindy Rubio-González
University of California, Davis
United States of America
crubio@ucdavis.edu

***Abstract*—Static bug detectors aim at helping developers to automatically find and prevent bugs. In this experience paper, we study the effectiveness of static bug detectors at identifying Null Pointer Dereferences or Null Pointer Exceptions (NPEs). NPEs pervade all programming domains from systems to web development. Specifically, our study measures the effectiveness of five Java static bug detectors: CheckerFramework, ERADICATE, INFER, NULLAWAY, and SPOTBUGS. We conduct our study on 102 real-world and reproducible NPEs from 42 open-source projects found in the BUGSWARM and DEFECTS4J datasets. We apply two known methods to determine whether a bug is found by a given tool, and introduce two new methods that leverage stack trace and code coverage information. Additionally, we provide a categorization of the tool's capabilities and the bug characteristics to better understand the strengths and weaknesses of the tools. Overall, the tools under study only find 30 out of 102 bugs (29.4%), with the majority found by ERADICATE. Based on our observations, we identify and discuss opportunities to make the tools more effective and useful.**

***Index Terms*—static bug detectors, null pointer exceptions, null pointer dereferences, bug finding, BugSwarm, Defects4J, Java**

I. INTRODUCTION

Defects in software are a common and troublesome fact of programming. Software defects can cause programs to crash, lose or corrupt data, suffer from security vulnerabilities, among other problems. Depending on the application domain, undesirable behavior can range from poor user experience to more severe consequences in mission critical applications [44]. Testing to uncover such software defects remains one of the most expensive tasks in the software development cycle [31].

There is a need for both precision and scalability when finding defects in real-world code. Furthermore, in an effort to increase their applicability, static bug detectors are often designed to target a large variety of software bugs. Many static bug detectors [2, 5, 7, 9, 10, 13–16] are currently being developed in industry and academia. Even with many tools to choose from, developers have some hesitation in using static bug detectors for a variety of reasons such as large number of bug warnings, high false positive rates, and inadequate warning messages [18, 26].

Previous studies have evaluated static bug detectors through various metrics: number of warnings [35], number of false negatives [38], tool performance [35], and recall [20, 41]. These studies have focused on popular tools that identify a large number of bug patterns, and their conclusions are drawn with respect to the overall bug-finding capabilities of the tools. In contrast, this paper evaluates static bug detectors with respect to their effectiveness at finding a common and serious kind of bug: Null Pointer Dereferences or Null Pointer Exceptions (NPEs).

NPEs pervade all programming domains from systems software to web development. For instance, as of August 2021, there are over 1,900 CVEs (Common Vulnerabilities and Exposures) that involve NPEs [3]. One such CVE describes a denial of service attack in early versions of Java (1.3 and 1.4) caused by crashing the Java Virtual Machine when calling a function with a `null` parameter [1]. In general, NPEs are problematic in memory-unsafe and object-oriented languages. NPEs occur when either a pointer to a memory location or an object is dereferenced while being uninitialized or explicitly set to `null`. Depending on the programming language, NPEs will result in either undefined behavior or a runtime exception.

This experience paper evaluates recall of static bug detectors with respect to a known set of *real* NPE bugs. The focus on NPEs allows to present an in-depth study of different approaches to find a same kind of bug, the characteristics of real-world NPEs, and the reasons that affect tool effectiveness. To the best of our knowledge, this is the *first study on the real-world effectiveness of static bugs detectors at finding NPEs.*

There are two orthogonal approaches to finding or preventing NPEs, which make use of either a static bug detector or a type-based null safety checker. The former uses dataflow analysis [6, 10, 23, 29, 30, 32, 34] to *find* null dereferences. Such approaches mainly differ on the complexity of their analyses. Some favor analysis scalability at the expense of missing real bugs and/or producing numerous false positives, e.g., intra/interprocedural and field sensitivity. The latter *prevents* NPEs via a type system with null-related information using dataflow analysis for type refinement. The type checker approach has been adopted in recent years [4, 14, 19, 33].

We study two popular Java static bug detectors: INFER [6, 15–17] and SPOTBUGS [10], and three popular type-based null safety checkers for Java: Checker Framework's Nullness Checker (CFNULLNESS) [19, 33], ERADICATE [4], and NULLAWAY [8, 14]. INFER uses separation logic and bi-abduction analysis [16] to infer pre/post conditions from procedures affecting memory. SPOTBUGS detects bugs based on a predefined set of bug patterns. CFNULLNESS verifies

2643-1572/21/$31.00 ©2021 IEEE
DOI 10.1109/ASE51524.2021.00035

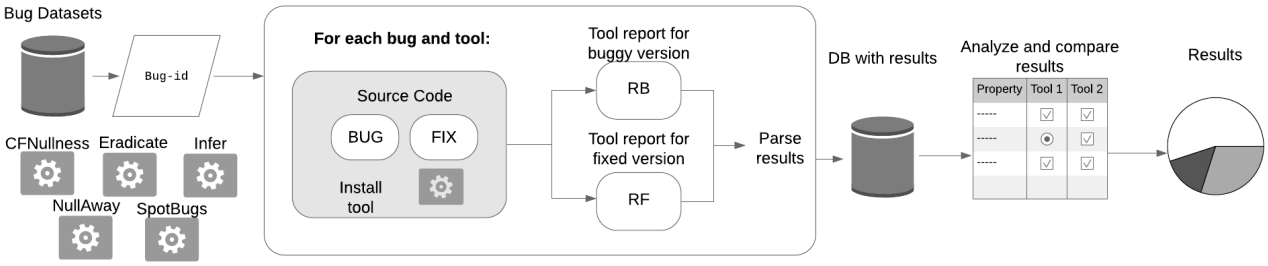

Fig. 1: Workflow for running tools, collecting reports, parsing results, and analyzing data.

the absence of NPEs via type checking nullable expression dereferences and assignments. ERADICATE is a type checker that performs flow-sensitive analysis to find possible null dereferences. Finally, NULLAWAY uses dataflow analysis to type check nullability in procedures and class fields.

In this study, we consider 102 *real-world* and *reproducible* NPEs found across 42 popular open-source Java projects. 76 of these NPEs belong to the BUGSWARM dataset [42] while the remaining 26 are from DEFECTS4J [27]. For each NPE, both datasets provide buggy and fixed versions of the programs along with scripts for compilation and testing. Furthermore, each program has a failing test due to an NPE. This makes both the BUGSWARM and DEFECTS4J datasets good candidates for this study; we want to run existing static bug detectors and type checkers on these programs to determine their effectiveness at detecting and preventing real NPEs.

The first challenge is to determine whether a tool finds or prevents a specific NPE bug. Tools may report the program location at which the null dereference occurs, or simply the location where the `null` value originates, which can be far from the dereference. The latter is particularly difficult to associate with the bug fix, which is often applied closer to the dereference site. Another difficulty lies in the large number of warnings to inspect. On average a tool produces from 122 to 1,307 bug warnings per program (in our dataset).

Previous work has partially automated the process of mapping bugs to warnings based on *static* information such as the code difference (diff) between buggy and fixed versions [20, 38], and by comparing the warnings produced for each version of the program [20]. In this paper, we observe that *dynamic* information can also be leveraged when an input exposing the NPE bug is available, which is the case for all the bugs in our dataset. We present two new mapping methods for NPEs that use (1) stack trace information, and (2) code coverage of tests that fail due to NPEs. Our experimental evaluation shows that these methods complement previous approaches.

We run CFNULLNESS, ERADICATE, INFER, NULLAWAY, and SPOTBUGS on our dataset of 102 real NPEs. We find that the tools produce a large number of warnings, including over 500,000 NPE warnings across all programs. We apply existing approaches, and our new methods, to identify the warnings that describe the bugs under study. Ultimately, we find that the tools detect only 30 out of 102 bugs (29.4%), with ERADICATE finding the majority of these.

The second challenge is to understand the reasons *why* tools fail to find NPEs to identify opportunities to improve their real-world effectiveness. This requires understanding the capabilities of the tools under study as well as the characteristics of the NPE bugs in our dataset. First, we conduct a detailed analysis of the tools' capabilities with respect to well-known program-analysis properties (e.g., flow sensitivity, context sensitivity, etc.), and we identify common sources of unsoundness. This process required us to manually inspect the source code of the tools and write tests. All of our findings were later confirmed by tool developers. Second, we manually inspect and categorize each NPE bug in the dataset with respect to the nature of the dereference and its context. Based on the tool results, and the tool and bug characterizations, we identify several open opportunities to improve static bug detectors that find NPEs.

The contributions of this paper are:

- We present two new methods that leverage dynamic information to map tool warnings to NPE bugs (Section II).
- We provide a categorization of the tool's capabilities and the bug characteristics to better understand the strengths and weaknesses of the tools under study (Section III).
- We evaluate CFNULLNESS, ERADICATE, INFER, NULLAWAY, and SPOTBUGS on a collection of 102 NPEs, from which only 29.4% of NPE bugs are detected (Section IV).
- We discuss the capabilities and limitations of each tool, and provide future directions for improving their real-world effectiveness (Section V).

II. METHODOLOGY

Here we describe the benchmark and tool selection, and the methodology to determine the effectiveness of the tools at finding NPEs. Figure 1 shows the main steps of our approach.

A. Benchmark Selection

Our study focuses on Null Pointer Exceptions (NPEs). We consider bugs from the BUGSWARM and DEFECTS4J datasets, both of which provide a bug classification based on runtime exceptions. Our selection criteria is: (1) the bug is due to an NPE, (2) there is a failing test due to the NPE, and (3) code coverage can be measured. Additionally, we control for unique builds when selecting BUGSWARM bugs. Our final dataset

consists of 76 NPE bugs from the BUGSWARM dataset and 26 from DEFECTS4J. The BUGSWARM NPE bugs belong to 32 Java projects hosted on GitHub that use the Maven build system, while the DEFECTS4J bugs belong to 10 Java projects that use the Ant build system. Note that all NPEs are reproducible, i.e., one can run the programs and observe a Null Pointer Exception being thrown. Furthermore, *we manually verified that each NPE bug in our study is an actual NPE*, i.e., a null object is eventually dereferenced. Each NPE instance consists of the source code that contains the bug, the source code that fixes the bug, and scripts to compile and test.

B. Tool Selection and Configuration

We conducted an extensive search for tools that find or prevent NPE bugs in Java projects. We focused on publicly available tools that are standalone and under active development. Out of nine tools, four [29, 30, 32, 34] did not satisfy at least one of these requirements. In this paper we study the remaining five tools: CFNULLNESS, ERADICATE, INFER, NULLAWAY, and SPOTBUGS. Note that INFER and SPOTBUGS find a large variety of bugs in addition to NPE bugs. CFNULLNESS, ERADICATE, and NULLAWAY exclusively specialize in NPEs. Below we describe each tool.

a) CFNULLNESS*:* A type checker written using the Checker Framework, which is available as a compiler plugin. CFNULLNESS works with nullness type annotations, `@Nullable` and `@NotNull`, and looks for violations in their use. Namely looking for dereferences on `@Nullable` expressions and for `@Nullable`-value assignments to `@NotNull` variables. CFNULLNESS produces compile-time warnings. We run CFNULLNESS using its default configuration.

b) ERADICATE*:* A type checker part of the INFER static-analysis suite. ERADICATE type checks for `@Nullable` annotations in Java programs by performing a flow-sensitive analysis to propagate null-related information through assignments and calls. ERADICATE produces warnings for accesses that could lead to an NPE. ERADICATE produces a report in JSON format that provides the stack trace, severity, and source location associated with each bug detected. We run ERADICATE using its default configuration.

c) INFER*:* A static-analysis tool developed by Facebook that finds a variety of bugs in Java, C/C++, and Objective-C programs. INFER uses bi-abduction analysis to find bugs including deadlocks, memory leaks, and null pointer dereferences. Similar to ERADICATE, INFER produces a report in JSON format that provides the stack trace, severity, and bug location. We use INFER's default setting, which runs the bi-abduction analysis.

d) NULLAWAY*:* A type checker for Java developed by Uber that applies various AST-based checkers to find NPE bugs. NULLAWAY is available as a plugin for Maven and Gradle. We use NULLAWAY's default configuration, which assumes that unannotated method parameters, return values, and class fields are *not* null. In such cases, the tool produces a warning when it is found that any of those locations could hold a `null` value. The user can add explicit `@Nullable` annotations to obtain more precise results.

e) SPOTBUGS*:* SPOTBUGS applies pattern matching and limited dataflow analysis to find a large variety of bugs such as infinite recursion, integer overflows, and null pointer dereferences. The tool produces an XML report listing bug warnings that include class name, method name, severity, and line numbers associated with the identified bug. SPOTBUGS is available as a plugin for a variety of build systems such as Ant, Gradle, and Maven. We run SPOTBUGS with effort level "max", which indicates that SPOTBUGS performs its interprocedural analysis. Also, we use two different error confidence threshold settings "low" and "high" ("low" confidence threshold may report a higher number of false positives).

C. Analysis of NPE Warnings

A challenge in this study is to determine whether a tool finds or prevents a specific NPE bug. In the case of NPEs, tools may report the program location at which the null dereference occurs, or simply report the location where the null value originates, which can be far from the dereference. The latter is particularly difficult to associate with the bug fix, which is often applied closer to the dereference site.

We consider four approaches for mapping bug warnings to actual bugs in the source code, i.e., determine whether a tool finds a given bug under study. Two of these approaches have been used in previous work: the CODE DIFF METHOD [20, 38] and the REPORT DIFF METHOD [20]. We explore two new approaches, which we refer to as the STACK TRACE METHOD and the COVERAGE METHOD.

Figure 2 shows an example of an NPE found in the OpenPnP[1] GitHub project as part of the BugSwarm dataset.[2] Method `saveDebugImage` is called on Line 8 of file `OpenCvVisionProvider.java` (see Figure 2b), where argument `debugMat` is `null`. Method `saveDebugImage` in file `OpenCvUtils.java` calls `toBufferedImage` on Line 6 (see Figure 2a), passing in `null`, which is then dereferenced on Line 11. The code highlighted (in green) represents the patch to fix the NPE. Figure 2e shows the stack trace, and Figures 2c and 2d show the warnings produced by SPOTBUGS and INFER, respectively.

1) CODE DIFF METHOD*:* This method takes as input the set of warnings reported for the buggy program and the set of patches from the GitHub code diff.[3] The analysis focuses on NPE bug warnings, and checks whether the source location of these warnings overlaps with the lines changed in the patches. However, this is based on an over-approximation; the lowest and highest line numbers associated with the patch in each changed file are considered.[4] If an overlapping line is found, then the warning is considered a *bug candidate*. We manually examine bug candidates to verify their validity.

[1]https://github.com/openpnp/openpnp

[2]BugSwarm image tag: `openpnp-openpnp-213669200`.

[3]A GitHub code diff may consist of several patch fragments.

[4]Previous work has also added a configurable number of lines before the starting point and after the ending point of the line range [20].

```
     @@ -2,6 +2,9 @@ public synchronized static Mat ...
2    2
3    3   }
4    4
5    5   public static void saveDebugImage(..., Mat mat) {
     6   + if (mat == null) {
     7   +     return;
     8   + }
6    9     saveDebugImage(...,OpenCvUtils.toBufferedImage(mat));
7    10    ...
8    11  }
9    12
10   13  public static BufferedImage toBufferedImage(Mat m) {
11   14    if (m.type() == CvType.CV_8UCI) {...} // NPE!
12   15  }
```

(a) GitHub diff in file `OpenCvUtils.java`.

```
1  public getTemplateMatches(BufferedImage template) {
2    ...
3    Mat debugMat = null;
4    if (LogUtils.isDebugEnabled()) {
5      debugMat = imageMat.clone();
6    }
7    ...
8    OpenCvUtils.saveDebugImage(..., debugMat);
9  }
```

(b) Null origin in file `OpenCvVisionProvider.java`.

```
<BugInstance rank="8" abbrev="NP" category="
    CORRECTNESS" priority="2" type="
    NP_NULL_PARAM_DEREF">
  <Method classname="org.openpnp.util.
    OpenCvUtils" name="saveDebugImage">
    <SourceLine classname="org.openpnp.util.
    OpenCvUtils" start="6" end="6"
    sourcefile="OpenCvUtils.java"/>
```

(c) SpotBugs XML report.

```
{"bug_class":"PROVER",
"kind":"ERROR",
"bug_type":"NULL_DEREFERENCE",
"qualifier":"object `debugMat` last assigned
     on line 3 could be null and is
    dereferenced by call to `saveDebugImage
    (...)` at line 8.",
"file":"OpenCvVisionProvider.java",
"severity":"HIGH",
 ...
}
```

(d) Infer JSON report.

```
java.lang.NullPointerException
  at org.openpnp.util.OpenCvUtils.toBufferedImage(OpenCvUtils.java:11)
  at org.openpnp.util.OpenCvUtils.saveDebugImage(OpenCvUtils.java:6)
  at org.openpnp.machine.reference.vision.OpenCvVisionProvider.getTemplateMatches(
    OpenCvVisionProvider.java:8)
  ...
```

(e) Stack trace for buggy program.

Fig. 2: GitHub diff, stack trace, SpotBugs XML report, and Infer JSON report for an NPE found by SpotBugsLT and Infer.

Consider the patch in Figure 2a. The line at the top (starting with @@) indicates that the patch includes original lines 2 through 6, and new lines 2 through 9 from file `OpenCvUtils.java`. Therefore, the approximated line range is 2 through 9 for the buggy program, i.e., the program before the fix. The SpotBugs report (see Figure 2c) includes the XML tag *SourceLine*: Line 6 of file `OpenCvUtils.java`. This location lies within the line range 2–9, thus the method correctly collects this warning as a bug candidate. On the other hand, even though Infer (see Figure 2d) successfully finds the bug, the Code Diff Method approach fails to map the warning because the report does not include lines close to the fix. In this case, using code diff information is not effective.

2) Report Diff Method*:* This method uses the set of bug warnings of the buggy program, and the set of warnings of its fixed version. The algorithm searches for NPE bug warnings that are *only* reported for the buggy program. The intuition is that the warning that describes the bug of interest should not be present in the bug report of the fixed program. Using this method, both SpotBugs and Infer are determined to have found the bug from Figure 2. This method is convenient because it only requires two bug reports. However, the absence of a bug warning in the fixed program does not necessarily mean that the bug of interest was found. The code change could have introduced "noise" that leads the tool to conclude that an *unrelated* bug warning is no longer a problem. We observe that this occurs often in practice (see Section IV-B).

3) Stack Trace Method*:* This approach requires the set of bug warnings of the buggy program, and the stack trace(s) produced when running the buggy program. As with previous methods, this approach only considers warnings related to NPE bugs. For each NPE warning, the algorithm retrieves the file and line number(s) associated with the warning, and checks whether those are included in the stack trace. If so, the warning is classified as a bug candidate.

Consider again the example from Figure 2. The SpotBugs report (Figure 2c) mentions Line 6 in file `OpenCVUtils.java`. The report pinpoints that there is a null parameter in a recursive call to `saveDebugImage`, which could result in an NPE. On the other hand, the Infer report (Figure 2d) lists a warning associated with file `OpenCvVisionProvider.java` on Line 8. The call to `saveDebugImage` in method `getTemplateMatches` is passed `debugMat` as argument, which could be `null` and result in an NPE. Note that Infer refers to a lower stack frame than SpotBugs, but the Stack Trace Method successfully maps both reports to the same bug because both

locations can be found in the stack trace (Figure 2e).

The STACK TRACE METHOD takes advantage of the nature of NPE bugs and their presence in the stack trace. Because NPE bugs correspond to Null Pointer Exceptions, the call stack is given at the time the exception occurs. This information is a valuable resource that leads to a more natural bug mapping than previous methods. However, this method requires an executable buggy program and a reproducible NPE. Also, this method provides a line in the stack trace that can be mapped to a bug warning, however, this does not necessarily mean that the tool found the correct dereference; there are long dereference chains that may be associated with the same line. Thus, as with previous methods, it is necessary to manually verify that the trace indeed matches the context of the NPE warning. We consider all available sources of information such as source code and code diff during manual inspection.

4) COVERAGE METHOD*:* This method is a general version of the STACK TRACE METHOD, but it includes *all* lines executed by the test that triggers the NPE. The input is the set of NPE warnings of the buggy program, and the lines covered (executed) by a test case that fails due to an NPE. The approach determines if the source location given in a warning is covered, in which case the warning is added to the set of bug candidates. This captures the scenario where the location of an NPE warning is far away from the actual dereference, which is particularly useful when analyzing the warnings produced by type checkers such as NULLAWAY and ERADICATE. The assumption is that even if the NPE warning and the actual dereference are located far away from each other, both source locations will be part of the execution trace. For example, consider a case in which a field is set to `null` in a constructor and the field is dereferenced in some method. Type checkers may produce a warning related to setting the field to `null`, but not a warning describing the dereference itself. However, in this case, both source locations will be part of the execution trace. Note that this approach requires the existence of a failing test that triggers the NPE, and the ability to execute the test. Both requirements are met for our dataset. As with other methods, we manually inspect all bug candidates to determine their validity.

III. BUG AND TOOL CHARACTERIZATION

A fundamental step in evaluating the effectiveness of static bug detectors is to understand their capabilities, and whether real-world bugs possess the desired characteristics to be detected. In this section, we characterize the dataset of real NPEs as well as the tools under study with respect to their approaches to find NPEs. We describe our methodology and results, which will be critical in Section IV to determine whether a given NPE *can* be found by the tools.

First, we performed a manual categorization of all 102 NPEs to determine the root cause of the null pointer dereferences. The categorization was performed separately by an author of this paper and two people external to the project. When in disagreement, the inspectors met to reach consensus.

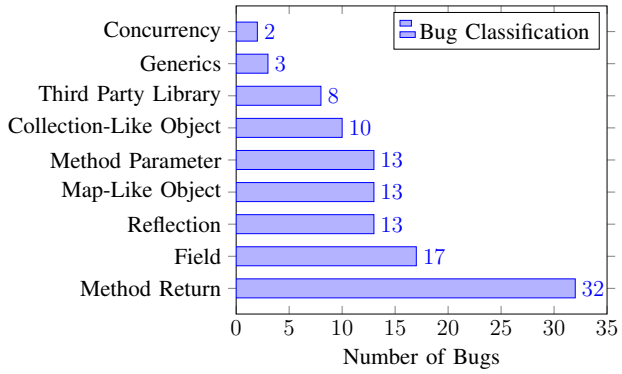

Fig. 3: Bug Classification Results

Using the source code, the GitHub code diff, and the build log, we identified the origin of the `null` value, and its dereference location. Based on this inspection, we identified nine general categories of NPEs with respect to what is dereferenced, and the context of the dereference. These categories along with their counts can be found in Figure 3. Note that an NPE can belong to multiple categories. The most common categories are when a method return value is dereferenced (32 bugs) and when a field is dereferenced (17 bugs).

As for the tool capabilities, we consider seven well-known program analysis properties: (1) intraprocedural, (2) interprocedural, (3) flow sensitive, (4) context sensitive, (5) field sensitive, (6) object sensitive, and (7) path sensitive [11].

We identified seven common sources of unsoundness: (1) handling of third-party libraries whose source code may not be available, (2) impure methods that have side effects and are non-deterministic, (3) concurrency, (4) reasoning about dynamic dispatch, (5) dealing with code that uses reflection, (6) field initialization after a constructor is called, and (7) generic parameters. Unsoundness can lead to false positives (incorrect bug warnings) and false negatives (missed bugs).

We studied CFNULLNESS, ERADICATE, INFER, NULLAWAY, and SPOTBUGS with respect to the above analysis characteristics and sources of unsoundness. In this process, we manually inspected the source code and documentation of the tools, and we wrote kernel test programs that exhibited different categories of behaviors to confirm tools' capabilities and limitations. Table I shows the tool capabilities, and Table II shows the sources of unsoundness for each tool. Below we describe our findings for each tool, which were confirmed by the corresponding developers.

a) CFNULLNESS*:* An ensemble of three checkers: (1) an intraprocedural flow-sensitive qualifier inference for the nullness of a particular object, (2) initialization checking, and (3) map key checking. It assumes `@NonNull` for unannotated code except for locals, and provides an analysis for iterating over null collections and arrays. Additionally, CFNULLNESS supports annotations to denote: (1) if a method has no side-effects or is deterministic, (2) the target of a reflection invocation, (3) and upper bounds of types for generic objects.

TABLE I: Tool Capabilities Confirmed by Developers. ✓= has capabilities, ✗= no capabilities, Partial = limited capabilities.

Tool	Intraproc.	Interproc.	Field Sensitive	Context Sensitive	Object Sensitive	Flow Sensitive	Path Sensitive
CFNULLNESS	✓	✗	✓	✗	✗	✓	✗
ERADICATE	✓	✗	Partial	✗	✗	✓	✓
INFER	✓	✓	✓	✗	✓	✓	✓
NULLAWAY	✓	Partial	Partial	✗	N/A	Partial	Partial
SPOTBUGS	✓	Partial	Partial	✗	✗	Partial	Partial

TABLE II: Sources of Unsoundness for the Tools. ✓= is sound, ✗= is unsound, Partial = unsound in some aspects.

Tool	Third Party Libs.	Impure Methods	Concurrency	Dynamic Dispatch	Reflection	Field Init.	Generic Types
CFNULLNESS	✓	✓	✓	✓	✓	✓	✓
ERADICATE	Partial	✗	✗	✗	✗	Partial	✗
INFER	Partial	Partial	✗	Partial	✗	✗	✗
NULLAWAY	✗	✗	✗	✓	✗	Partial	✗
SPOTBUGS	✗	✗	✗	✗	✗	✗	✗

b) ERADICATE*:* An intraprocedural flow-sensitive analysis for the propagation of nullability through variable assignments and function calls. ERADICATE also raises an alarm for accesses to fields that have annotated nullability, however its field initialization checker is subject to ongoing work. ERADICATE's nullability annotations allow for the annotation of methods, fields, and method parameters with `@Nullable` annotations. As detailed in Table II, ERADICATE provides built-in models of the JDK and Android SDK and supports user-specified nullability signatures for other third-party libraries, which helps mitigate false negatives.

c) INFER*:* An interprocedural analysis that supports tracking object aliasing, side effects in methods, and dynamic types of objects. All our tests were successful when running INFER, showing that the tool is interprocedural and field sensitive. A caveat is that INFER does not find uninitialized fields, but it can find null dereferences to fields that have been initialized. As shown in Table II, INFER partially supports third-party libraries via an internal model of the JDK. For impure methods, INFER tracks some effects in methods, e.g., if a method sets `this.field = null`, the effect will be tracked at the call site. Tracking dynamic types of objects is useful to refine the control-flow graph. However, this only occurs in the context of the entry point of the analysis.

d) NULLAWAY*:* A flow-sensitive type refinement analysis to infer nullness of local variables that includes a field initialization checker. NULLAWAY assumes that unannotated code *cannot* be null. For methods, fields, and method parameters annotated with the `@Nullable` annotation, NULLAWAY ensures no dereferences, and that their value will not be assigned to a non-null field or argument. Our tests showed that NULLAWAY finds local and object field dereferences without annotations. With annotations, NULLAWAY can find null dereferences of method parameters and return values. NULLAWAY is able to avoid dynamic dispatch as a source of unsoundness by ensuring that methods that are overridden have the same nullability as its parent's class. NULLAWAY's field initialization is unsound. For example, the analysis does not check fields that are read by methods called from constructors.

e) SPOTBUGS*:* A null-pointer analysis inherited from FINDBUGS [24] that combines forward and backward dataflow analyses for tracking `null` values. The analysis provides limited tracking of object fields; it does not support aliasing and volatile fields, and it assumes that any method can modify a field of an object passed as argument. Additionally, SPOTBUGS provides a null-related annotation `@CheckForNull` to denote values that must be null-checked prior to a dereference. Our tests confirmed the intraprocedural nature of SPOTBUGS, however we were unable to expose SPOTBUGS' field sensitivity. Lastly, SPOTBUGS infers parameter and return value information intraprocedurally if these are null checked, and it suffers from all sources of unsoundness as shown in Table II.

IV. EXPERIMENTAL EVALUATION

This experimental evaluation is designed to answer the following research questions:

RQ1 How prevalent are NPEs among all warnings?
RQ2 How effective are bug mapping methods for NPEs?
RQ3 How effective are static bug detectors for NPEs?
RQ4 What are the reasons bug detectors miss NPEs?

We ran CFNULLNESS, ERADICATE, INFER, NULLAWAY, and SPOTBUGS on our dataset of 102 programs with real NPE bugs to generate bug reports for the buggy and fixed versions of the programs. We ran the tools on the full programs, and verified that the files relevant to the bug and fix were indeed analyzed by the tools. We considered two settings for SPOTBUGS: low and high thresholds. The results are presented as SPOTBUGSLT and SPOTBUGSHT, respectively. We automatically parsed the bug reports to extract and normalize relevant information, which we stored in a MySQL database.

Our study is fully reproducible. The dataset of real reproducible NPE bugs from BUGSWARM and DEFECTS4J is publicly available as well as the tools we study. The scripts for performing the experiments and all data described in this section is publicly accessible.[5]

[5]https://github.com/ucd-plse/Static-Bug-Detectors-ASE-Artifact

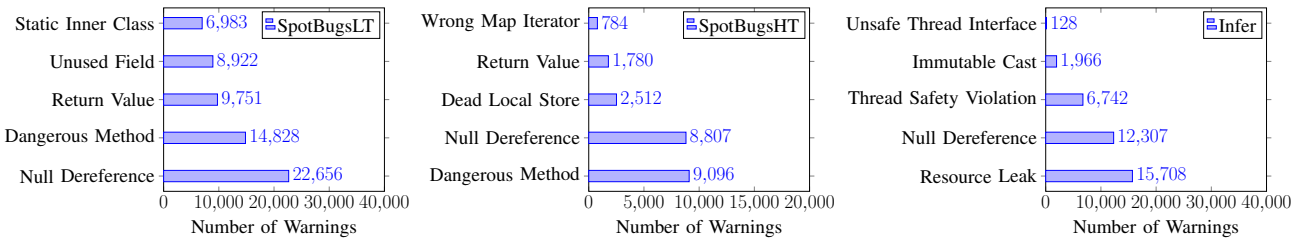

Fig. 4: SPOTBUGSLT, SPOTBUGSHT, and INFER distribution of top 5 warnings.

TABLE III: Number of *all* warnings and NPE warnings produced by each tool."Avg All" and "Avg NPEs" refer to the average number of warnings produced *per program*.

Tool	All	NPEs	Avg All	Avg NPEs
CFNULLNESS	231,860	231,860 (100%)	1,137	1,137
ERADICATE	266,682	266,682 (100%)	1,307	1,307
INFER	37,035	12,307 (33.2%)	181	60
NULLAWAY	25,065	25,065 (100%)	122	122
SPOTBUGSHT	49,555	8,807 (17.8%)	243	43
SPOTBUGSLT	129,183	22,656 (17.5%)	633	111

A. RQ1: Prevalence of NPE Warnings

Table III shows the total and average number of warnings produced by each tool when analyzing the programs. There are a total of 739,380 warnings across the 102×2 programs in our dataset. ERADICATE yields the largest number of warnings with 266,682, all of which are NPE warnings. Similarly, CFNULLNESS has the second highest number of NPE warnings with a total of 231,860. SPOTBUGSLT produces the third highest number of warnings with 129,183 warnings, and SPOTBUGSHT follows with 49,555 warnings. Unlike ERADICATE, CFNULLNESS and NULLAWAY, SPOTBUGS can generate over a hundred different types of non-NPE warnings while INFER generates seven.

Figure 4 shows the top five types of warnings for SPOTBUGSLT, SPOTBUGSHT, and INFER. It is observed that NPEs are one of the most prevalent warnings for these tools: the most common for SPOTBUGSLT, and the second most common for both SPOTBUGSHT and INFER. Indeed, NPEs constitute from 17.5% to 33.2% of the total warnings produced by these tools. For SPOTBUGSHT, we observe a reduction in total number of warnings and NPE warnings with respect to SPOTBUGSLT of 61.6% and 61.1%, respectively.

Finally, NULLAWAY produces the fewest warnings (all of them are NPE warnings), with a total of 25,065.

RQ1: NPE warnings are prevalent in all the tools studied. A total of 567,377 NPE warnings (76.7% of all warnings) are produced for our dataset. The percentage of NPE warnings for SPOTBUGSLT is 17.5%, SPOTBUGSHT 17.8%, and INFER 33.2%.

TABLE IV: Bugs mapped by each method. We show the number of correct mappings / the total number of mappings. Column "Bugs Found" gives the total number of bugs found per tool. Inside parenthesis are the number of bugs that a tool found but not others. 30 unique bugs are found across tools.

	Method				
Tool	Code	Report	Stack	Covered	Bugs Found
CFNULLNESS	6/56	5/18	7/26	5/56	11 (2)
ERADICATE	10/51	7/24	7/22	8/52	20 (5)
INFER	3/23	2/13	9/12	7/27	10 (1)
NULLAWAY	1/17	0/21	1/4	5/26	5 (2)
SPOTBUGSHT	4/18	4/6	1/5	2/13	4 (0)
SPOTBUGSLT	6/46	6/13	6/12	3/26	9 (1)
Total	30/211	24/95	31/81	30/200	30 Unique

B. RQ2: Effectiveness of Bug Mapping Methods

We applied the four methods discussed in Section II-C to find whether the tool warnings describe the bugs of interest. In total, all methods together correctly find 30 distinct bugs out of 102 bugs (29.4%). All bug candidates were manually inspected. Table IV summarizes the results.

An effective mapping method is defined as having high recall and precision. The STACK TRACE METHOD is the most effective among the four, mapping 31 bugs with a precision of 38.2%. All the NPEs mapped to a warning were contained within the STACK TRACE METHOD, except for four. On the other hand, while the CODE DIFF METHOD and COVERAGE METHOD produce the largest number of bug candidates across all tools, they also suffer from the lowest precision, 14.2% and 15.0%, respectively. The REPORT DIFF METHOD mapped the lowest number of true bugs in comparison to other methods (24 bugs), but its precision of 25.3% was still higher than that of the CODE DIFF METHOD and COVERAGE METHOD. The results show that the four methods are primarily complementary of each other as they map different types of information.

RQ2: The STACK TRACE METHOD is the most effective with 81 bug candidates, of which 31 were true bugs (38.2%). The CODE DIFF METHOD and COVERAGE METHOD had similar recall than the STACK TRACE METHOD, but a lower precision of 14.2% and 15.0%, respectively. The REPORT DIFF METHOD had the lowest recall but a higher precision than CODE DIFF METHOD and COVERAGE METHOD.

```
protected Object decode(Channel channel, ...){
- if (channel == null) {
+ if (channel != null) {
      channel.write(response, remoteAddress);
  }
}
```

(a) NPE bug found by both SPOTBUGS and ERADICATE.

```
public class GrblCntrllr extends AbstractCntrllr {
- capabilities = null;
+ capabilities = new GrblUtils.Capabilities()
protected void pauseStreamingEvent(){
  if (this.capabilities.REAL_TIME) { ... }
}
```

(b) NPE bug dereferencing field of an object not found by any tool.

```
protected void ldCmdVerSheet(String sheetName) {
  Sheet sheet = switchToSheet(sheetName, false);
+ if(sheet==null) return;
  while(i<sheet.getRows()) { ... }
}
```

(c) NPE bug due to null dereference of a return value.

```
private void verifyDecodedPosition() {
- if(p.gNtk()!=null){
+ if(p.gNtk()!=null && p.gNtk().gTwrs()!=null){
  for (Twr Twr : p.gNtk().gTwrs()){
}
```

(d) NPE bug with dereferencing object returned from a method.

Fig. 5: Examples of NPE diffs from the dataset.

C. RQ3: Effectiveness of Tools at Finding NPEs

Overall, the tools find 30 distinct bugs out of 102 bugs (29.4%). The breakdown per tool is shown in Table IV. ERADICATE finds the most bugs with 20 out of 30. CFNULLNESS finds 11 bugs, INFER 10 bugs, and SPOTBUGSLT 9 bugs. NULLAWAY and SPOTBUGSHT find the fewest bugs with 5 and 4, respectively. We examined the overlap among bugs found by each tool. The two tools with the most overlap are CFNULLNESS and ERADICATE with 7 bugs. Interestingly, each tool finds bugs not found by other tools (also shown in Table IV). This shows that the tools are complementary, and that practitioners could benefit from running multiple tools. A challenge to this is the large number of warnings to inspect.

An example of a bug found by CFNULLNESS, INFER, and SPOTBUGSLT was given in Figure 2a. We show the diff between a buggy version (with an NPE bug) and the fixed version of the GitHub project `openpnp/openpnp` (a robotic pick and place machine). The call to the buggy method that causes the NPE is located on Line 6 of the buggy program. The fix for this NPE bug consists of adding a null check for parameter `mat` in `saveDebugImage`.

Figure 5a shows an example of a bug found by CFNULLNESS, ERADICATE, SPOTBUGSHT, and SPOTBUGSLT. Here we show the diff between a buggy version and the fixed version of project `traccar/traccar` (a GPS tracking system). The bug was that the null check was flipped, incorrectly dereferencing `channel` when `null`. The fix simply consists of changing the comparison operator from == to !=. A possible reason why INFER did not find this bug is that INFER does not gather information from checks. Since Figure 5a includes a null check, SPOTBUGS is able to reason that `channel` is dereferenced when null, leading to an NPE.

We conducted an additional experiment on a random sample of 40 programs[6] from our initial set for which annotations were inferred using IntelliJ IDEA's Infer Nullity [7]. IntelliJ Idea infers both `@Nullable` and `@NotNull` annotations. Note that 49 out of the 102 programs originally include some nullness annotations. We ran all tools on the annotated programs, except for INFER which does not use annotations. IntelliJ added 34,229 `@Nullable` and 167,236 `@NotNull` annotations. We applied the COVERAGE METHOD to map warnings. This resulted in 2, 3, 0, and 2 additional bugs found by CFNULLNESS, ERADICATE, NULLAWAY, and SPOTBUGSLT, respectively. These accounted for three unique bugs across all tools. Despite the small increase in bugs found, the results are promising as annotating less than half of the programs resulted in finding 10% more bugs in total.

[6]The process could not be automated due to the IDE, thus the sample.

> **RQ3**: Overall, the tools have low effectiveness at finding NPE bugs. Out of the 102 bugs in our dataset, ERADICATE found 20 bugs (19.6%), CFNULLNESS found 11 (10.8%), INFER found 10 (9.8%), SPOTBUGSLT found 9 bugs (8.8%), NULLAWAY found 5 (4.9%), and SPOTBUGSHT found 4 (3.9%). Additional annotations resulted in finding 3 more bugs.

D. RQ4: Reasons Bug Detectors Miss NPEs

We are interested in understanding the reasons why bug detectors fail to find real NPEs. We start by discussing the characteristics of the bugs that the tools find based on the characterization of 102 bugs from our dataset and the tools themselves (see Section III). We then discuss the characteristics of those bugs that the tools fail to find.

a) CFNULLNESS*:* CFNULLNESS found 11 bugs including every category shown in Figure 3. These included 3 dereferences to a method return value and 2 dereferences of a map object. The sound properties of CFNULLNESS allow it to find classes of bugs that the other tools cannot. For example, CFNULLNESS also found bugs due to concurrency, field initialization, generics, and reflection. The lack of necessary annotations in the projects under study inhibits CFNULLNESS's ability to find all of the bugs in those categories.

b) ERADICATE*:* ERADICATE found 20 bugs where 9 dereferenced a method return value, 3 dereferenced an object field, 1 retrieved a value from a map object, and the rest dereferenced a method parameter. Despite using a partial

model of the JDK, ERADICATE missed bugs in other third-party libraries. ERADICATE does not handle concurrency and reflection. These limitations explain some of the false negatives, while others can be explained by the lack of full field initialization checks and dynamic dispatch.

c) INFER*:* INFER found 10 bugs that included 4 dereferences of a method parameter, 4 dereferences of a method return value (one of which is from a JDK library), a dereference of a list, and a dereference of an object field. These NPEs are interprocedural in nature, which aligns with our characterization of INFER in Section III. However, INFER did not find the remaining NPEs that involve method parameters, method return values, or object fields, which we would expect to be captured by interprocedural analysis. One reason is that INFER does not take into account existing null checks.

INFER has an internal partial model of the JDK, which enables reasoning about certain library methods. Surprisingly, despite the fact that INFER supports field sensitivity, and was successful at finding such bugs in our tests, it missed many other field sensitive bugs. Note that INFER does not have a check for field initialization so it does not find uninitialized fields, but it does support fields set to `null`. Such an example is shown in Figure 5b. Additionally, INFER does not find NPEs that involve reflection, concurrency, maps, or use of third-party libraries outside of the JDK.

d) NULLAWAY*:* NULLAWAY found 5 bugs, all of which dereference a return value. This shows the challenge in placing annotations in the *right* place to be beneficial. NULLAWAY's main sources of false negatives are its assumptions that unannotated code is not `null` and that third-party libraries do not return `null`. While manual tests written during our categorization revealed correct warnings about dereferenced fields, real bugs that share these characteristics were not detected. Such an example is shown in Figure 5b, where an unannotated field (considered non-null) is being assigned `null`. This represents a strict violation of the assumption that the field cannot hold a `null` value, and should result in a warning. Finally, in the process of running NULLAWAY, one of the programs crashed the tool. The problem was due to a buggy treatment of certain methods in the standard Java library. We reported the bug to NULLAWAY developers, and it is now fixed in the latest release.

e) SPOTBUGS*:* SPOTBUGS found 9 NPEs, of which 5 occur when dereferencing a method parameter, 3 when dereferencing a method return value, and another when dereferencing a field. In all cases, there is at least one null check within the method for the object being dereferenced, but the programmer dereferences the object in a path that is not checked. The null checks enable SPOTBUGS to reason about the NPEs intraprocedurally (Section III). The remaining NPEs in our sample that dereference a method's return value or parameter are not found because they require interprocedural reasoning. Additionally, the 17 NPEs that involve the dereference of an object field are not found by SPOTBUGS. SPOTBUGS fails to find any bugs dealing with reflection, concurrency, third-party libraries, maps, and lists. This conforms to our tool characterization; SPOTBUGS does not provide complete field sensitivity.

> **RQ4**: SPOTBUGS misses NPEs that require interprocedural analysis. INFER performs interprocedural analysis but does not have a field initialization check nor does it handle some path-sensitive information from null checks. NULLAWAY relies on nullness annotations but does not handle maps nor third-party libraries. ERADICATE deals with third-party libraries better than other tools, but it still misses bugs due to partial field initialization checking. CFNULLNESS provides sound analyses to handle reflection and initialization which allows finding bugs that other tools cannot. However, the lack of annotations can still lead to missed bugs.

E. Threats to Validity

Although we conducted this study on a substantial number of real-world NPEs, our results cannot be generalized. We attempted to reduce this threat by including a large number of NPE bugs from a diverse set of 42 projects from two Java bug datasets. It is possible that we may have missed other tools that are eligible for our study. We still believe that the five tools considered are good representatives of popular and widely-used state-of-the-art static bug detectors for NPEs. The four different mapping methods used in this paper are not perfect, and may lead to false positives. To alleviate this threat, we manually inspected all warnings that were deemed to be bug candidates. Anything requiring human intervention can be error prone and subjective. To mitigate this threat and reduce bias, we involved two people external to our project in the categorization of bugs. Finally, we consulted tool developers to confirm our findings regarding tool capabilities and limitations, as discussed in Section III.

V. LESSONS LEARNED

This section describes some opportunities for improvement.

a) Need for reducing or ranking warnings: Over 500,000 NPE warnings were generated across all tools and programs, with NPEs being in the top-2 warnings for every tool. The average number of warnings per program was in the hundreds, which is a cumbersome amount. Because of this, we had to employ a combination of mapping methods and manual verification to determine if a bug was found. In our case it is known that an NPE exists, and the goal is to determine whether the tools find it. However, this is not the usual setting for tool usage; developers do not know beforehand of the existence of bugs, or else the tools would not be needed. Thus, the large number of warnings is especially problematic in a real setting where true bugs are unknown and all warnings must be inspected.

A bug ranking system could help in navigating the large number of warnings. All tools studied, except for NULLAWAY, provide severity warning information, but this information did not correlate to finding the NPEs under study. For example,

SPOTBUGS provides a severity ranking: "concerning", "troubling", "scary", and "scariest". However, the true bugs found were not associated with the most severe category, but with the "troubling" and "scary" categories. This shows the need for more conservative strategies to process warnings, or to label warnings that are more likely to be true bugs.

Two main approaches for ranking warnings are found in the literature, and could be applied in the context of static bug detectors for NPEs. The first solely focuses on ranking warnings of a specific program version without considering information such as warnings produced for other versions of the program. Examples in this category learn a classifier via methods ranging from bayesian networks, decision trees, and neural networks [21, 43]. The second approach uses the difference of warnings between a previous and the current version of the program, or self-adapts through user feedback [22, 36]. A promising approach to aid static bug detectors for NPEs would be to learn a project-specific classifier that has user-feedback on predictions. This would benefit users as the tool learns, over time, domain-specific project characteristics, which would eventually lead to higher precision.

b) Need for automatically inferring nullability annotations: There is an inherent burden in writing annotations. Analyzers that depend on annotations could benefit from automated inference of nullability annotations. Running IntelliJ IDEA's Infer Nullity on 40 programs enabled the tools to find an additional 3 bugs. This shows that there is promise in annotation-based approaches for bug finding. However, there is room for improvement in annotation inference as the analysis still missed annotations that could have lead to finding more bugs. Furthermore, it was difficult to automate the process of annotating code using IntelliJ, which may prevent its use in many scenarios. There exists work that applies static analysis to infer non-null annotations for object fields in a subset of Java [25], which could be potentially used to aid annotation-based NPE bug detectors but it is not publicly available.

c) Need for reasoning about collection-like data structures: A pain point for all tools studied is reasoning about the nullability of objects inside a collection-like data structure such as an array. Users can add annotations to indicate that a data structure can be `null`, but there is no mechanism to annotate the nullability of individual elements in the data structure. CFNULLNESS, ERADICATE, and NULLAWAY overcome this challenge for map-like objects by assuming that the `get` interface may return a "nullable" value. A similar approach could be adopted each time an element from other collection-like data structure is retrieved. Incorporating such strategy would enable the tools to successfully find 10 additional bugs.

d) Need for reasoning about reflection: Reasoning about reflection imposes a challenge for any static analysis. All of the tools in our study are unsound when it comes to reflection, except CFNULLNESS. Since most of the tools can leverage annotations, a potential approach for handling reflection is user-provided annotations. This is exactly what CFNULLNESS does. This is done via a list of targets, a priori, of what class or method is being operated on for certain reflection calls. This approach has been implemented in other analyses [28, 37, 39, 40] for Java, where analysis precision was observed to improve. Indeed, incorporating the above strategy would enable the NPE bug detectors to find 13 additional bugs, from which CFNULLNESS successfully finds one given the existing annotations.

VI. RELATED WORK

a) Static Analyzer Studies: Rutar et al. [35] compare the static analyzers PMD, FindBugs, JLint, Bandera, and ESC/Java 2 on a small suite of programs. The authors present a taxonomy of bugs found by each tool showing that no tool subsumes the other. The study focuses on runtime and number of warnings produced. Johnson et al. [26] conduct a study in which 20 developers are interviewed on their experiences using static analysis tools. The study finds that the main reason why developers do not use tools is false positives.

Habib and Pradel [20] study the static analyzers INFER, ERROR PRONE, and SPOTBUGS to determine how many of *all* bugs in DEFECTS4J can be found by these tools. The authors use the code diff and the bug report mapping methods. The study finds that only 27 bugs out of 594 bugs (4.5%) were detected, of which only 2 were NPEs. Tomassi [41] conducts a study that compares ERROR PRONE and SPOTBUGS to find how many of *all* bugs in a sample of 320 BUGSWARM artifacts are found. The author found that only one bug was found by SPOTBUGS. Instead, we focus on a specific kind of bug, NPEs, and present a detailed analysis of the capabilities and the limitations of five popular tools that find NPEs.

Ayewah and Pugh [12] run Coverity, Eclipse, FindBugs, Fortify, and XYLEM on different versions of the build system Ant. The authors classify the null dereferences reported by each tool (plausible, implausible, or impossible), and explore the usefulness of using null-related annotations. Most recently, Banerjee et al. [14] presented the tool NULLAWAY and performed a comparison to the Checker Framework's Nullness analysis [33], and INFER's Eradicate looking at build-time overhead. While Ayewah and Pugh [12] study false positives in one version of Ant, Banerjee et al. [14] focus on measuring false negatives in Uber's Android apps. We study the recall of five popular bug detectors, including NULLAWAY, on 102 real and reproducible NPEs from 42 open-source projects.

b) Tools to Find Null Pointer Dereferences: Ayewah et al. [13] present a static analysis tool called FINDBUGS, the predecessor of SPOTBUGS. FINDBUGS finds a wide variety of bugs including null pointer dereferences. Hovemeyer and Pugh [24] extend FINDBUGS's NPE finding capabilities by improving the precision of the analysis. These improvements were a result of a better model of the core API of JDK, changing how errors on exception paths are handled, improving field tracking, and finding guaranteed dereferences. We include SPOTBUGS in our study.

Papi et al. [33] introduce the Checker Framework, which allows for pluggable type systems for Java. They evaluate five checkers, including the Nullness checker, running them over significant sized code bases. The checkers find real bugs

and confirmed the absence of others. We include the Checker Framework in our study.

Nanda and Sinha [32] develop a demand-driven dataflow analysis for null-dereference bugs in Java. By being path-sensitive and context-sensitive, the analysis allows for a low false positive rate, and an improved precision over FINDBUGS and JLint. Romano et al. [34] use the analysis from Nanda and Sinha [32] to find variables and paths that lead to possible null pointer dereferences. The authors use a genetic algorithm to generate tests that trigger the null pointer dereferences. Loginov et al. [29] develop a sound interprocedural analysis based on abstract interpretation called *expanding-scope* algorithm. Madhavan and Komondoor [30] demonstrate a sound, demand-driven, interprocedural, context-sensitive dataflow analysis to verify whether a dereference will be safe or not. None of the above tools [29, 30, 32, 34] are publicly available.

VII. CONCLUSION

In this experience paper, we studied the effectiveness of popular Java static bug detectors CFNULLNESS, ERADICATE, INFER, NULLAWAY, and SPOTBUGS on 102 real NPEs from 42 open-source projects. We identified the capabilities of the tools and the characteristics of the NPE bugs in our dataset. We discussed the problem of mapping tool warnings to actual NPE bugs, and investigated four mapping methods, including two new approaches that leverage stack trace and code coverage information, from which the stack-trace based was the most effective. Overall, the tools detected a total of 30 out of 102 bugs. We conducted an additional experiment annotating 40 programs using IntelliJ, which resulted in 3 new bugs found. Finally, we leveraged the characteristics of the tools and the bugs in our dataset to gain insights into why the tools missed certain types of bugs. We concluded by discussing opportunities for improving NPE bug detection. We provide the link to a public repository that contains both our scripts and the data produced in our experimental evaluation.

ACKNOWLEDGMENT

This work was supported in part by National Science Foundation award CNS-2016735, a Facebook Testing and Verification research award, and a UC Davis Graduate Fellowship. We would like to thank Aditya V. Thakur and Premkumar T. Devanbu for their feedback and suggestions. We also thank Amy Cu, Raisa Putri, Robert Furth, and Ryan Jae for their help manually classifying NPEs, and replicating our experimental results. Lastly, we would like to thank the developers of each tool for their prompt answers to our questions.

REFERENCES

[1] CVE-2013-1134. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2003-1134, 2021.
[2] Checkstyle. https://github.com/checkstyle/checkstyle, 2021.
[3] CVE Null Pointer. https://cve.mitre.org/cgi-bin/cvekey.cgi?keyword=null+pointer, 2021.
[4] Eradicate. https://fbinfer.com/docs/eradicate.html, 2021.
[5] Error Prone. https://github.com/google/error-prone, 2021.
[6] Infer. http://fbinfer.com/, 2021.
[7] IntelliJ. https://www.jetbrains.com/idea/, 2021.
[8] NullAway. https://github.com/uber/NullAway, 2021.
[9] PMD. https://pmd.github.io/, 2021.
[10] SpotBugs. https://spotbugs.github.io/, 2021.
[11] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison-Wesley Longman Publishing Co., Inc., USA, 2006.
[12] N. Ayewah and W. Pugh. Null dereference analysis in practice. In *Proceedings of the 9th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, PASTE '10, pages 65–72, New York, NY, USA, 2010. ACM. doi: 10.1145/1806672.1806686. URL http://doi.acm.org/10.1145/1806672.1806686.
[13] N. Ayewah, D. Hovemeyer, J. D. Morgenthaler, J. Penix, and W. Pugh. Using static analysis to find bugs. *IEEE Softw.*, 25(5):22–29, 2008. doi: 10.1109/MS.2008.130. URL https://doi.org/10.1109/MS.2008.130.
[14] S. Banerjee, L. Clapp, and M. Sridharan. Nullaway: Practical type-based null safety for java. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2019, pages 740–750, New York, NY, USA, 2019. ACM. doi: 10.1145/3338906.3338919. URL http://doi.acm.org/10.1145/3338906.3338919.
[15] C. Calcagno and D. Distefano. Infer: An automatic program verifier for memory safety of C programs. In M. G. Bobaru, K. Havelund, G. J. Holzmann, and R. Joshi, editors, *NASA Formal Methods - Third International Symposium, NFM 2011, Pasadena, CA, USA, April 18-20, 2011. Proceedings*, volume 6617 of *Lecture Notes in Computer Science*, pages 459–465. Springer, 2011. doi: 10.1007/978-3-642-20398-5_33. URL https://doi.org/10.1007/978-3-642-20398-5_33.
[16] C. Calcagno, D. Distefano, P. W. O'Hearn, and H. Yang. Compositional shape analysis by means of bi-abduction. In Z. Shao and B. C. Pierce, editors, *Proceedings of the 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2009, Savannah, GA, USA, January 21-23, 2009*, pages 289–300. ACM, 2009. doi: 10.1145/1480881.1480917. URL https://doi.org/10.1145/1480881.1480917.
[17] C. Calcagno, D. Distefano, J. Dubreil, D. Gabi, P. Hooimeijer, M. Luca, P. W. O'Hearn, I. Papakonstantinou, J. Purbrick, and D. Rodriguez. Moving fast with software verification. In K. Havelund, G. J. Holzmann, and R. Joshi, editors, *NASA Formal Methods - 7th International Symposium, NFM 2015, Pasadena, CA, USA, April 27-29, 2015, Proceedings*, volume 9058 of *Lecture Notes in Computer Science*, pages 3–11. Springer, 2015. doi: 10.1007/978-3-319-17524-9_1. URL https://doi.org/10.1007/978-3-319-17524-9_1.
[18] M. Christakis and C. Bird. What developers want and need from program analysis: an empirical study. In D. Lo, S. Apel, and S. Khurshid, editors, *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, ASE 2016, Singapore, September 3-7, 2016*, pages 332–343. ACM, 2016. doi: 10.1145/2970276.2970347. URL https://doi.org/10.1145/2970276.2970347.
[19] W. Dietl, S. Dietzel, M. D. Ernst, K. Muslu, and T. W. Schiller. Building and using pluggable type-checkers. In R. N. Taylor, H. C. Gall, and N. Medvidovic, editors, *Proceedings of the 33rd International Conference on Software Engineering, ICSE 2011, Waikiki, Honolulu , HI, USA, May 21-28, 2011*, pages 681–690. ACM, 2011. doi: 10.1145/1985793.1985889. URL https://doi.org/10.1145/1985793.1985889.
[20] A. Habib and M. Pradel. How many of all bugs do we find? a study of static bug detectors. In M. Huchard, C. Kästner, and G. Fraser, editors, *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE 2018, Montpellier, France, September 3-7, 2018*, pages 317–328. ACM, 2018. doi: 10.1145/3238147.3238213. URL https://doi.org/10.1145/3238147.3238213.
[21] Q. Hanam, L. Tan, R. Holmes, and P. Lam. Finding patterns in static analysis alerts: improving actionable alert ranking. In P. T. Devanbu, S. Kim, and M. Pinzger, editors, *11th Working Conference on Mining Software Repositories, MSR 2014, Proceedings, May 31 - June 1, 2014, Hyderabad, India*, pages 152–161. ACM, 2014. doi: 10.1145/2597073.2597100. URL https://doi.org/10.1145/2597073.2597100.
[22] K. Heo, M. Raghothaman, X. Si, and M. Naik. Continuously reasoning about programs using differential bayesian inference. In K. S. McKinley and K. Fisher, editors, *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019, Phoenix, AZ, USA, June 22-26, 2019*, pages 561–575. ACM, 2019. doi: 10.1145/3314221.3314616. URL https://doi.org/10.1145/3314221.3314616.
[23] D. Hovemeyer and W. Pugh. Finding bugs is easy. In J. M. Vlissides and D. C. Schmidt, editors, *Companion to the 19th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2004, October 24-28, 2004, Vancouver, BC,*

Canada, pages 132–136. ACM, 2004. doi: 10.1145/1028664.1028717. URL https://doi.org/10.1145/1028664.1028717.

[24] D. Hovemeyer and W. Pugh. Finding more null pointer bugs, but not too many. In M. Das and D. Grossman, editors, *Proceedings of the 7th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering, PASTE'07, San Diego, California, USA, June 13-14, 2007*, pages 9–14. ACM, 2007. doi: 10.1145/1251535.1251537. URL https://doi.org/10.1145/1251535.1251537.

[25] L. Hubert, T. P. Jensen, and D. Pichardie. Semantic foundations and inference of non-null annotations. In G. Barthe and F. S. de Boer, editors, *Formal Methods for Open Object-Based Distributed Systems, 10th IFIP WG 6.1 International Conference, FMOODS 2008, Oslo, Norway, June 4-6, 2008, Proceedings*, volume 5051 of *Lecture Notes in Computer Science*, pages 132–149. Springer, 2008. doi: 10.1007/978-3-540-68863-1_9. URL https://doi.org/10.1007/978-3-540-68863-1_9.

[26] B. Johnson, Y. Song, E. R. Murphy-Hill, and R. W. Bowdidge. Why don't software developers use static analysis tools to find bugs? In D. Notkin, B. H. C. Cheng, and K. Pohl, editors, *35th International Conference on Software Engineering, ICSE '13, San Francisco, CA, USA, May 18-26, 2013*, pages 672–681. IEEE Computer Society, 2013. doi: 10.1109/ICSE.2013.6606613. URL https://doi.org/10.1109/ICSE.2013.6606613.

[27] R. Just, D. Jalali, and M. D. Ernst. Defects4j: a database of existing faults to enable controlled testing studies for java programs. In C. S. Pasareanu and D. Marinov, editors, *International Symposium on Software Testing and Analysis, ISSTA '14, San Jose, CA, USA - July 21 - 26, 2014*, pages 437–440. ACM, 2014. doi: 10.1145/2610384.2628055. URL https://doi.org/10.1145/2610384.2628055.

[28] O. Lhoták and L. J. Hendren. Scaling java points-to analysis using SPARK. In G. Hedin, editor, *Compiler Construction, 12th International Conference, CC 2003, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2003, Warsaw, Poland, April 7-11, 2003, Proceedings*, volume 2622 of *Lecture Notes in Computer Science*, pages 153–169. Springer, 2003. doi: 10.1007/3-540-36579-6_12. URL https://doi.org/10.1007/3-540-36579-6_12.

[29] A. Loginov, E. Yahav, S. Chandra, S. Fink, N. Rinetzky, and M. G. Nanda. Verifying dereference safety via expanding-scope analysis. In B. G. Ryder and A. Zeller, editors, *Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2008, Seattle, WA, USA, July 20-24, 2008*, pages 213–224. ACM, 2008. doi: 10.1145/1390630.1390657. URL https://doi.org/10.1145/1390630.1390657.

[30] R. Madhavan and R. Komondoor. Null dereference verification via over-approximated weakest pre-conditions analysis. In C. V. Lopes and K. Fisher, editors, *Proceedings of the 26th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2011, part of SPLASH 2011, Portland, OR, USA, October 22 - 27, 2011*, pages 1033–1052. ACM, 2011. doi: 10.1145/2048066.2048144. URL https://doi.org/10.1145/2048066.2048144.

[31] G. J. Myers, T. Badgett, T. M. Thomas, and C. Sandler. *The art of software testing*, volume 2. Wiley Online Library, 2004.

[32] M. G. Nanda and S. Sinha. Accurate interprocedural null-dereference analysis for java. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, pages 133–143, Washington, DC, USA, 2009. IEEE Computer Society. doi: 10.1109/ICSE.2009.5070515. URL http://dx.doi.org/10.1109/ICSE.2009.5070515.

[33] M. M. Papi, M. Ali, T. L. C. Jr., J. H. Perkins, and M. D. Ernst. Practical pluggable types for java. In B. G. Ryder and A. Zeller, editors, *Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2008, Seattle, WA, USA, July 20-24, 2008*, pages 201–212. ACM, 2008. doi: 10.1145/1390630.1390656. URL https://doi.org/10.1145/1390630.1390656.

[34] D. Romano, M. D. Penta, and G. Antoniol. An approach for search based testing of null pointer exceptions. In *Fourth IEEE International Conference on Software Testing, Verification and Validation, ICST 2011, Berlin, Germany, March 21-25, 2011*, pages 160–169. IEEE Computer Society, 2011. doi: 10.1109/ICST.2011.49. URL https://doi.org/10.1109/ICST.2011.49.

[35] N. Rutar, C. B. Almazan, and J. S. Foster. A comparison of bug finding tools for java. In *Proceedings of the 15th International Symposium on Software Reliability Engineering*, ISSRE '04, pages 245–256, Washington, DC, USA, 2004. IEEE Computer Society. doi: 10.1109/ISSRE.2004.1. URL http://dx.doi.org/10.1109/ISSRE.2004.1.

[36] H. Shen, J. Fang, and J. Zhao. Efindbugs: Effective error ranking for findbugs. In *Fourth IEEE International Conference on Software Testing, Verification and Validation, ICST 2011, Berlin, Germany, March 21-25, 2011*, pages 299–308. IEEE Computer Society, 2011. doi: 10.1109/ICST.2011.51. URL https://doi.org/10.1109/ICST.2011.51.

[37] M. Sridharan, S. Artzi, M. Pistoia, S. Guarnieri, O. Tripp, and R. Berg. F4F: taint analysis of framework-based web applications. In C. V. Lopes and K. Fisher, editors, *Proceedings of the 26th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2011, part of SPLASH 2011, Portland, OR, USA, October 22 - 27, 2011*, pages 1053–1068. ACM, 2011. doi: 10.1145/2048066.2048145. URL https://doi.org/10.1145/2048066.2048145.

[38] F. Thung, Lucia, D. Lo, L. Jiang, F. Rahman, and P. T. Devanbu. To what extent could we detect field defects? an empirical study of false negatives in static bug finding tools. In M. Goedicke, T. Menzies, and M. Saeki, editors, *IEEE/ACM International Conference on Automated Software Engineering, ASE'12, Essen, Germany, September 3-7, 2012*, pages 50–59. ACM, 2012. doi: 10.1145/2351676.2351685. URL https://doi.org/10.1145/2351676.2351685.

[39] F. Tip, C. Laffra, P. F. Sweeney, and D. Streeter. Practical experience with an application extractor for java. In B. Hailpern, L. M. Northrop, and A. M. Berman, editors, *Proceedings of the 1999 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages & Applications (OOPSLA '99), Denver, Colorado, USA, November 1-5, 1999*, pages 292–305. ACM, 1999. doi: 10.1145/320384.320414. URL https://doi.org/10.1145/320384.320414.

[40] F. Tip, P. F. Sweeney, C. Laffra, A. Eisma, and D. Streeter. Practical extraction techniques for java. *ACM Trans. Program. Lang. Syst.*, 24 (6):625–666, 2002. doi: 10.1145/586088.586090. URL https://doi.org/10.1145/586088.586090.

[41] D. A. Tomassi. Bugs in the wild: examining the effectiveness of static analyzers at finding real-world bugs. In *Proceedings of the 2018 ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2018, Lake Buena Vista, FL, USA, November 04-09, 2018*, pages 980–982, 2018. doi: 10.1145/3236024.3275439. URL https://doi.org/10.1145/3236024.3275439.

[42] D. A. Tomassi, N. Dmeiri, Y. Wang, A. Bhowmick, Y.-C. Liu, P. T. Devanbu, B. Vasilescu, and C. Rubio-González. Bugswarm: Mining and continuously growing a dataset of reproducible failures and fixes. In *Proceedings of the 41st International Conference on Software Engineering*, ICSE '19, pages 339–349, Piscataway, NJ, USA, 2019. IEEE Press. doi: 10.1109/ICSE.2019.00048. URL https://doi.org/10.1109/ICSE.2019.00048.

[43] L. Yu, W. Tsai, W. Zhao, and F. Wu. Predicting defect priority based on neural networks. In L. Cao, J. Zhong, and Y. Feng, editors, *Advanced Data Mining and Applications - 6th International Conference, ADMA 2010, Chongqing, China, November 19-21, 2010, Proceedings, Part II*, volume 6441 of *Lecture Notes in Computer Science*, pages 356–367. Springer, 2010. doi: 10.1007/978-3-642-17313-4_35. URL https://doi.org/10.1007/978-3-642-17313-4_35.

[44] M. Zhivich and R. K. Cunningham. The real cost of software errors. *IEEE Secur. Priv.*, 7(2):87–90, 2009. doi: 10.1109/MSP.2009.56. URL https://doi.org/10.1109/MSP.2009.56.