Effective Unit Test Generation for Java Null Pointer Exceptions

Myungho Lee
Korea University
Republic of Korea
myungho_lee@korea.ac.kr

Jiseong Bak
Korea University
Republic of Korea
js_bak@korea.ac.kr

Seokhyeon Moon
Technology Research, Samsung SDS
Republic of Korea
shyeon.mun@samsung.com

Yoon-Chan Jhi
Technology Research, Samsung SDS
Republic of Korea
yoonchan.jhi@samsung.com

Hakjoo Oh*
Korea University
Republic of Korea
hakjoo_oh@korea.ac.kr

ABSTRACT

In this experience paper, we share our experience on enhancing automatic unit test generation to more effectively find Java null pointer exceptions (NPEs). NPEs are among the most common and critical errors in Java applications. However, as we demonstrate in this paper, existing unit test generation tools such as Randoop and EvoSuite are not sufficiently effective at catching NPEs. Specifically, their primary strategy of achieving high code coverage does not necessarily result in triggering diverse NPEs in practice. In this paper, we detail our observation on the limitations of current state-of-the-art unit testing tools in terms of NPE detection and introduce a new strategy to improve their effectiveness. Our strategy utilizes both static and dynamic analyses to guide the test case generator to focus specifically on scenarios that are likely to trigger NPEs. We implemented this strategy on top of EvoSuite, and evaluated our tool, NpeTest, on 108 NPE benchmarks collected from 96 real-world projects. The results show that our NPE-guidance strategy can increase EvoSuite's reproduction rate of the NPEs from 56.9% to 78.9%, a 38.7% improvement. Furthermore, NpeTest successfully detected 89 previously unknown NPEs from an industry project.

ACM Reference Format:
Myungho Lee, Jiseong Bak, Seokhyeon Moon, Yoon-Chan Jhi, and Hakjoo Oh. 2024. Effective Unit Test Generation for Java Null Pointer Exceptions. In *39th IEEE/ACM International Conference on Automated Software Engineering (ASE '24), October 27-November 1, 2024, Sacramento, CA, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3691620.3695484

1 INTRODUCTION

Null pointer exceptions (NPEs) are one of the most common and fatal errors in Java applications [1–6]. NPE is a critical software defect because dereferencing a null pointer always makes the program crash, causing undefined behavior of the entire system. In addition to the hazardous impacts of NPEs on software, NPEs frequently occur in practice, making them even more fatal. For example, recent industry reports [7, 8] on the causes for errors in real-world Java applications, often discussed in technical blogs, show that NPEs account for the most significant portion of the reported crashes. Therefore, software testing is mandatory to reduce the risk of NPEs during the software development process.

Unit Testing. Unit testing has been one of the most widely used software testing techniques for object-oriented programming languages such as Java. With well-designed test cases that represent the usage scenarios of a certain unit to test, unit testing validates that each unit of the software performs as expected, where a unit is typically an individual method or object. Due to its characteristics that enable testing of an individual single unit with diverse usage scenarios, unit testing helps facilitate software maintainability and is used to prevent future regressions, which improves overall software quality during the development process. From this perspective, unit tests are widely utilized to detect software defects. However, finding bug-triggering unit tests is a complex and time-consuming task, which becomes more difficult with respect to the size and complexity of software systems.

Automatic Unit Test Generation. To reduce the burdens of developers on designing unit tests, automatic test case generation techniques have been proposed with two major approaches: random testing and search-based software testing. Both methods generate test cases by automatically synthesizing method call sequences for the target unit, without assuming existing test drivers.

Random testing is a simple yet effective approach to automatically generating method sequences for object-oriented programs. It randomly synthesizes a sequence of method calls to generate test cases. Among various strategies, feedback-directed random testing has been considered superior to pure random testing, which is implemented in tools such as Randoop [9]. Rather than blindly generating test cases, Randoop leverages the knowledge obtainable through the execution of generated test cases. It runs contract checkers for each generated object to maintain a pool of valid objects. This feedback-directed approach has become a primary test generation strategy for programs written in Java and C#.

Search-based software testing (SBST), on the other hand, formulates the test generation as an optimization problem concerning certain coverage criteria (e.g., branch coverage, exception coverage). In particular, EvoSuite [10] is the state-of-the-art SBST tool for Java, which leverages genetic algorithms (e.g., DynaMOSA [11] and MOSA [12]) to optimize the test suite. It first randomly generates the initial population similar to random testing, and then

*Corresponding author.

ASE '24, October 27-November 1, 2024, Sacramento, CA, USA

ACM ISBN 979-8-4007-1248-7/24/10.
https://doi.org/10.1145/3691620.3695484

finds an optimal test suite concerning target coverage criteria. Over the last decade, SBST has been popular and extensively used for object-oriented programs [13–17].

Observation on NPE Detection. We observed that state-of-the-art tools, Randoop and EvoSuite, are not sufficiently effective at catching diverse NPEs in practice. These unit testing techniques primarily strive for high code coverage with the expectation that obtaining high code coverage will ultimately lead to bug detection. Unfortunately, however, we experienced that achieving higher code coverage does not necessarily result in better NPE-finding performance; using a highly tuned configuration for EvoSuite to increase line coverage from 64.5% to 77.8% on our benchmark programs ended up improving its NPE-reproduction rate marginally from 55.7% 56.9% (Section 4.3). This is because software bugs, especially NPEs, usually occur under certain conditions [18–20]. For example, NPEs are detected only if the value of the dereferenced pointer remains `null` at the time the corresponding buggy statement is executed. Hence, detecting NPEs requires taking one more step from just achieving high code coverage.

Our Approach. To take unit testing techniques one step further for effective NPE detection, we propose a new strategy that guides the testing process to generate test cases that are more likely to explore NPE-prone regions, where NPE may occur. To this end, our approach leverages both static and dynamic analyses. We use static analysis to identify the methods that may trigger NPEs and compute backward dependencies to pinpoint the statements and variables that need to be mutated to reach those methods. We utilize dynamic analysis to gather runtime information and adaptively spend more testing budget on less explored NPE-related code regions.

Experiments show that our approach significantly improves the existing unit test generator in terms of NPE detection. We implemented our approach on top of EvoSuite, and evaluated its performance on 108 known NPEs collected from 96 real-world projects [3, 21–24] in comparison with Randoop and EvoSuite. In summary, NpeTest showed a reproduction rate of 78.9% for those NPEs while Randoop and EvoSuite achieved 33.7% and 56.9%, respectively. Regarding the number of detected bugs, 12 bugs were exclusively found by NpeTest, 1 by EvoSuite, and 0 by Randoop.

We also conducted an industrial case study, applying the three tools, Randoop, EvoSuite, and NpeTest, to a software project actively used and maintained within a large IT company. Through the case study, we found a total of 91 previously unknown NPEs: NpeTest detected 89 NPEs, whereas EvoSuite and Randoop detected 82 and 52 NPEs, respectively. This case study prompted the development team to plan incorporating NpeTest into their development pipeline, enhancing overall code security and robustness.

Contributions. We summarize our contributions below:

- **Observation**: We show that increasing code coverage does not always lead to better NPE detection. Using highly tuned EvoSuite did not meaningfully improve NPE-reproduction rates.
- **New Technique**: We present NpeTest, a new NPE-detection strategy for EvoSuite. To our knowledge, NpeTest is the first technique for unit test generators that focuses on NPE detection.
- **Evaluation and Comparison**: We demonstrate the effectiveness of NpeTest by evaluating it on 108 real-world NPEs and comparing it with Randoop and EvoSuite.
- **Industrial Case Study**: We show that NpeTest can be useful in industrial contexts by conducting a case study on a software project actively maintained within a major IT company.

2 MOTIVATING EXAMPLE

Figure 1 describes an NPE found in Qpid Proton-J project[1]. The root cause of this NPE is the `null` literal assigned to the variable `amqpType` in the `deduceTypeFromClass` method (line 4, Figure 1b), which is returned without refinement. The NPE is thrown at line 8 in Figure 1a, when dereferencing the return value of the `getType` method (line 7, Figure 1a)—which internally calls the `deduceTypeFromClass` method shown in Figure 1b and returns the result directly.

However, the conditions under which the variable `amqpType` is not refined during the execution are not trivial. The type of the first argument should be set properly, which is determined by the argument of the `calculateSize` method in Figure 1a. For example, as shown in Figure 1b, if the type of the first argument when calling `deduceTypeFromClass` is an array type, it passes the true branch at line 5 and `amqpType` will be redefined, which never triggers NPE. Also, considering that this method is called in the while loop (line 4, Figure 1a), the input `map` of the `calculateSize` method should contain at least one element.

Figure 1c is an NPE-triggering test case written by developers, which leads to NPE whose stack trace is in Figure 1d. By adding the element in the mapping variable `map` at line 5, this test case can enter inside the while loop, which satisfies the first step to trigger the NPE. Additionally, the instantiation of `Map<Integer, Object>` causes the `Object` type (i.e., `Class<Object>`) to be passed to the first parameter of the `deduceTypeFromClass` method. This parameter bypasses the condition at line 5 in Figure 1b as its type is not an array type, leading `deduceTypeFromClass` to return `null`, which triggers an NPE in Figure 1a.

In order to generate such test cases triggering the NPE, unit test generation tools must focus on mutating various types for generic type parameters of `Map` and find an appropriate one that bypasses the branch conditions in `deduceTypeFromClass` not to refine the value of `amqpType`. Although EvoSuite, the state-of-the-art unit test generation tool, We found that EvoSuite and Randoop failed to generate such test cases due to the large space of test cases and statements to be mutated. Additionally, due to the large space of the methods to be tested, problematic methods to focus on should be determined carefully for better efficiency.

NpeTest follows such direction in generating test cases via static and dynamic analyses. First, our static analysis identifies statements (e.g., Lines 4–5 in Figure 1c) in test cases that are highly correlated with problematic method arguments and variables (e.g., "map" variable) that are likely to occur NPEs. Second, NpeTest mutates test cases that explore NPE-related regions (e.g., Line 8 in Figure 1a) more aggressively than the other test cases with the aid of dynamic analysis. More specifically, NpeTest monitors a sequence of called methods during execution for each generated test case and

[1]https://github.com/apache/qpid-proton-j

```java
class MapType {
  EncoderImpl encoder;
  int calculateSize(final Map<?, ?> map) {
    for (Entry e : map.entrySet()) {
      Object k = e.getKey();
      if (fixedKeyType == null) {
        AMQPType t = encoder.getType(k);
        enc = t.getEncoding(k); // NPE
    }}}}
```

(a) MapType.java

```java
class EncoderImpl {
  AMQPType<?> deduceTypeFromClass(
      Class<?> cl, Object o) {
    AMQPType<?> amqpType = null;
    if (cl.isArray()) {
      amqpType = _arrayType;
    } else { /* ... */ }
    return amqpType;}
}
```

(b) EncoderImpl.java

```java
DecoderImpl dec = new DecoderImpl();
EncoderImpl enc = new EncoderImpl(dec);
MapType mapt = new MapType(enc, dec);
Map<Integer, Object> map = new HashMap<>();
map.put(0, new Object());
mapt.getEncoding(map);
```

(c) Test case that triggers the NPE

```
java.lang.NullPointerException
  at MapType.calculateSize(MapType.java:8)
  at MapType.getEncoding(...)
  at TestCase.test(TestCase.java:8)
```

(d) The stack trace from NPE-triggering test case

Figure 1: An NPE from the project Apache Qpid Proton-j's revision 02998b3

prioritizes the test cases covering the NPE-related methods (e.g., "calculateSize" method in Figure 1d). With the help of static and dynamic analyses, NpeTest indicates the problematic methods and statements in the test case to focus on for mutation and successfully generates test cases triggering NPEs.

3 APPROACH

In this section, we present our approach to guide unit test generators for better NPE detection. NpeTest consists of two components: (1) **static analysis** which identifies NPE-prone regions and performs dependency analysis to track the statement of test case relevant to NPE-prone regions, and (2) **dynamic analysis** which gathers runtime information to adaptively update the guidance.

3.1 Search-Based Software Testing

Algorithm 1 shows the simplified test case generation process of EvoSuite, a search-based software testing (SBST) tool, which NpeTest relies on. Initially, the algorithm identifies the coverage goals *covGoals* (Line 1), which serve as objectives to guide the test case generations. It also builds an initial parent population by randomly generating N test cases (Line 2). The algorithm then generates offspring population from the parent population by randomly applying various mutation operators such as statement crossover, insertion, deletion, and change (Line 4). Subsequently, it evaluates all test cases within both parent and offspring populations by computing their fitness values with respect to each of *covGoals*. The promising *TCs* are selected as the next *parentTCs*, for further test case exploration (Line 5). In addition, the algorithm can dynamically update *covGoals* to guide test case generations toward more promising code regions, thereby achieving more *covGoals* (Line 6). Once the given time-budget expires (Line 7), the algorithm returns the set of test cases, each of which covers at least one coverage goal, as the final solution (Lines 8–9).

Workflow of NpeTest. Algorithm 2 outlines the workflow of our approach, NpeTest, which builds upon EvoSuite (Algorithm 1).

Algorithm 1 Simplified Test Case Generation Process - EvoSuite

Input: A class under test (CUT) C and a time-budget *timeout*.
Output: A set of test cases *TCs*.

1: $covGoals \leftarrow \text{getGoals}(C)$
2: $parentTCs \leftarrow \text{buildInitPopulation}(C)$
3: **repeat**
4: $\quad childTCs \leftarrow \text{genTests}(parentTCs)$
5: $\quad parentTCs \leftarrow \text{updateSet}(parentTCs \bigcup childTCs, covGoals)$
6: $\quad covGoals \leftarrow \text{updateGoals}(C, covGoals)$
7: **until** *timeout* reached
8: $TCs \leftarrow \text{getSolution}(parentTCs, covGoals)$
9: **return** *TCs*

Algorithm 2 NPE Guided Strategy - NpeTest

Input: A class under test (CUT) C and a time-budget *timeout*.
Output: A set of test cases *TCs*.

1: $\mathbb{R} \leftarrow \text{staticAnalysis}(C)$ ▷ Static analysis
2: $covGoals_{NPE} \leftarrow \text{getNPEGoals}(C, \mathbb{R})$
3: $M \leftarrow \text{getNPEFunctions}(C, \mathbb{R})$ ▷ Remove NPE-free methods
4: $TCs \leftarrow \text{buildInitPopulation}(C, M)$
5: **repeat**
6: $\quad TC \leftarrow \text{genNPETC}(TCs, M, \mathbb{R})$ ▷ Dynamic & Static analyses
7: $\quad TCs \leftarrow \text{updateSet}(\{TC\} \bigcup TCs, covGoals_{NPE})$
8: $\quad M \leftarrow \text{updateMUTs}(TC, covGoals_{NPE})$ ▷ Dynamic analysis
9: $\quad TCs \leftarrow \text{computeTCScore}(TCs, M)$ ▷ Dynamic analysis
10: **until** *timeout* reached
11: $TCs \leftarrow \text{getSolution}(TCs, covGoals)$
12: **return** *TCs*

NpeTest leverages both static and dynamic analyses to generate more NPE-triggering test cases (Lines 1–3, 6 and 8).

NpeTest first performs static analysis on the given class C (Line 1) and identifies NPE-prone methods ($\mathbb{R}$) where NPE can be triggered. The results $\mathbb{R}$ contain the information of the NPE-prone regions, represented by program locations where NPE may occur, and the method arguments that are highly related to those regions. With an analysis result $\mathbb{R}$, NpeTest computes the coverage goals $covGoals_{NPE}$, which additionally has a coverage goal for NPE detection (Line 2). It also collects a set of callable NPE-prone methods M that are identified by the static analyzer (Line 3).

In contrast to EvoSuite, which relies on *random selection* of test cases and their statements to produce offspring populations, NpeTest uses static and dynamic analyses to *selectively choose* test cases and statements that are likely to trigger NPEs when mutated (Line 6). In addition, after updating the population based on the NPE detection coverage goals (Line 7), NpeTest refines the set of methods under test M by removing those for which all NPEs have been detected, allowing it to concentrate on methods with undetected NPEs (Line 8). Then, NpeTest computes the *testcase-level NPE-likely score* which indicates how likely the NPE may occur.

3.2 Static Analysis

Our static analyzer is (1) to identify all NPE-prone regions and the methods in a class under test (CUT) and (2) to prioritize the statements to be mutated in a given test case.

Language. We designed static analyzer to follow the general definition of the Java language. A program P is a sequence of classes, where a class C is a sequence of methods. A method $\mathtt{m}$ consists of a return type, a sequence of parameters, and a sequence of statements. The type of each method and variable is either a primitive type (e.g., boolean) or a reference type (e.g., array). We consider a statement $\mathtt{S}$ and an expression $\mathtt{E}$ as follows:

$$\begin{array}{lcl} \mathtt{S} & \rightarrow & x = \mathtt{E} \mid return\ \mathtt{E} \mid if\ \mathtt{E}\ \mathtt{S}_1\ \mathtt{S}_2 \mid while\ \mathtt{E}\ \mathtt{S} \mid \mathtt{S}_1\ ;\ \mathtt{S}_2 \mid \epsilon \\ \mathtt{E} & \rightarrow & new\ C() \mid call(\mathtt{m}) \mid \mathtt{E}_1 == \mathtt{E}_2 \mid n \mid \mathtt{null} \mid x \mid \mathtt{E.E} \mid \ldots \end{array}$$

where a variable x can be either a local or a field variable in a class. In this paper, we consider each generated test case TC as a unique method that follows the grammar above. We assume that the names of each class and method are uniquely defined.

Path Construction. Given a CUT, we first construct a control-flow graph (CFG) of each method from a codebase, where each node represents an atomic statement. Based on the CFG, we compute a set T_{exp} of a pair of target expression exp and a line number loc (i.e., $(exp, loc) \in T_{exp}$) for each method $\mathtt{m}$. We gather the expression into a set T_{exp} if it satisfies at least one of the following conditions:

(1) An expression $\mathtt{E}_1$ in a form of "$\mathtt{E}_1.\mathtt{E}_2$".
(2) An invocation of a method which is an NPE-prone method.
(3) A return variable when the return type is a reference type.

Using the set T_{exp}, we classify any methods as NPE-safe methods if no target expressions exist in a method $\mathtt{m}$ (i.e., $T_{exp}(\mathtt{m}) == \emptyset$) or all target expressions only satisfy the last condition (3).

Example 1. Consider the methods presented in Figure 2. For each method, a set of collected target expressions are as follows:

$$\begin{array}{l} T_{exp}(addEdge) : \{(getEdge(src), 3), (src, 4), (getEdge(target), 5)\}, \\ T_{exp}(getEdge) \ : \{(vertexMap, 2), (vertexMap, 5), (ec, 7)\}. \end{array}$$

For both methods addEdge and getEdge, since T_{exp} is not empty and the target expressions in each set are not the return variables except for the variable "ec" in getEdge method, they are NPE-prone methods at this moment.

With the set T_{exp}, we construct a finite set $Set(Info_{path})$ of path information $Info_{path}$ for each target expression. Each path information of a method $\mathtt{m}$ is a tuple ($path$, $isNull$, exp), where $path$ is a sequence of statements, $isNull$ is a mapping variable from reference variables to boolean values, and exp is a single target expression from $T_{exp}(\mathtt{m})$. Note that each reference variable in $isNull$ is initially assigned to $true$. A path $path$ is obtained by adding relevant statements via backward propagation starting from the target expression exp until the propagation meets one of the following conditions:

(1) The propagation reaches to the entry point of the method.
(2) exp is in a form of method invocation (e.g., $call()$ in $call().z()$).
(3) $call(\mathtt{m}')$ when the method $\mathtt{m}'$ has side-effects on exp.
(4) $x = \mathtt{E}$ where x is the firstly defined variable relevant to exp.
(5) $\mathtt{E}$ in a branch condition is relevant to exp.

For simplicity, we assume that the invocation of any methods ($call()$) in the method $\mathtt{m}$ does not have side effects on the target expression exp.

Nullable Path Identification. To analyze whether the target expression can be null in a given path, we define a null checker ($\Theta : \mathtt{S} \times isNull \rightarrow isNull$), which follows the rules below:

$$\begin{array}{rcl} \Theta(x := \mathtt{null}, isNull) & = & isNull[x \mapsto true] \\ \Theta(x := y, isNull) & = & isNull[x \mapsto isNull[y]] \\ \Theta(x := new\ C(), isNull) & = & isNull[x \mapsto false] \\ \Theta(x := call(\mathtt{m}'), isNull) & = & isNull[x \mapsto Ret_{null}(\mathtt{m}')] \\ \Theta(x == \mathtt{null}, isNull) & = & isNull[x \mapsto true], \\ \Theta(\neg(x == \mathtt{null}), isNull) & = & isNull[x \mapsto false], \end{array}$$

where $Ret_{null}(\mathtt{m})$ indicates whether the method $\mathtt{m}$ may return null. Θ updates the mapping variable $isNull$ until it reaches the end of the path. If $isNull[exp]$ remains $false$ for all paths towards the given target expression exp, we can conclude that NPE never occurs when dereferencing the expression exp. Otherwise, there may exist NPE if there exists at least one path where the value of exp remains $true$. If the target expression exp in a method $\mathtt{m}'$ is a return variable, we update $Ret_{null}(\mathtt{m}')$ with the value of $isNull(exp)$. Note that all target expressions in $isNull$ are initially set to $true$.

Example 2. Consider the expression (ec, 7) of $T_{exp}(getEdge)$ in the Example 1. We can build paths towards ec as follows:

$path_1 \rightarrow (ec, 7)$:
`[ ec = new Edge<>(edgeFactory, vertex); return ec ],`
$path_2 \rightarrow (ec, 7)$: `[ !(ec == null); return ec ],`

At the entry of $path_1$, the checker Θ maps the variable ec to $false$ in $isNull$; the instantiation of a constructor call always returns a non-null object. Since all paths towards the return statement result in non null, Ret_{null} of the method getEdge is set to false. With this analysis result, NpeTest can infer that addEdge method is an NPE-safe method; getEdge method never returns null (i.e., Ret_{null} is $false$), and line 3 and 5 in Figure 2a are safe from NPEs.

NPE-likely Score Computation. After nullable path identification is done on all methods, we compute the NPE-likely score

```java
public boolean addEdge(V src, V target, E e) {
  if (src == null) return false;
  getEdge(src).addEdge(e);
  if (!src.equals(target)) {
     getEdge(target).addEdge(e);
  }
  return true; }
```

(a) addEdge method

```java
public Edge<V, E> getEdge(V vertex) {
  Edge<V, E> ec = vertexMap.get(vertex);
  if (ec == null) {
     ec = new Edge<>(edgeFactory, vertex);
     vertexMap.put(vertex, ec);
  }
  return ec; }
```

(b) getEdge method

Figure 2: A simplified code example

$\mathbb{S}_{NPE}(\mathtt{m})$ for the given method m as follows:

$$\mathbb{S}_{path}(\mathtt{m}) = \frac{|mayNull(Set(Info_{path}))|}{|Set(Info_{path})|}$$
$$\mathbb{S}_{NPE}(\mathtt{m}) = \mathbb{S}_{path}(\mathtt{m}) + \sum_{\mathtt{m}' \in \Phi(\mathtt{m})} \mathbb{S}_{NPE}(\mathtt{m}')$$

where $\Phi(\mathtt{m})$ is the set of methods invoked in the given method m. $mayNull(Set(Info_{path}))$ returns a set of path information whose $isNull[exp]$ returns *true*; in other words, it returns a set of paths that may cause NPEs. This NPE-likely score $\mathbb{S}_{NPE}$ is later used for test case selection during mutation; a test case which calls the methods with higher $\mathbb{S}_{NPE}$ is more likely to be selected for mutation.

Mutation Target Selection. We first define a test case *TC* with *n* statements as a quadruple $(\mathtt{Stmts}, \mathtt{MUT}, \mathbb{S}_{TC}, \delta)$, where Stmts is a sequence of statements (i.e., $< \mathtt{S}_1, \mathtt{S}_2, ... \ \mathtt{S}_n >$) and MUT represents method under test in the test case *TC*. $\mathbb{S}_{TC}$ and δ are the weight annotated on the given test case *TC* and a set of executed methods when running *TC*, respectively.

Besides nullable path identification, we perform dependency analysis to obtain parameter variables of the method m which have dependency on any *exp* in $T_{exp}(\mathtt{m})$. Based on the paths we constructed, we backwardly compute the dependencies and the mapping variable $\mathbb{P}$ which maps a method m to a set of indices of the parameters of m, where the parameters have dependencies on the target expressions *exp*.

Given a test case *TC* for mutation, instead of randomly selecting statements to be mutated, NpeTest selects statements and variables that can trigger NPEs in a method MUT (Line 6 in Algorithm 2). More precisely, NpeTest first identifies the arguments *Args* of MUT which are located on the same indices in $\mathbb{P}(\mathtt{MUT})$. From the statement $\mathtt{S}_n$ which invokes MUT, it computes backward dependencies by propagating the sequence of statements Stmts to collect all statements that have dependency on *Args*. Then, for mutation, NpeTest selects statements and variables from the collected set of statements instead of a set of all statements Stmts.

3.3 Dynamic Analysis

The goal of dynamic analysis is to guide the mutation generation process to actively explore NPE-prone methods by monitoring the execution results of test cases. Instrumentation of EvoSuite allows NpeTest to dynamically obtain a set of executed method calls δ for each test case, as well as the information of exceptions caused during execution.

```
java -jar ./evosuite.jar -Dsearch_budget 300 -class [TARGET_CLASS]
      -projectCP [CLASS_PATH] -Dassertions false
      -Dcatch_undeclared_exceptions false -generateMOSuite
      -Dalgorithm=DynaMOSA -Dstatistics_backend=NONE
      -Dshow_progress=false -Dnew_statistics=false -Dcoverage=false
      -Dinline=true -Dp_functional_mocking=0.8
      -Dp_reflection_on_privagte=0.5 - ...
```

Figure 3: Fine-tuned options for EvoSuite

Method Under Test Refinement. Using the information of runtime exceptions, NpeTest dynamically refines the set of methods under test. More precisely, NpeTest maintains a set of target expressions for each method (i.e., $(exp, loc) \in T_{exp}$). If NPEs occur during test case execution, NpeTest gathers the information of the method m and the NPE-triggered error location *loc* from the runtime exception error logs, and removes the corresponding target expression *exp* from $T_{exp}(\mathtt{m})$ (Line 8 in Algorithm 2). By dynamically pruning out the method m of which all NPEs are detected (i.e., $T_{exp}(\mathtt{m}) == \emptyset$), our dynamic analyzer enables unit test generators to spend more time on the NPE-related methods that have not been explored enough.

Testcase-level NPE-likely Score Computation. Using the aforementioned sequence of executed method calls, NpeTest calculates and maintains *testcase-level NPE-likely score* (i.e., $\mathbb{S}_{TC}$). For each test case *TC*, a quadruple of $(\mathtt{Stmts}, \mathbb{S}_{TC}, \mathtt{m}, \delta)$, we compute $\mathbb{S}_{TC}$ as the sum of $\mathbb{S}_{NPE}$ of all methods executed over *TC* execution (i.e. $\mathbb{S}_{TC} = \sum_{\mathtt{m} \in \delta} \mathbb{S}_{NPE}(\mathtt{m})$). All test cases are annotated with the computed $\mathbb{S}_{TC}$ (Line 9 in Algorithm 2), and NpeTest performs weighted sampling based on the score $\mathbb{S}_{TC}$ to select the test case from the population to be mutated (Line 6).

4 EVALUATION

In this section, we experimentally evaluate NpeTest. to answer the following three research questions.

- **RQ1:** How effectively can NpeTest generate unit tests that detect the known NPEs? How does it compare to the existing unit test generators?
- **RQ2:** What is the correlation between code coverage and the ability to find NPEs? Is achieving high code coverage effective for NPE detection?
- **RQ3:** Can NpeTest actually help software developers willing to ensure the software quality in the industry?

Table 1: Benchmarks collected from prior works [3, 21–24]. $\text{Project}_{\text{rep}}$: # of reproducible projects in our experimental environment. NPE: # of known NPEs from reproducible projects $\text{Project}_{\text{rep}}$. NPE_{test}: # of NPEs occured in a test case itself. $\text{NPE}_{\text{outside}}$: # of NPEs triggered outside of the target project (e.g., Map library), $\text{Null}_{\text{test}}$: # of NPEs triggered by passing `null` directly to the argument of method, $\text{Null}_{\text{untrackable}}$: # of NPEs that is hard to track the source of Null, Duplicated: # of duplicated NPEs

				Excluded NPEs					Final benchmarks	
Source	Project	$\text{Project}_{\text{rep}}$	NPE	NPE_{test}	$\text{NPE}_{\text{outside}}$	$\text{Null}_{\text{test}}$	$\text{Null}_{\text{untrackable}}$	Duplicated	**Project**	**NPE**
NPEX [3]	59	59	65	0	0	7	1	0	**53**	**57**
Genesis [23]	16	14	14	0	2	3	0	0	**9**	**9**
Bears [21]	18	18	20	1	4	7	0	0	**8**	**8**
BugSwarm [22]	76	60	68	22	6	2	5	5	**21**	**28**
Defects4J [24]	26	10	11	0	1	2	0	2	**5**	**6**
Total	195	161	178	23	13	21	6	7	**96**	**108**

4.1 Evaluation Setting

We implemented NpeTest on top of the latest version of EvoSuite [10], which was last updated in February 2024 on GitHub at the time of writing. For performance comparison with existing unit test generation tools, we selected EvoSuite [10] and Randoop [9] as baselines, where both tools are widely adopted in the field of unit test generation with their exceptional performance. In particular, EvoSuite demonstrated its performance superiority at the International Workshop on Search-Based Software Testing (i.e., SBST), an annual unit test generation tool competition, winning 10 out of 12 competitions since 2013. Randoop, on the other hand, is a well-established test case generation tool built upon feedback-directed random testing. We conducted 25 evaluation experiments for each tool with a time budget of 5 minutes on the benchmark classes where NPEs occurred, using a total of 125 minutes of CPU time for each benchmark. All experiments were done on a Linux machine running Ubuntu 20.04 with 64 CPUs and 256GB memory, powered by AMD Ryzen Threadripper 3990X 64-Core Processor.

Options for EvoSuite. In addition to the essential parameters such as target class and time budget, users can also configure EvoSuite with over 30 optional parameters to fine-tune its performance. This presents challenges in finding optimal parameter settings, as it requires significant effort for tuning optimal parameters, which is beyond the scope of this paper. Therefore, we utilized the parameter settings set by the EvoSuite team[2], which are chosen for their demonstrated effectiveness in maximizing line/branch coverage and bug detection ratio. The EvoSuite team has used these settings to participate in the SBST competitions, where they have achieved success by winning the competition 10 times out of 12 from 2013 to 2024 [25–27]. Figure 3 shows the fine-tuned options we used for our evaluation.

Benchmarks. To evaluate the effectiveness of our technique on NPE detection, we collected real-world NPE benchmarks from the literature, which have been used in the prior works [3, 4, 21, 23]. These benchmark projects have been discussed in the recent APR (Automatic Program Repair) work that targets NPEs [3]. We also collected benchmark projects from a static analysis study [28], which are composed of 76 and 26 NPEs from BugSwarm [22] and Defects4J [24], respectively. All benchmarks are provided with the buggy version of each project, a test case triggering the NPEs, and the line number where NPE occurs. In total, we gathered 195 buggy projects that can be built with Maven.

Table 1 shows a detailed explanation of our benchmark selection. Specifically, we manually investigated each benchmark program with the given NPE-triggering test cases and excluded the benchmarks in which we failed to build or reproduce NPEs in our experimental environment. We also excluded NPEs that are not located in the source code of programs and whose null values originate from literal `null` in the test case or Java native code. After removing duplicated NPEs, we finally collected 96 buggy projects with 108 known NPEs for our benchmark suite.

4.2 Effectiveness of NpeTest

Table 2 shows the average reproduction rates over 25 trials to generate NPE-triggering test cases. As a main metric to evaluate bug-finding abilities of test case generation tools, we define the reproduction rate as the frequency with which the NPE is detected among a fixed number of iterations (25 trials in our experiment); that is, the reproduction rate represents the probability of tools to detect NPEs. We used this metric because the reproduction rate can provide more detailed information about the performance improvement for each unique bug. More specifically, using the reproduction rate makes it possible to identify which specific NPEs were detected more effectively (i.e., with a higher reproduction rate).

For simplicity, we excluded the results in a table when all three tools failed to detect the known NPEs among all trials, which remains 74 known NPEs in total. In summary, NpeTest, on average, succeeded to generate test cases detecting the known NPEs with 45.2% and 22.4% more reproduction rates than Randoop and EvoSuite, respectively. In terms of the number of NPEs detected in any of the 25 trials, NpeTest found 73 NPEs while Randoop and EvoSuite detected 25 and 59 NPEs, respectively. We computed the p-value for the Mann-Whitney U test on our experimental results, where the p-value was 0.0003. Hence, we could conclude that our results were statistically significant at $\alpha = 0.05$.

[2]https://www.youtube.com/watch?v=Vwxu6TtzBYs&t=19879s

Table 2: The reproduction rate results for 25 trials on benchmark projects gathered from the prior studies [3, 21–24]. The projects where all three tools failed to detect NPEs are excluded from this table. Project: The name of the buggy project with its abbreviated NPE-labeling ID (if necessary). The number in a parenthesis represents each unique NPE in the same project.

Project	Randoop	EvoSuite	NPETest	Project	Randoop	EvoSuite	NPETest	Project	Randoop	EvoSuite	NPETest
NPEX											
Activiti-c45	0	0	24	Hivemall-04fa	0	68	100	Log4j_2-7441	100	100	100
Aries_JPA	0	100	80	Http-f633(1)	0	0	88	Ninja-16aa	100	64	100
Avro	100	100	100	Http-f633(2)	0	0	76	Nutz-87a4	0	76	100
Commons_Conf	100	100	100	Http-f633(3)	0	72	84	OpenNLP-6079	0	100	100
Commons_DBCP	100	16	100	Http-f633(4)	0	72	84	OpenPDF-a89d	0	100	100
Commons_Pool	0	92	100	IoTDB-9bce	0	40	40	PDFBox-5558	100	100	100
CXF-2094	100	56	84	Jest-f34f	0	8	0	Qpid-0299 (1)	0	0	16
Directory	0	80	100	jsoup-8b83	100	40	100	Qpid-0299 (2)	0	0	88
Easy_Rules	0	64	100	jsoup-b841	0	100	100	Sharding-82b1	0	0	4
Fastjson-650a	0	8	64	JSqlParser	0	68	96	Sharding-9833	0	40	100
Feign-9c5a	0	0	28	Karaf-b92d	0	0	28	Sharding-c08f	0	8	8
FOP-10e0d1c2	100	40	100	Log4j_2-5b7b	0	0	16	ZooKeeper	0	16	32
Hessian_Lite	0	100	96	Log4j_2-6a23	100	100	100				
BugSwarm											
ACS_Commons	0	0	64	OkHttp-9361(2)	0	96	100	REST-1546(5)	100	100	100
Artemis_odb	100	100	76	Petergeneric	0	100	100	REST-2078	0	40	100
BungeeCord-1303	0	100	100	REST-1546(1)	0	28	88	Universal-1724(1)	100	0	56
Byte-1405	100	92	100	REST-1546(2)	100	20	92	Universal-1724(2)	100	0	64
Byte_Buddy-9579	100	100	100	REST-1546(3)	100	100	100	Universal-6766	0	64	72
Dubbo-4166	100	88	100	REST-1546(4)	100	100	100	Yamcs-1863	0	76	100
OkHttp-9361(1)	0	88	100								
Defects4J				Genesis				Bears			
Cli-30	0	28	100	Activiti-31c8	0	24	92	Bears-189	0	100	100
Cli-30	0	32	80	Checkstyle-be8	0	60	80	Bears-222	0	20	40
Csv-11	0	80	72	DataflowJavaSDK	0	0	8	Bears-56	100	100	100
Csv-9	100	72	92	JavaPoet-aee5	100	100	100	Bears-70	0	0	28
Math-70	100	100	100	Javaslang-0dab	100	100	100	Bears-88	0	100	92
Math-79	0	76	100	Jongo-9743	0	0	4				
								Average	**33.7%**	**56.9%**	**78.9%**

As shown in Table 2, NpeTest generally shows the highest reproduction rate over 70 NPEs benchmarks. For instance, on "Fastjson-650a" from NPEX benchmark suite, NpeTest could successfully generate NPE-detecting test-cases for 16 times over 25 trials (i.e., 16/25 = 0.64) while EvoSuite showed the reproduction rate of 8%. Randoop, however, failed to detect NPEs within the time-budget. On other benchmark program, "Hivemall-04fa", NpeTest succeeded to generate NPE-triggering test cases in all trials while EvoSuite was able to detect the corresponding NPE 17 out of 25 trials.

Surprisingly, Randoop, a random-based testing tool, shows the best performance on some of the NPE benchmarks. On "Commons_DBCP" from BugSwarm, for instance, Randoop successfully generated the test cases for detecting NPEs in all 25 trials when the reproduction rate of EvoSuite was only 16%. NpeTest could always detect the NPE in all 25 trials.

One interesting point from Table 2 is that Randoop shows either 100% or 0% reproduction rate for NPE detection on our benchmark suite. We observed that the benchmark programs which Randoop shows 100% are the cases where the NPEs are triggered with a trivial and simple method sequence.

```java
DriverAdapterCPDS c = new DriverAdapterCPDS();
c.toString();
```

For example, a test case above is an NPE-triggering test case for the benchmark project "Commons_DBCP". The constructor call in this test case take no argument, which makes it easy to synthesize. EvoSuite, however, frequently failed to generate such test case as shown in Table 2; The reproduction rate of EvoSuite for this project is 16%. This example shows that existing test case generators are not specially designed for efficient NPE detection.

Additionally, we investigated the cases where all the tools failed to detect NPEs, and observed that those cases are related to the restricted search space for test case generation. As an example, the test case below triggers NPE in the `PatriciaTrie` class from Apache Commons Collections revision 796114ea:

```java
1 Trie<String, Integer> trie = new PatriciaTrie<Integer>();
2 trie.put("aa", 1); trie.put("ab", 1);
3 SortedMap<String, Integer> prefixMap = trie.prefixMap("a");
4 prefixMap.clear();
```

This test case has an extra method call (`clear` at line 4) with a return value of MUT (`prefixMap` at line 3). The `clear` method is a virtual method defined in Java native library, but it indirectly calls a method belonging to an inner class of `PatriciaTrie`, which triggers an NPE inside the inner class class of `PatriciaTrie`. If a test case generator contains the methods from the Java native library for test case generation, this NPE-triggering test case can be synthesized somehow. It, however, will result in intractably large search space, which might negatively affect the overall performance.

Figure 4 illustrates the number of unique NPEs detected by each tool. As shown in the Figure 4, NpeTest could find 14 more NPEs

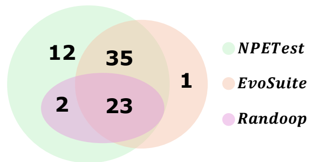

Figure 4: A Venn diagram illustrating the number of unique NPEs found by each tool.

Table 3: The number of unique NPEs detected by each unit test generation tool with different time budgets.

Time	Tools	NPEX	BugSwarm	Defects4J	Genesis	Bears	Total
5 min	Randoop	10	10	2	2	1	25
	EvoSuite	29	16	6	4	4	59
	NPETest	37	19	6	6	5	73
10 min	Randoop	10	10	2	2	1	25
	EvoSuite	29	16	6	5	5	61
	NPETest	37	19	6	6	6	74
30 min	Randoop	10	10	2	2	1	25
	EvoSuite	29	17	6	6	5	62
	NPETest	37	19	6	6	6	74

Table 4: Bug-finding ability of EvoSuite with different settings. EvoSuite$_{Def}$: EvoSuite with no fine-tuned options.

Tool	NPEX	BugSwarm	Defects4J	Genesis	Bears	Total
EvoSuite	50.7%	68.0%	64.7%	47.3%	64.0%	56.9%
EvoSuite$_{Def}$	48.8%	62.7%	83.3%	45.3%	60.0%	55.7%

which EvoSuite failed to detect. However, NpeTest could not generate any test cases triggering the known NPE in "Jest-f34f" benchmark from NPEX while EvoSuite could detect the corresponding NPE. The interesting point is that the reproduction rate of EvoSuite for this benchmark was only 8%, which means that EvoSuite also struggled to generate NPE-triggering test cases. We investigate the case and observed that imprecision of our static analyzer misdirected NpeTest into focusing on other methods rather than the one in which the known NPE occurs. NpeTest failed to detect NPE in "Jest-f34f" due to both incorrect instructions and difficulties in generating NPE-triggering test cases.

Performance Across Varying Time Budgets. Table 3 represents the number of unique bugs detected by each tool when different time-budget is given. As shown in the table, extending the testing time may enhance the NPE detection ability of the unit test generation tool; however, the improvement remains relatively modest. In particular, the results of Randoop remained the same regardless of the time budgets, and NpeTest and EvoSuite could successfully detect 1 and 3 more NPEs when longer time budget is given. Additionally, the reproduction rates of the newly detected NPEs were not particularly high. For instance, the reproduction rate of an NPE in "Bears-46", detected with an extended time budget, was below 20% for both NpeTest and EvoSuite. These experimental results indicate that the correlation between longer testing time and NPE detection is insignificant, highlighting the need for NPE-guided approaches to effectively detect NPEs.

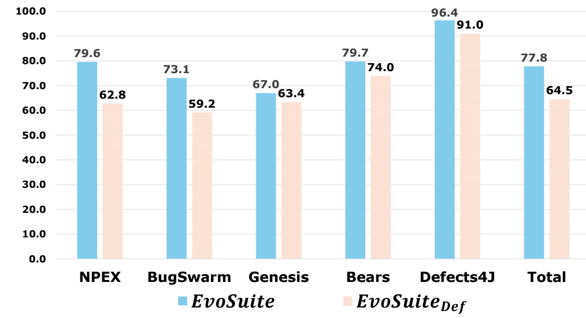

Figure 5: A graph for an average line coverage of EvoSuite and EvoSuite$_{Def}$ on each benchmark suite.

4.3 Correlation between Code Coverage and NPE Detection

To observe the correlation between code coverage and NPE detection ability, we evaluated EvoSuite with different options. More specifically, EvoSuite is evaluated with the (highly tuned) options we discussed in Section 4.1 while EvoSuite$_{Def}$ uses the default options without tuning. Likely to Table 2, we additionally evaluated EvoSuite$_{Def}$ with 25 trials and a time budget of 5 minutes for each benchmark. Figure 5 and Table 4 demonstrate the performance difference when using EvoSuite with different options in terms of code coverage and reproduction rate of NPEs, respectively. We chose *line coverage*, which is fundamentally measured by EvoSuite, as the metric for code coverage comparison.

As demonstrated in Figure 5, the fine-tuned option could greatly enhance the performance of code coverage. EvoSuite achieved line coverage of 77.8% on average while EvoSuite$_{Def}$ showed 64.5%, which is a 20.8% improvement. In contrast, considering the reproduction rate for NPE detection, the use of highly tuned options in EvoSuite yielded only a mere improvement, as shown in Table 4. More precisely, the tuned options increased the reproduction rate from 55.7% to 56.9%, representing a 2.2% improvement.

Code Coverage of NpeTest. Interestingly, NpeTest achieved less code coverage than EvoSuite and EvoSuite$_{Def}$ while it showed the best performance on NPE detection in Table 2. Even with specialized fine-tuning options to achieve high coverage, NpeTest showed 1.3% less line coverage compared to EvoSuite$_{Def}$. The reason for low line coverage is that NpeTest has smaller search space (i.e., methods under test) than EvoSuite. First, as described in Section 3.2, NpeTest analyzes all methods of a given target class, distinguishes those without NPE candidates, and then removes them from a set of methods under test (MUT). Therefore, NpeTest never tests the NPE-free methods unless they are called from other NPE-prone methods. For example, on a program "ZooKeeper" from NPEX benchmark, NpeTest achieved 40.4% line coverage while EvoSuite achieved 59.3%. The number of public methods is 116, and NpeTest initially maintained 89 methods which is 76.8% out of total MUTs (i.e., method under test) while EvoSuite has 116 methods. Second, NpeTest monitors the execution result of each generated test cases and eliminates any MUT in which all NPE candidates are detected (Section 3.3). NpeTest performs this elimination regardless of the remaining code regions in methods to cover, which leads to lower line coverage than EvoSuite.

4.4 Industrial Case Study

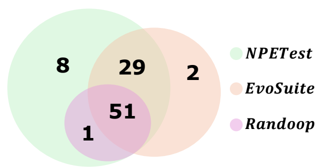

Figure 6: A Venn diagram illustrating the number of unique NPEs found by each tool on an industrial project.

To compare the practical feasibility of the three subject tools, we conducted a case study focusing on a proprietary cryptographic library used within an IT company. We selected this library for its moderate complexity and comprehensive development and testing standards. It consists of 84 public classes and 13,669 lines of code, with a 76% line coverage achieved through manually written unit tests. The library is utilized for critical security functions, such as database encryption, and is certified under ISO/IEC 15408 (CC certification) [29].

The development and testing processes of the library included:

- **Unit Testing and Coverage**: Ensuring functionality validation and achieving targeted line/branch coverage through unit tests.
- **Peer Code Review**: Rigorous review of each pull request with active developer participation.
- **Code Quality Metrics**: Employing metrics like Cyclomatic Complexity [30] to keep code maintainable.

The tools NpeTest, EvoSuite, and Randoop were executed on the library, allocating 5 minutes per class. To address the tools' non-deterministic nature, the experiment was repeated 25 times. Surprisingly, the tools revealed a total of 91 previously unknown NPEs, all of which were confirmed as true positives by the library development team after reviewing the stack traces and the unit tests generated by the tools. Figure 6 illustrates the number of NPEs detected by each tool. NpeTest found 89 NPEs, including 9 that EvoSuite missed and 37 Randoop missed. On the other hand, EvoSuite and Randoop detected 82 and 52 NPEs, respectively, with EvoSuite finding only 2 additional NPEs not detected by NpeTest. The result corroborates our previous evaluation results: *NpeTest is also more effective at detecting NPEs within the industrial library than the other tools.*

Figure 7 shows an NPE of the subject library detected by NpeTest. To avoid exposing proprietary code, we anonymized it. An NPE at Line 15 was detected through a unit test generated by NpeTest. This exception arose at line 15 due to the improper handling of the return value from `encodings.get(getNodeText(doc,ENCODING))`. This value was passed to the `setEncodingType` method defined at line 8. Specifically, since the return value of `getNodeText(doc, ENCODING)` was not in the HashMap `encodings`, `encodings.get` returned `null`. Consequently, `setEncodingType(`*int*`)` received `null` as an argument, which triggered the NPE.

Since `serverMsg` comes from a remote server, the running application should validate `serverMsg` before using it. Without validation, an invalid `serverMsg` could compromise the running application's functionality and system integrity. In this case, the invalid

```
1
2   class CryptoPolicySVO {
3     static Map<String, Integer> encodings = new HashMap<>();
4     static {
5       encodings.put("BASE64", 1); encodings.put("HEX", 2); }
6     ...
7     private int encoding;
8     private void setEncodingType(int encoding) {
9       this.encoding = encoding; }
10
11    public CryptoPolicySVO deserialize(String serverMsg) {
12      Document doc = parseXML(serverMsg);
13      CryptoPolicySVO svo = new CryptoPolicySVO();
14      svo.setAlogithmType(getNodeText(doc, ALGORITHM));
15      svo.setEncodingType(encodings.get(getNodeText(doc,
            ENCODING))); // triggers NPE
16      ...
17      return svo; }
18  }
```

Figure 7: An NPE example from the industrial case study.

encoding triggers the NPE, causing the application to terminate abnormally. Furthermore, this NPE reveals a security risk due to inadequate validity checks on a security policy, which an attack could exploit by sending maliciously crafted policies to gain unauthorized access and compromise system integrity. We believe this case study demonstrates NpeTest's ability to detect critical errors that may have been overlooked during the development process.

Although all 91 NPEs were confirmed as true positives by the development team, the number of detected NPEs was surprising given the rigorous development process. To understand why such errors were not caught during development, we interviewed the development team and obtained several reasons for these oversights.

Firstly, developers sometimes intentionally omit `null` checks at the beginning of every method, considering this practice inefficient. However, this can lead to vulnerabilities, especially in public methods. Secondly, to achieve unit test coverage, private methods were exposed as public. This was based on the assumption that the library users would not misuse them, coupled with a misplaced belief that their mandatory obfuscation tool would prevent misuse by obfuscating the names of those former private methods, making them hard to find. This practice can expose critical functionalities, compromising security. Finally, developers expected users to strictly follow the example code scenarios, underestimating the users' creativity. This oversight does not account for the myriad ways users might employ the code, leading to unanticipated vulnerabilities.

Although these practices were likely adopted due to the complexities of developing software within an industrial setting, we recommended the following changes to prevent recurrence of these issues.

- Validate parameters in public methods. At least, use annotations like `@NonNull`, which assist IDEs and static analyzers in identifying potential NPEs without impacting runtime performance.
- Maintain private methods' visibility. Exposing them solely for testing compromises design integrity. If target coverage is not achievable through public methods, explore possibilities of unreachable code or engage in further QA discussions, rather than considering a redesign.

- Clearly define method contracts in Javadoc, outlining valid usage scenarios such as context initialization and method call sequence. Implement internal state tracking to ensure adherence to these guidelines.
- Encourage the use of `Optional`, empty strings, empty collections, or the null object pattern instead of returning `null` to avoid null usage.

Through the case study and our recommendations for addressing the identified issues, the developers have not only acknowledged an improvement in their understanding on secure coding practices but also plan to incorporate NpeTest into the development pipeline, thereby enhancing overall code security and robustness.

5 DISCUSSION

5.1 Lessons Learned

We now summarize lessons learned from our experiments on the NPE detection capabilities of state-of-the-art unit test generators.

Lesson 1 - The current unit test generators are not sufficient for NPE detection. Unit testing techniques have been widely used to verify that each unit of software behaves as expected or to detect bugs. However, as shown in RQ1, EvoSuite failed to detect NPEs that could easily be detected by the random testing tool Randoop [9], and only detected 59 out of 108 NPEs in total. NpeTest, which utilizes static and dynamic analyses on EvoSuite, could detect 73 unique NPEs. These experimental results show that the current unit test generators are not sufficient for NPE detection, and adopting an additional approach like NpeTest can significantly improve bug detection capability.

Lesson 2 - Achieving high code coverage is not necessary to improve NPE detection capability. The recent technical studies [31, 32] in unit testing techniques strive for high code coverage with the expectation that it will detect more bugs. Achieving high code coverage, however, does not always result in better NPE-detection ability. As shown in RQ2, while fine-tuned option parameters of EvoSuite increased the achieved line coverage by 20.8%, it could only improve the reproduction rate of NPE detection by 2.2%. In contrast, NpeTest achieved 18.8% less line coverage than EvoSuite, but was able to detect 15 more unique NPEs that EvoSuite failed to find, and showed a 22.4% higher reproduction rate on average. In line with our observation, several studies [33–35] in the field of software testing, including fuzzing techniques, have claimed that code coverage has an insignificant correlation with bug detection. Along with these studies, our experimental results also indicate that achieving high code coverage does not necessarily contribute to the improvement of NPE detection capability.

Lesson 3 - The importance of an integrated approach to detecting NPEs in industrial software development. The case study emphasizes the importance of adopting a comprehensive approach to detecting NPEs in industrial software development. Despite the rigorous testing and development process in place, the three subject tools were able to detect a significant number of previously unknown NPEs, highlighting the limitations of manual testing and code reviews. Furthermore, our interviews revealed that certain development practices, although well-intentioned, can inadvertently introduce vulnerabilities. By integrating automated testing tools like NpeTest into the development pipeline, in addition to the manual testing and code reviews, we believe that developers can proactively identify and address potential NPEs.

5.2 Limitation

In addition to its low code coverage described in Section 4.3, NpeTest has inherent limitations in detecting other types of bugs than NPEs. More precisely, through static and dynamic analysis, NpeTest intentionally skips testing methods that are free of NPEs or those for which all NPEs have been detected throughout the testing process, potentially missing bugs that exist in the skipped methods. We investigated Java standard runtime exceptions detected by NpeTest and EvoSuite, and found that both tools found 6 runtime exception types (e.g., "NegativeArraySizeException"). Although the number of different bug types discovered by each tool was the same, we observed that NpeTest failed to detect certain non-NPE bugs which EvoSuite successfully identified. For example, considering the "ArrayIndexOutOfBounds" runtime exception, NpeTest only detected 25 bugs while EvoSuite found 32 bugs. Among the exceptions NpeTest failed to detect, three exceptions occurred in NPE-free methods that were not the testing targets of NpeTest. Other exceptions arose in methods where NpeTest ceased generating test cases during testing because it had detected all identified NPEs in those methods. Given that NpeTest is specialized and focused on NPE detection, we consider its limitations in non-NPE bug detection and low code coverage to be manageable, especially in light of the improvements made in NPE detection.

5.3 Threats to Validity

(1) To evaluate the best performance of EvoSuite, we used a set of fine-tuned options from SBST'22 [25]. Because we conducted our evaluations on a set of benchmark programs different from the one used in SBST'22, these values for options may not be appropriate to achieve the best performance of EvoSuite on some of our benchmarks. However, note that finding optimal parameter settings of EvoSuite for each benchmark is a challenging task [26] and is beyond the scope of this paper. (2) As we mentioned in Section 4.1, we eliminated the programs we failed to build in our experiment settings. The experiment results may become different from what we have observed in our experiment if those programs were properly built and used for our evaluation. (3) We conducted our experiments for 5 minutes on each benchmark with 25 trials. The time-budget for experiments may not be sufficient to achieve the best performance of both EvoSuite and NpeTest.

6 RELATED WORK

Static Analysis to Find NPEs. Static analysis has been extensively studied to find and prevent bugs in development process. As one of the most common and fatal errors in Java applications, much research on detecting NPEs with static analysis approach have been introduced [5, 36–45]. Based on their approaches to statically analyze NPEs, these researches can be classified into two groups [28]: static bug detectors and type-based null safety checkers.

Static bug detectors designed for NPEs mainly use dataflow analysis to find null dereferences [36–42]. For example, INFER [36] is the static analyzer that uses bi-abduction analysis to detect null pointer dereferences, memory leaks, etc. INFER performs disjunctive analysis with maintaining limited set of disjunctive paths. Similar to INFER, SPOTBUGS [37] also utilizes dataflow analysis tailored for Java programs, but it also uses pattern matching algorithms for error detection.

Type-based null safety checkers leverage a *pluggable type system*, which is implemented via annotation syntax in Java [5, 43–45]. For instance, NULLAWAY [5] detects any possible violations of nullability annotation (e.g., whether nullable values can be passed to the method parameter annotated with `@NonNull`). NULLAWAY is an open source tool and being maintained by `Uber`, allowing users to set configuration that affects the assumption of *nullability* of unannotated variables.

Our work also performs static analysis based on dataflow analysis, to prioritize NPE-prone methods in test generation. Our focus, however, differs from the other static bug detectors such as INFER, which focus on locating exact bug location. Our static analysis is to calculate the scores of each method based on *how much the method is exposed to NPE-related paths*, which states that NPEs may occur in the method.

Unit Test Generation. Unit testing is a crucial step of the software development process, ensuring the reliability and robustness of software systems. To streamline unit testing, many studies have presented various approaches for automatically generating test cases by synthesizing arbitrary method call sequences.

Random testing explores the vast search space of possible method sequences in a random manner. The search space consists of the parameter spaces of each method, possible values and API sequences to create an instance of each type [46]. In theory, exhaustive search can be achieved through backtracking, covering the entire search space. Our approach is also closely related to this line of approach in that we heuristically define parameter search space in seed test generation. One of the most representative tools is feedback-guided random testing tool RANDOOP [9], which uses dynamic feedback from contract checkers to maintain valid object pools.

On the other hand, search-based testing not only generates method sequences randomly, but also applies optimizations to guide test code generations towards achieving more coverage goals. SBST approaches formulate test genearation problems as optimization problems and employ genetic algorithms to solve them. EVOSUITE [10], a widely-used SBST tool, incorporates several sophisticated genetic algorithms for test code generations [10–12, 47]. Over the past decade, EVOSUITE has undergone significant evolution, driven by researchers and practitioners. Notably, DynaMOSA, the current default algorithm of EVOSUITE and an evaluation subject of this paper, has achieved the highest code coverage and bug detection ratio (i.e, mutation kill ratio) among the presented genetic algorithms for test code generation [11].

In addition, Rojas et al. [48] has investigated the impact of seeding strategies on SBST, with large-scale empirical analysis [48]. We adopt several key concepts such as seeding strategy introduced in the research aiming to improve SBST, especially for NPE detection.

Static Analysis-Guided Unit Test Generation. Static analysis has been used to enhance test generation algorithm. Especially, for NPE, Romano et al. [49] took search-based approach to find NPE, which focuses on covering NPE-related paths obtained by static null dereference analysis designed by Nanda and Sinha [41]. Particularly, the search-based approach is used to generate test inputs, most of which are of primitive types. Thus, this work does not consider the situation where creation of fresh object states is required to trigger NPEs, which our approach does.

EvoObj [50] uses static analysis to facilitate complex object creation and evaluates effectiveness by improving EVOSUITE in terms of branch coverage. In random testing context, Ma et al. [51] uses static analyses to improve RANDOOP. For example, it, prior to test generation, performs impurity analysis to find methods that can alter the states of the objects, and uses them when selecting methods to mutate test cases. Ma et al. [51] implemented GRT (Guided Random Testing), an extended version of RANDOOP leveraging static and dynamic analyses. Our work is similar to GRT in that we performs pre-analysis before generating test, but our analysis is focusing on finding NPE-prone program paths.

7 CONCLUSION

Null pointer exceptions (NPEs) are one of the most common errors in Java applications. Although several approaches have been proposed to generate test cases for Java, finding NPEs via testing still remains a significant challenge despite its importance. In this experience paper, we presented NPETEST, a strategy for unit testing techniques specialized for NPE detection. To this end, we proposed static and dynamic analysis techniques, guiding the test case generators to find test cases which explore NPE-prone regions more efficiently. Experimental results demonstrated that NPETEST can significantly improve the NPE-detection ability of the state-of-the-art unit test generators.

DATA AVAILABILITY

The artifact of NPETEST is publicly available in Zenodo [3] and GitHub[4]. It includes all benchmarks, tools, and the experimental results in Section 4, except the industrial case study.

ACKNOWLEDGEMENT

This work was supported by Samsung SDS, Institute of Information & communications Technology Planning & Evaluation (IITP) grant funded by the Korea government(MSIT) (No.2020-0-01337,(SW STAR LAB) Research on Highly-Practical Automated Software Repair, 40%), ICT Creative Consilience Program through the Institute of Information & Communications Technology Planning & Evaluation(IITP) grant funded by the Korea government(MSIT) (IITP-2024-2020-0-01819, 10%), the National Research Foundation of Korea(NRF) grant funded by the Korea government(MSIT)(No. 2021R1A5A1021944, 20%), and the Institute of Information & Communications Technology Planning & Evaluation(IITP) grant funded by the Korea government(MSIT) (No.RS-2022-II220277, Development of SBOM Technologies for Securing Software Supply Chains, 10%).

[3] https://doi.org/10.5281/zenodo.13371822
[4] https://github.com/kupl/NPETestArtifact

REFERENCES

[1] R. Coelho, L. Almeida, G. Gousios, A. van Deursen, Unveiling exception handling bug hazards in android based on github and google code issues, in: 2015 IEEE/ACM 12th Working Conference on Mining Software Repositories, 2015, pp. 134–145. doi:10.1109/MSR.2015.20.

[2] S. H. Tan, Z. Dong, X. Gao, A. Roychoudhury, Repairing crashes in android apps, in: Proceedings of the 40th International Conference on Software Engineering, 2018, pp. 187–198.

[3] J. Lee, S. Hong, H. Oh, Npex: Repairing java null pointer exceptions without tests, in: 2022 IEEE/ACM 44th International Conference on Software Engineering (ICSE), 2022, pp. 1532–1544. doi:10.1145/3510003.3510186.

[4] X. Xu, Y. Sui, H. Yan, J. Xue, Vfix: Value-flow-guided precise program repair for null pointer dereferences, in: Proceedings of the 41st International Conference on Software Engineering, ICSE '19, IEEE Press, 2019, p. 512–523. URL: https://doi.org/10.1109/ICSE.2019.00063. doi:10.1109/ICSE.2019.00063.

[5] S. Banerjee, L. Clapp, 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, Association for Computing Machinery, New York, NY, USA, 2019, p. 740–750. URL: https://doi.org/10.1145/3338906.3338919. doi:10.1145/3338906.3338919.

[6] N. Karimipour, J. Pham, L. Clapp, M. Sridharan, Practical inference of nullability types, in: Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2023, Association for Computing Machinery, New York, NY, USA, 2023, p. 1395–1406. URL: https://doi.org/10.1145/3611643.3616326. doi:10.1145/3611643.3616326.

[7] A. Zhitnitsky, The top 10 exception types in production java applications – based on 1b events, 2016. https://www.harness.io/blog/10-exception-types-in-production-java-applications.

[8] N. Andrews, We crunched 1 billion java logged errors – here's what causes 97% of them, 2021. https://www.overops.com/blog/we-crunched-1-billion-java-logged-errors-heres-what-causes-97-of-them-2/.

[9] C. Pacheco, S. K. Lahiri, M. D. Ernst, T. Ball, Feedback-directed random test generation, in: 29th International Conference on Software Engineering (ICSE'07), 2007, pp. 75–84. doi:10.1109/ICSE.2007.37.

[10] G. Fraser, A. Arcuri, Whole test suite generation, IEEE Transactions on Software Engineering 39 (2013) 276–291. doi:10.1109/TSE.2012.14.

[11] A. Panichella, F. M. Kifetew, P. Tonella, Automated test case generation as a many-objective optimisation problem with dynamic selection of the targets, IEEE Transactions on Software Engineering 44 (2018) 122–158. doi:10.1109/TSE.2017.2663435.

[12] A. Panichella, F. M. Kifetew, P. Tonella, Reformulating branch coverage as a many-objective optimization problem, in: 2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST), 2015, pp. 1–10. doi:10.1109/ICST.2015.7102604.

[13] G. Fraser, M. Staats, P. McMinn, A. Arcuri, F. Padberg, Does automated white-box test generation really help software testers?, in: Proceedings of the 2013 International Symposium on Software Testing and Analysis, ISSTA 2013, Association for Computing Machinery, New York, NY, USA, 2013, p. 291–301. URL: https://doi.org/10.1145/2483760.2483774. doi:10.1145/2483760.2483774.

[14] G. Fraser, A. Arcuri, A large-scale evaluation of automated unit test generation using evosuite, ACM Trans. Softw. Eng. Methodol. 24 (2014). URL: https://doi.org/10.1145/2685612. doi:10.1145/2685612.

[15] S. Shamshiri, R. Just, J. M. Rojas, G. Fraser, P. McMinn, A. Arcuri, Do automatically generated unit tests find real faults? an empirical study of effectiveness and challenges (t), in: 2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE), 2015, pp. 201–211. doi:10.1109/ASE.2015.86.

[16] J. M. Rojas, M. Vivanti, A. Arcuri, G. Fraser, A detailed investigation of the effectiveness of whole test suite generation, Empirical Software Engineering (2016) 1–42. URL: http://dx.doi.org/10.1007/s10664-015-9424-2. doi:10.1007/s10664-015-9424-2.

[17] M. M. Almasi, H. Hemmati, G. Fraser, A. Arcuri, J. Benefelds, An industrial evaluation of unit test generation: Finding real faults in a financial application, in: 2017 IEEE/ACM 39th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP), 2017, pp. 263–272. doi:10.1109/ICSE-SEIP.2017.27.

[18] Investigating faults missed by test suites achieving high code coverage, Journal of Systems and Software 144 (2018) 106–120.

[19] H. Hemmati, How effective are code coverage criteria?, in: 2015 IEEE International Conference on Software Quality, Reliability and Security, 2015, pp. 151–156. doi:10.1109/QRS.2015.30.

[20] P. S. Kochhar, D. Lo, J. Lawall, N. Nagappan, Code coverage and postrelease defects: A large-scale study on open source projects, IEEE Transactions on Reliability 66 (2017) 1213–1228. doi:10.1109/TR.2017.2727062.

[21] F. Madeiral, S. Urli, M. Maia, M. Monperrus, Bears: An extensible java bug benchmark for automatic program repair studies, 2019, pp. 468–478. doi:10.1109/SANER.2019.8667991.

[22] D. A. Tomassi, N. Dmeiri, Y. Wang, A. Bhowmick, Y. Liu, P. T. Devanbu, B. Vasilescu, C. Rubio-González, Bugswarm: mining and continuously growing a dataset of reproducible failures and fixes, in: ICSE, IEEE / ACM, 2019, pp. 339–349.

[23] F. Long, P. Amidon, M. Rinard, Automatic inference of code transforms for patch generation, in: Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2017, Association for Computing Machinery, New York, NY, USA, 2017, p. 727–739. URL: https://doi.org/10.1145/3106237.3106253. doi:10.1145/3106237.3106253.

[24] R. Just, D. Jalali, M. D. Ernst, 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, Association for Computing Machinery, New York, NY, USA, 2014, p. 437–440. URL: https://doi.org/10.1145/2610384.2628055. doi:10.1145/2610384.2628055.

[25] S. Schweikl, G. Fraser, A. Arcuri, Evosuite at the sbst 2022 tool competition, in: 2022 IEEE/ACM 15th International Workshop on Search-Based Software Testing (SBST), 2022, pp. 33–34. doi:10.1145/3526072.3527526.

[26] A. Arcuri, G. Fraser, Parameter tuning or default values? an empirical investigation in search-based software engineering, Empirical Software Engineering 18 (2013) 594–623.

[27] J. M. Rojas, J. Campos, M. Vivanti, G. Fraser, A. Arcuri, Combining multiple coverage criteria in search-based unit test generation, in: Search-Based Software Engineering: 7th International Symposium, SSBSE 2015, Bergamo, Italy, September 5-7, 2015, Proceedings 7, Springer, 2015, pp. 93–108.

[28] D. A. Tomassi, C. Rubio-González, On the real-world effectiveness of static bug detectors at finding null pointer exceptions, in: 2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE), 2021, pp. 292–303. doi:10.1109/ASE51524.2021.9678535.

[29] ISO/IEC, ISO/IEC 15408-1:2009, 2009. https://www.iso.org/standard/50341.html.

[30] A. H. Watson, T. J. McCabe, Structured testing: A testing methodology using the cyclomatic complexity metric, 1996. http://www.mccabe.com/pdf/mccabe-nist235r.pdf.

[31] Y. Lin, Y. S. Ong, J. Sun, G. Fraser, J. S. Dong, Graph-based seed object synthesis for search-based unit testing, ESEC/FSE 2021, Association for Computing Machinery, New York, NY, USA, 2021, p. 1068–1080. URL: https://doi.org/10.1145/3468264.3468619. doi:10.1145/3468264.3468619.

[32] C. Lemieux, J. P. Inala, S. K. Lahiri, S. Sen, Codamosa: Escaping coverage plateaus in test generation with pre-trained large language models, in: 2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE), IEEE, 2023, pp. 919–931.

[33] L. Inozemtseva, R. Holmes, Coverage is not strongly correlated with test suite effectiveness, in: Proceedings of the 36th International Conference on Software Engineering, ICSE 2014, Association for Computing Machinery, New York, NY, USA, 2014, p. 435–445. URL: https://doi.org/10.1145/2568225.2568271. doi:10.1145/2568225.2568271.

[34] P. S. Kochhar, D. Lo, J. Lawall, N. Nagappan, Code coverage and postrelease defects: A large-scale study on open source projects, IEEE Transactions on Reliability 66 (2017) 1213–1228. doi:10.1109/TR.2017.2727062.

[35] M. Böhme, L. Szekeres, J. Metzman, On the reliability of coverage-based fuzzer benchmarking, in: Proceedings of the 44th International Conference on Software Engineering, ICSE '22, Association for Computing Machinery, New York, NY, USA, 2022, p. 1621–1633. URL: https://doi.org/10.1145/3510003.3510230. doi:10.1145/3510003.3510230.

[36] F. Inc., A tool to detect bugs in java and c/c+++/objective-c code before it ships, 2022. Available: https://fbinfer.com.

[37] spotbugs community, Spotbugs, 2023. Available: https://spotbugs.github.io.

[38] D. Hovemeyer, W. Pugh, Finding bugs is easy, SIGPLAN Not. 39 (2004) 92–106. URL: https://doi.org/10.1145/1052883.1052895. doi:10.1145/1052883.1052895.

[39] A. Loginov, E. Yahav, S. Chandra, S. Fink, N. Rinetzky, M. Nanda, Verifying dereference safety via expanding-scope analysis, in: Proceedings of the 2008 International Symposium on Software Testing and Analysis, ISSTA '08, Association for Computing Machinery, New York, NY, USA, 2008, p. 213–224. URL: https://doi.org/10.1145/1390630.1390657. doi:10.1145/1390630.1390657.

[40] R. Madhavan, R. Komondoor, Null dereference verification via over-approximated weakest pre-conditions analysis, in: Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications, OOPSLA '11, Association for Computing Machinery, New York, NY, USA, 2011, p. 1033–1052. URL: https://doi.org/10.1145/2048066.2048144. doi:10.1145/2048066.2048144.

[41] M. G. Nanda, S. Sinha, Accurate interprocedural null-dereference analysis for java, in: Proceedings of the 31st International Conference on Software Engineering, ICSE '09, IEEE Computer Society, USA, 2009, p. 133–143. URL: https://doi.org/10.1109/ICSE.2009.5070515. doi:10.1109/ICSE.2009.5070515.

[42] I. Dillig, T. Dillig, A. Aiken, Static error detection using semantic inconsistency inference, SIGPLAN Not. 42 (2007) 435–445. URL: https://doi.org/10.1145/1273442.1250784. doi:10.1145/1273442.1250784.

[43] F. Inc., Eradicate | infer, 2022. Available: https://fbinfer.com/docs/checker-eradicate.

[44] W. Dietl, S. Dietzel, M. D. Ernst, K. Muslu, T. W. Schiller, Building and using pluggable type-checkers, in: 2011 33rd International Conference on Software Engineering (ICSE), 2011, pp. 681–690. doi:10.1145/1985793.1985889.

[45] M. M. Papi, M. Ali, T. L. Correa, J. H. Perkins, M. D. Ernst, Practical pluggable types for java, in: Proceedings of the 2008 International Symposium on Software Testing and Analysis, ISSTA '08, Association for Computing Machinery, New York, NY, USA, 2008, p. 201–212. URL: https://doi.org/10.1145/1390630.1390656. doi:10.1145/1390630.1390656.

[46] C. Csallner, Y. Smaragdakis, Jcrasher: An automatic robustness tester for java, Softw., Pract. Exper. 34 (2004) 1025–1050. doi:10.1002/spe.602.

[47] A. Arcuri, Many independent objective (mio) algorithm for test suite generation, in: Search Based Software Engineering: 9th International Symposium, SSBSE 2017, Paderborn, Germany, September 9-11, 2017, Proceedings 9, Springer, 2017, pp. 3–17.

[48] J. M. Rojas, G. Fraser, A. Arcuri, Seeding strategies in search-based unit test generation, Software Testing, Verification and Reliability 26 (2016) 366–401. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/stvr.1601. doi:https://doi.org/10.1002/stvr.1601. arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/stvr.1601.

[49] D. Romano, M. Di Penta, G. Antoniol, An approach for search based testing of null pointer exceptions, in: 2011 Fourth IEEE International Conference on Software Testing, Verification and Validation, 2011, pp. 160–169. doi:10.1109/ICST.2011.49.

[50] Y. Lin, Y. S. Ong, J. Sun, G. Fraser, J. S. Dong, Graph-based seed object synthesis for search-based unit testing, in: Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2021, Association for Computing Machinery, New York, NY, USA, 2021, p. 1068–1080. URL: https://doi.org/10.1145/3468264.3468619. doi:10.1145/3468264.3468619.

[51] L. Ma, C. Artho, C. Zhang, H. Sato, J. Gmeiner, R. Ramler, Grt: Program-analysis-guided random testing (t), in: 2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE), 2015, pp. 212–223. doi:10.1109/ASE.2015.49.