NPEX: Repairing Java Null Pointer Exceptions without Tests

Junhee Lee
Korea University
Republic of Korea
junhee_lee@korea.ac.kr

Seongjoon Hong∗
Korea University
Republic of Korea
seongjoon@korea.ac.kr

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

ABSTRACT
We present NPEX, a new technique for repairing Java null pointer exceptions (NPEs) without tests. State-of-the-art NPE repair techniques rely on test suites written by developers for patch validation. Unfortunately, however, those are typically future test cases that are unavailable at the time bugs are reported or insufficient to identify correct patches. Unlike existing techniques, NPEX does not require test cases; instead, NPEX automatically infers the repair specification of the buggy program and uses the inferred specification to validate patches. The key idea is to learn a statistical model that predicts how developers would handle NPEs by mining null-handling patterns from existing codebases, and to use a variant of symbolic execution that can infer the repair specification from the buggy program using the model. We evaluated NPEX on real-world NPEs collected from diverse open-source projects. The results show that NPEX significantly outperforms the current state-of-the-art.

ACM Reference Format:

1 INTRODUCTION
Null pointer exceptions (NPEs) are perhaps the most infamous bug in Java. NPEs represent a serious flaw of a program because dereferencing a null pointer always causes the program to crash. Furthermore, NPEs are highly prevalent in real-world Java applications [2, 8, 9, 32, 55, 72]. For example, NPEs take up 37.2% and 40.2% of crashes in open-source projects [32] and Android applications [55], respectively, and recent studies show that NPEs are the most prevailing uncaught exception in production environments [2, 72]. Yet, fixing NPEs remains challenging because simply avoiding crashes is often incorrect and finding a correct fix out of a wide range of candidates is nontrivial.

Our Approach. In this paper, we present a new approach, called NPEX, for repairing NPEs. Like existing APR techniques, NPEX follows the standard generate-and-validate approach. The difference, though, is that NPEX replaces the test-based patch validation phase of the existing approach by a novel technique that can validate patches without relying on test cases.

We avoid using test cases as a validation oracle for two reasons. First, because test cases are typically unavailable at the time a bug is reported [25], they cannot be effectively used by a repair tool that aims to fix the bug as soon as it is detected. Furthermore, using a test suite as a repair specification is likely to produce incorrect patches fitted only to the given tests [28, 54, 69]. For example, as we demonstrate in this paper, even the state-of-the-art VFIX, which reduces incorrect patches using a customized patch space, often fails to fix diverse NPEs due to overfitting.

We use two key ideas to validate NPE patches without tests. First, we use a statistical model that predicts how developers would handle NPEs. To learn such a model, we collect various null-handling patterns available in existing codebases. For example, from null-handling code \((x != null)? x.m() : 0\), we extract the knowledge that the expected return value of \(x.m()\) is 0 when \(x\) is null and therefore an NPE occurs at \(x.m()\). We then generalize this pattern using program-independent features for expressions and surrounding contexts. Second, we use the model to infer the expected behavior of a buggy program. To this end, we use a variant of symbolic execution that interprets NPE-triggering expressions in the buggy program using the model’s prediction. The result of this symbolic execution is used as the repair specification of the buggy
The evaluation results show that our approach substantially improves upon the current state-of-the-art. We implemented our approach as a tool, NPEX, using existing methods for fault localization and patch generation [13, 67]. We used 119 NPE bugs collected from prior work [36, 41, 67] as well as open-source projects, and compared the performance of NPEX with two state-of-the-art techniques, VFix [67] and GENESIS [36], for repairing NPEs. The results show that NPEX can correctly fix 51% of those bugs even without test cases while GENESIS and VFix fixed 22% and 42%, respectively, with test cases.

Contributions. We summarize our contributions below:

- We present a new technique for validating NPE patches without test cases. The key idea is to infer the expected behavior of a buggy program by combining statistical learning and symbolic execution.
- We present NPEX, an end-to-end patch generation system for Java NPEs. NPEX is able to fix NPEs given a buggy program and crashing input only.
- We demonstrate the effectiveness of NPEX in comparison with current state-of-the-arts. Our results are reproducible; the source code of NPEX and the benchmarks are publicly available.¹

2 OVERVIEW

This section motivates and illustrates NPEX with examples.

2.1 Motivating Example

Figure 1 describes an NPE bug² found in project tablesaw. Method compare in Figure 1(a) checks if two objects associated with temp and org are equivalent. The NPE occurs when dereferencing o1, i.e., o1.equals(o2), at line 4, where o1 is null if elements are missing at position row of org. In practice, NPEs are typically reported with bug-triggering input only; in this case, the bug report was given³ with the single NPE-triggering input shown in Figure 1(b).

Given the buggy program (Figure 1(a)) and NPE-triggering input (Figure 1(b)), NPEX generates the patch in Figure 1(c), which is exactly the same as the developer’s patch. Note that producing the correct patch is nontrivial because there are various ways of avoiding the NPE. For example, all the following candidate patches eliminate the NPE when inserted right before line 4, but their semantics differs from that of the developer patch:

- if (o1 == null) return false;
- if (o1 == null) return true;
- if (o1 == null) o1 = new Object();

To exclude these incorrect patches, NPEX automatically infers the expected specification of the buggy program and validates candidate patches against it. In this case, NPEX infers that “when o1 is null, compare should return true if o2 is null and false otherwise”. None of the incorrect patches satisfy this specification and are therefore rejected by NPEX.

Existing test-based techniques, e.g., VFix [67] and GENESIS [36], do not work well when the test suite only includes the crashing input. VFix would generate the first incorrect patch, if(o1 == null) return false, since its ranking heuristic prioritizes patches that skip statements containing NPEs and return default values. GENESIS generated a patch that implements a logic to remove all Columns with a missing value, which is obviously incorrect.

2.2 How NPEX Works

The distinctive feature of NPEX is in the patch validation phase. We explain how NPEX validates patches with the buggy program in Figure 2, where method invocation obj.getClass() at line 3

---

¹https://github.com/kupil/npe
²https://github.com/jtablesaw/tablesaw/commit/65596d8
³https://github.com/jtablesaw/tablesaw/issues/798
raises an NPE when variable obj is not initialized. The method m is supposed to return the canonical name of the obj’s class. When obj or its canonical name (name) is null, however, m is expected to return the default value (valueIfNull).

Assume that we are given the variable obj at line 3 as the fault expression. Assume further that two candidate patches, P₁ and P₂, in Figure 3 are generated during the repair process. While both patches succeed to eliminate the given NPE, P₁ is incorrect because it returns null, instead of valueIfNull, when obj is null, which violates the intended behavior of the method. The goal of NPEX is to invalidate P₁ while validating the correctness of P₂.

Learning a Null-Handling Model. NPEX uses a statistical null-handling model to validate patches, which is learned from existing codebases and used to repair new, unseen buggy programs. The null-handling model, denoted M, predicts how developers would handle an NPE by inferring an alternative expression to replace the NPE-triggering expression. For example, it infers that, when obj is null, the entire NPE-triggering expression obj.getClass() at line 3 should be interpreted as if it were the expression null. Also, when cls is null, the model infers that the NPE-triggering expression cls.getCanonicalName() at line 4 should be interpreted as null. In summary, when instantiated for the example program in Figure 2, the model M can be treated as the following function:

\[
M = \{ \begin{array}{l}
\text{null.getClass()} \mapsto \text{null} \\
\text{null.getCanonicalName()} \mapsto \text{null}
\end{array} \}.
\]

NPEX learns such a model by mining various null-handling patterns available in codebases. The intuition is that existing null-handling patterns written by developers are good references for handling NPEs (and inferring alternative expressions). For instance, consider the null-handling code snippet available in project rapidoid⁴:

\[
\text{if } (\text{args}[1] \neq \text{null}) \text{ ? args}[1].\text{getClass()} : \text{null}
\]

from which we find that expression null can be alternatively used for the NPE-triggering expression, args[1].getClass(), when args[1] is null. Generalizing this, we learn the first replacement

\[
\text{null.getClass()} \mapsto \text{null}
\]

of the model M in (1). The codebase may have different patterns for the same method getClass. For example, the following pattern is also available in rapidoid:

\[
\text{if } \text{obj} \neq \text{null} \text{ ? obj.getClass()} : \text{Object.class}
\]

from which we infer the following:

\[
\text{null.getClass()} \mapsto \text{Object.class}
\]

When applying the model, we resolve the conflict by choosing the most appropriate one based on the surrounding code context (Section 3.1); for our example, we assumed the pattern in (2) was chosen. The knowledge for handling getCanonicalName can be inferred from the following code snippet in project dozer⁵:

\[
\text{if } \text{destCls} \neq \text{null} \text{ ? destCls.getCanonicalName()} : \text{null}
\]

from which we obtain the second replacement in (1).

### Specification Inference

Once a model is learned, we can use it to infer the expected behavior of a buggy program. To do so, we run a variant of symbolic execution on the buggy program that interprets NPE-triggering expressions using the null-handling model. The result of this symbolic execution will be used as the repair specification against which we validate candidate patches.

Consider the buggy method m in Figure 2. Our symbolic execution begins with the initial state \((\pi_{init}, \sigma_{init})\), where \(\pi_{init}\) is the initial path condition, i.e., \(\pi_{init} = \emptyset\), and \(\sigma_{init}\) is the initial symbolic store, i.e., \(\sigma_{init} = \{\text{obj} \mapsto \alpha, \text{valueIfNull} \mapsto \beta\}\), that maps formal parameters to fresh symbols \(\alpha\) and \(\beta\).

At line 3, we encounter the method invocation, obj.getClass(), whose base variable obj is the fault expression that causes the NPE. The key difference between normal symbolic execution and our variant is that we interpret such an NPE-triggering expression (obj.getClass()) using the alternative expression inferred by the null-handling model. For example, the model in (1) infers null as the alternative expression for obj.getClass() when obj is null. Thus, our symbolic execution produces the following two states as output of line 3:

\[
\begin{align*}
\sigma_1 &= (\pi_1, \sigma_1) = (\alpha = \text{null}, \sigma_{init}[\text{cls} \mapsto \text{null}]) \\
\sigma_2 &= (\pi_2, \sigma_2) = (\alpha \neq \text{null}, \sigma_{init}[\text{cls} \mapsto \text{f(a)}])
\end{align*}
\]

State \(\sigma_1\) represents the output of the NPE-triggering execution where obj is null (denoted by path condition \(\alpha = \text{null}\)). In this case, store \(\sigma_1\) is obtained using the null-handling model as follows:

\[
\sigma_1 = \sigma_{init}[\text{cls} \mapsto M(\text{null.getClass()})] = \sigma_{init}[\text{cls} \mapsto \text{null}]
\]

where we inferred the expected meaning of the method invocation using \(M\). State \(\sigma_2\) represents the normal execution where no NPE occurs. We use an uninterpreted function symbol to represent the return value of the external method; \(\text{f(a)}\) denotes the symbolic value that obj.getClass() evaluates to.

With \(\sigma_1\) and \(\sigma_2\) as input states, symbolic execution of line 4 results in the following output states:

\[
\begin{align*}
\sigma_3 &= (\pi_3, \sigma_3) = (\alpha = \text{null}, \sigma_1[\text{name} \mapsto \text{null}]) \\
\sigma_4 &= (\pi_4, \sigma_4) = (\alpha \neq \text{null}, \sigma_2[\text{name} \mapsto \text{g(f(a))}])
\end{align*}
\]

Note that the execution with \(\sigma_1\) encounters a new NPE because cls holds null in \(\sigma_1\) and method getCanonicalName is called on it. Thus, we use the null-handling model again to infer the expected behavior of null.getCanonicalName and obtain store \(\sigma_3\) as follows:

\[
\sigma_3 = \sigma_1[\text{name} \mapsto M(\text{null.getCanonicalName})] = \sigma_1[\text{name} \mapsto \text{null}].
\]

State \(\sigma_4\) is the result of executing line 4 with input state \(\sigma_2\), where \(\text{g(f(a))}\) is the symbolic value representing cls.getCanonicalName().

We complete symbolic execution by analyzing the if statement at line 5 with states \(\sigma_3\) and \(\sigma_4\) as input, which produces the following three states as output:

\[
\begin{align*}
\sigma_5 &= (\alpha = \text{null}, \sigma_3[\text{ret} \mapsto \beta]) \\
\sigma_6 &= (\alpha \neq \text{null} \land \text{g(f(a))} = \text{null}, \sigma_4[\text{ret} \mapsto \beta]) \\
\sigma_7 &= (\alpha \neq \text{null} \land \text{g(f(a))} \neq \text{null}, \sigma_4[\text{ret} \mapsto \text{g(f(a))}])
\end{align*}
\]

The input state \(\sigma_3\) results in \(\sigma_5\) taking only the true branch as the value of name is null in \(\sigma_3\). With \(\sigma_6\), we consider both true and false branches, producing \(\sigma_6\) and \(\sigma_7\), respectively. Finally, we obtain the

---

⁴https://github.com/rapidoid/rapidoid
⁵https://github.com/DozerMapper/dozer
symbolic summary $S_n$ of method $m$ by removing information about local variables in those states:

$$S_n = \begin{cases} (\alpha = \text{null}, \sigma_{\text{init}}[\text{ret} \mapsto \beta]), \\ (\alpha \neq \text{null} \land g(f(\alpha)) = \text{null}, \sigma_{\text{init}}[\text{ret} \mapsto \beta]), \\ (\alpha \neq \text{null} \land g(f(\alpha)) \neq \text{null}, \sigma_{\text{init}}[\text{ret} \mapsto g(f(\alpha))]) \end{cases}$$

We consider the symbolic summary $S_n$ as the repair specification of the buggy method $m$. That is, our repair specification is the summary of the instrumented semantics of the buggy method $m$ where each NPE-triggering expression is replaced by an alternative expression using the null-handling model.

**Patch Validation.** Now we validate candidate patches against the inferred specification $S_n$. We conclude that a candidate patch is correct iff running a normal symbolic execution on it produces a summary equivalent to $S_n$. For example, the summary $S_1$ of the incorrect patch $P_1$ in Figure 3 is computed as follows:

$$S_1 = \begin{cases} (\alpha = \text{null}, \sigma_{\text{init}}[\text{ret} \mapsto \text{null}]), \\ (\alpha \neq \text{null} \land g(f(\alpha)) = \text{null}, \sigma_{\text{init}}[\text{ret} \mapsto \beta]), \\ (\alpha \neq \text{null} \land g(f(\alpha)) \neq \text{null}, \sigma_{\text{init}}[\text{ret} \mapsto g(f(\alpha))]) \end{cases}$$

Note that $S_1$ and $S_n$ do not agree with the return value, (ret) when $\alpha$ is null, i.e., $\sigma_{\text{init}}[\text{ret} \mapsto \text{null}] \neq \sigma_{\text{init}}[\text{ret} \mapsto \beta]$, concluding that patch $P_1$ does not satisfy the repair specification. By contrast, symbolic execution of the correct patch $P_2$ produces a summary exactly equivalent to $S_n$. We can check the equivalence of summaries using an off-the-shelf SMT solver.

3 OUR PATCH VALIDATION TECHNIQUE

In this section, we describe how NPEX validates patches in detail. Our approach consists of two phases: (1) learning a null handling model (Section 3.1) from a codebase, and (2) validating candidate patches using the model (Section 3.2).

**Programs.** A Java program $P \in Pgm$ is a sequence of class declarations, where a class declaration is a pair of a class name and a sequence of method declarations. A method declaration consists of a return type, a method name, a formal parameter, and a body statement. We write $pm$ and $bod(m)$ for the parameter and body statement of method $m$, respectively. A type $T$ is either a primitive type (we only consider int for simplicity) or a reference type for custom classes ($C$). We consider the usual statements ($S$) and expressions ($E$) in Java:

$$S \rightarrow x = e \mid \text{return } e \mid \text{if } E \mid S_1 \mid S_2 \mid \text{while } E \mid S \mid S_1 ; S_2 \mid E$$

$$E \rightarrow n \mid \text{null} \mid x \mid x.m(y) \mid \text{new } C() \mid E_1 \equiv E_2 \mid E \rightarrow E$$

We do not consider field access ($x.y$) because it is very rare to directly access public fields in real-world Java programs; fields are typically accessed via getter methods that our language supports. We assume the program is statically typed and write type($e$) for the type of expression $e$. We assume that method names are unique. The body of a while statement may include control statements break or continue. A variable is either a local or this. We write $E_{\text{NPE}}$ for the set of expressions where NPEs occur; in our language, $E_{\text{NPE}}$ represents the set of method invocations, i.e., $x.m(y)$, where NPEs occur when $x$ is a null pointer.

3.1 Learning a Null-Handling Model

The goal of the learning phase is to learn a null-handling model

$$M \in Pgm \times E_{\text{NPE}} \rightarrow E$$

from a dataset of programs. Given a program $P \in Pgm$ and its NPE-triggering expression $e_{\text{NPE}} \in E_{\text{NPE}}$, $M(P, e_{\text{NPE}})$ predicts an alternative expression that can be used as a substitute for $e_{\text{NPE}}$ to correctly handle the NPE. We construct the model in the following steps.

**Collecting Null-Handling Patterns.** The first step is to collect a dataset $D$ of null-handling patterns from a codebase. Let $P = \{P_1, P_2, \ldots, P_m\}$ be a collection of programs. The dataset $D$ is of the type $D \subseteq Pgm \times E_{\text{NPE}} \times E$. That is, $D$ is a set of tuples $(P, e_{\text{NPE}}, e)$ where $P \in P$ is a program in the codebase, $e_{\text{NPE}}$ is an NPE expression in $P$, and $e$ is an expression that is alternatively used in $P$ when $e_{\text{NPE}}$ causes an NPE.

To collect the dataset $D$, we traverse the abstract syntax tree of each program $P \in P$ and observe how NPEs are handled. For example, from ternary expression of the form $x == \text{null} \? e : x.m(y)$, we collect tuple $(P, x == \text{null} \? e : x.m(y), e)$, meaning that, when $x$ is a null pointer and hence an NPE occurs at expression $x.m(y)$, we can alternatively use expression $e$ instead of $x.m(y)$. When collecting null-handling patterns, we mainly consider such ternary expressions as the NPE-triggering expression ($x.m(y)$) and the corresponding alternative expression ($e$) are clearly identifiable. Note that other several null-handling patterns can be translated to ternary expressions. For example, boolean expression $x == \text{null} \? x.m(y) : \text{true}$ is translated into $x == \text{null} ? \text{true} : x.m(y)$ and represented by tuple $(P, x.m(y), \text{true})$.

**Generalization.** Once we collect null-handling patterns, we generalize them by abstracting alternative expressions. The main purpose of this step is to discard program-dependent information such as local variable and user-defined class names. We also make the dataset more amenable to learning by considering a finite subset of alternative expressions. The output of this step is the following:

$$\widehat{D} \subseteq Pgm \times E_{\text{NPE}} \times \widehat{E}$$

where $\widehat{E}$ denotes abstract expressions defined as follows:

$$\widehat{E} \in \mathbb{N}_D \cup \{\text{null}, \text{ARG}, \text{NEW}\} \cup \{\text{ARG} == \text{null} \mid \text{null} \cup \{\text{null}\} \cup \{\text{true}\}\}$$

Here, $\mathbb{N}_D$ denotes a finite set of integers that frequently appear in the dataset $D$. NPEX uses $\mathbb{N}_D = \{-1, 0, 1\}$ because they were the top-3 most popular integers in $D$. We do not distinguish names of method arguments and represent them by an abstract element, denoted ARG. A new expression (new $C()$) is abstracted to NEW where its type information ($C$) is discarded. An equality test ($E_1 == E_2$) is abstracted into ARG == $\backslash$ when $E_1$ is an argument variable and $E_2$ is generalized to a literal in $\mathbb{N}_D \cup \{\text{null}\}$. We discard other cases and simply represent them by $\backslash$. Specifically, we define function $\alpha : E_{\text{NPE}} \times E \rightarrow \widehat{E}$ for generalization as follows:

$$\alpha(e_{\text{NPE}}, e) = \begin{cases} e \quad \text{if } e \in \mathbb{N}_D \cup \{\text{null}\} \\ \text{NEW} \quad \text{if } e = \text{new } C() \\ \text{ARG} \quad \text{if } e_{\text{NPE}} == x.m(y) \land e == y \\ \text{ARG} == n \quad \text{if } e_{\text{NPE}} == x.m(y) \land e == (y == n) \land n \in \mathbb{N}_D \cup \{\text{null}\} \\ \text{T} \quad \text{otherwise} \end{cases}$$
With \( \alpha \), the generalized dataset \( \hat{D} \) is obtained as follows:
\[
\hat{D} = \{(P, \psi_{\text{NPE}}, \alpha(\psi_{\text{NPE}}, e)) \mid (P, \psi_{\text{NPE}}, e) \in D\}.
\]

**Feature Representation.** Next, we represent NPE expressions as feature vectors to generate the training data \( \mathcal{D} \):
\[
\mathcal{D} \subseteq \{0, 1\}^n \times \hat{E}.
\]
To do so, we assume \( n \) boolean features: \( \Phi = \{\phi_1, \phi_2, \ldots, \phi_n\} \), where each feature \( \phi_i : Pgm \times E_{\psi_{\text{NPE}}} \rightarrow \{0, 1\} \) is a predicate on program and NPE expression pairs and describes characteristics of an NPE expression and its surrounding code context. For instance, a feature may describe whether an NPE expression belongs to a try/catch block. We write \( \Phi(P, \psi_{\text{NPE}}) \) for the feature vector of \( (P, \psi_{\text{NPE}}) \). With \( \Phi \), we can generate the training data \( \mathcal{D} \) as follows:
\[
\mathcal{D} = \{(\Phi(P, \psi_{\text{NPE}}), \tilde{e}) \mid (P, \psi_{\text{NPE}}, e) \in \hat{D}\}.
\]
We use 31 features in Table 1. Here, each feature is a predicate on method calls, \( x.m(y) \), since \( \psi_{\text{NPE}} \) denotes the set of method invocations in our language. The features are divided into three classes: method name features, method body features, and context features. The method name features describe which keywords appear in the name \( f \) of the called method. We used 20 keywords as the name features. To select these keywords, we collected all the method names from our codebase, split them into keywords by the camel case, and ranked the top 20 by their frequency. The method body features check if the body statement contains a specific AST component. We designed those two classes of features to identify what kind of methods is invoked. For example, features \#8 and \#27 are strong indicators for getter methods. The context features capture the syntactic code contexts around an NPE expression.

**Training a Model.** From the training data \( \mathcal{D} \), we train a probabilistic multi-label classifier to learn a probability distribution over \( \hat{E} \) for a given feature vector. Let \( Pr \) be the learned probability distribution; given a program \( P \), an NPE expression \( \psi_{\text{NPE}} \), and abstract expression \( \varepsilon \in \hat{E}, Pr(\varepsilon | \Phi(P, \psi_{\text{NPE}})) \) denotes the probability of the alternative expression of \( \psi_{\text{NPE}} \) being \( \varepsilon \). We computed \( Pr \) using an off-the-shelf learning algorithm for Random Forest Classifier.

We can construct the null-handling model \( M \) from \( Pr \), with an additional process that concretizes an abstract expression into a type-compatible concrete expression. We define \( \gamma : E_{\psi_{\text{NPE}}} \times \hat{E} \rightarrow \hat{E} \cup \{\perp\} \) for concretization, which converts an abstract expression into a concrete expression as follows:
\[
y(x.m(y), \tilde{e}) = \begin{cases} n & \text{if } \tilde{e} = n \land T = \text{int} \\
\text{null} & \text{if } \tilde{e} = \text{null} \land T \neq \text{int} \\
\text{new } C(\tilde{e}) & \text{if } \tilde{e} = \text{NEW} \land T = C \\
y(\tilde{e}) & \text{if } \tilde{e} = \text{ARG} \land \text{type}(y) = T \\
y == e & \text{if } \tilde{e} = \text{ARG} \land e \land T = \text{int} \\
\perp & \text{otherwise}
\end{cases}
\]
where \( T \) denotes the type of expression \( x.m(y) \). With \( \gamma \), our null-handling model \( M : Pgm \times E_{\psi_{\text{NPE}}} \rightarrow \hat{E} \) is defined as follows:
\[
M_P(\psi_{\text{NPE}}) = \gamma(\psi_{\text{NPE}}, \text{argmax } Pr(\varepsilon \mid \Phi(P, \psi_{\text{NPE}})))
\]
where \( C = \{\varepsilon \in \hat{E} \mid \gamma(\psi_{\text{NPE}}, \varepsilon) \neq \perp\} \) is the set of concretizable expressions and we pick one with the highest probability.

### Table 1: Features for method invocations

<table>
<thead>
<tr>
<th>Class</th>
<th>#</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name Features</td>
<td>1</td>
<td>return type is void</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>'Error'</td>
</tr>
<tr>
<td></td>
<td>7</td>
<td>'Success'</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>'get'</td>
</tr>
<tr>
<td></td>
<td>9</td>
<td>'set'</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>'is'</td>
</tr>
<tr>
<td></td>
<td>11</td>
<td>'add'</td>
</tr>
<tr>
<td></td>
<td>12</td>
<td>'close'</td>
</tr>
<tr>
<td></td>
<td>13</td>
<td>'Empty'</td>
</tr>
<tr>
<td></td>
<td>14</td>
<td>'Value'</td>
</tr>
<tr>
<td>Body Features</td>
<td>21</td>
<td>return type is void</td>
</tr>
<tr>
<td></td>
<td>22</td>
<td>method returns a literal</td>
</tr>
<tr>
<td></td>
<td>23</td>
<td>thrown exceptions are annotated</td>
</tr>
<tr>
<td></td>
<td>24</td>
<td>null check expression exists</td>
</tr>
<tr>
<td></td>
<td>25</td>
<td>method returns a constructor call</td>
</tr>
<tr>
<td></td>
<td>26</td>
<td>method is the base of another invocation</td>
</tr>
<tr>
<td></td>
<td>27</td>
<td>method returns a field</td>
</tr>
<tr>
<td>Context Features</td>
<td>28</td>
<td>caller method is private</td>
</tr>
<tr>
<td></td>
<td>29</td>
<td>null pointer is assigned to an array</td>
</tr>
<tr>
<td></td>
<td>30</td>
<td>null pointer is assigned to a field</td>
</tr>
<tr>
<td></td>
<td>31</td>
<td>null pointer is assigned to a public field</td>
</tr>
</tbody>
</table>

### 3.2 Validating Patches using the Model

Next we use the model to validate the correctness of a candidate patch. To this end, we first infer the correctness specification of a buggy program using the learnt null-handling model and then check if the patch candidate satisfies the inferred specification.

**Specification Inference.** We use the null-handling model to infer the correct behavior of a buggy program. Suppose a buggy program \( P_{\psi_{\text{NPE}}} \in Pgm \) is given and the buggy (NPE-triggering) expression \( P_{\psi_{\text{NPE}}} \) is \( \psi_{\text{NPE}}, m(y) \), where \( \psi_{\text{NPE}} \) is the fault variable whose value is null along the buggy trace. For simplicity, we assume there is only a single NPE in the buggy program. Suppose also that a null-handling model \( M \) learned from an existing codebase is given. The goal of specification inference is then to infer the desired behavior of the buggy program when the NPE is correctly fixed.

We infer the correctness specification by running a variant of symbolic execution on the buggy program and interpreting the NPE-triggering expression \( \psi_{\text{NPE}}, m(y) \) using the null-handling model. To this end, we first define a normal symbolic execution procedure and explain how to extend it to use the null-handling model.

We consider the output of symbolic execution as the specification of the input program. The output of our symbolic execution is a table \( \Sigma \) from methods to summaries:
\[
\Sigma \in \text{SumTable} = \text{Method} \rightarrow \text{Summary}
\]
where a summary is defined as follows:
\[
\begin{align*}
S & \in \text{Summary} = \mathcal{P}(\text{State}) \\
s & \in \text{State} = \mathcal{P}(\text{PC} \times \text{Store}) \\
p & \in \text{PC} = \mathcal{P}(\text{SymVal} \times \{=, \neq\} \times \text{SymVal}) \\
\sigma & \in \text{Store} = \text{Var} \rightarrow \text{SymVal} \\
v & \in \text{SymVal} = \mathcal{Z} + \text{Class} + \{\text{null}\} + \text{Symbol}
\end{align*}
\]
A summary (Summary) is a set of program states (State) and a state consists of a path condition (PC) and a store (Store). A path condition is a collection of branch conditions, where a branch condition is an equality of symbolic values or its negation. A store is a map from
variables to symbolic values. Symbolic values include integers, class types, null, and symbols representing method parameters.

For scalability, we have designed our symbolic execution, denoted SymExec: Pgm → SumTable, to be compositional and bounded [11]. It analyzes each method of a program only once by calculating its summary in isolation using summaries of callees methods. Loops and recursive call cycles are unrolled finite times prior to the analysis. For presentation simplicity, we assume that method names are unique (i.e., no method overriding and overloading) and ternary expressions are converted to 1f statements. With these assumptions, it is enough to define symbolic execution for the following subset of statements and expressions:

\[
S \rightarrow x := e \mid \text{return } x \mid \text{if } (x = := y) \mid S_1 \mid S_2 \mid S_1 ; S_2
\]

\[
E \rightarrow \text{null} \mid \text{new } C()
\]

The procedure SymExec is defined as follows, which computes method summaries in a bottom-up manner:

\[
\text{SymExec}(P) = \left\{ m_i \mapsto F(\text{body}(m_i), S', s'_{\text{init}}) \right\}_{i=1}^{n}
\]

where \( m_1, \ldots, m_n \) are a sequence of methods sorted according to the reverse topological order of the call-graph, \( S' \) is the partial summary table for methods \( m_1, \ldots, m_{i-1} \), and the initial state \( s'_{\text{init}} \) is defined by \( (\emptyset, \left\{ p_i \mapsto \sigma_{m_i} \right\}) \) which indicates that the formal parameter \( p_i \) of method \( m_i \) is bound to a fresh symbolic value \( \sigma_{m_i} \). The semantic function \( F: \text{Stmt} \times \text{SumTable} \times \text{State} \rightarrow F(\text{State}) \) is defined in a standard manner as follows:

\[
F(x := n, \Sigma, (\pi, \sigma)) = \{ (\pi, \sigma[x := n]) \}
\]

\[
F(x := y, \Sigma, (\pi, \sigma)) = \{ (\pi, \sigma[x := \sigma(y)]) \}
\]

\[
F(\text{return } x, \Sigma, (\pi, \sigma)) = \{ (\pi, \text{new } C()) \}
\]

\[
F(\text{if } x == y, S_1, S_2, \Sigma, (\pi, \sigma)) = \{ (\pi \cup \{ \text{if } x == y : S_1, S_2, \Sigma, (\pi, \sigma) \}) \}
\]

\[
F(\text{new } C(), \Sigma, (\pi, \sigma)) = \{ (\pi, \text{new } C()) \}
\]

where \( s_1, s_2 = (\pi \cup \{ \sigma(x) = \sigma(y) \}, \sigma), (\pi \cup \{ \sigma(x) \neq \sigma(y) \}, \sigma) \) and \( \{ \sigma_{m_1} \} \in \Sigma[m] \Rightarrow \sigma_{m_1} \equiv \sigma(z) \}.

Now we describe how we infer the repair specification of a buggy program \( P_{\text{bug}} \). We do so by analyzing \( P_{\text{bug}} \) with a variant of symbolic execution, denoted \( \text{SymExec}_{\text{M}}^{\text{MPE}} \), where \( \text{MPE} \) refers to the localized fault variable that causes the NPE we aim to fix and \( \text{M} \) is the null-handling model. The overall procedure remains the same:

\[
\text{SymExec}_{\text{M}}^{\text{MPE}}(P) = \left\{ m_i \mapsto F_{\text{MPE}}^{\text{MPE}}(\text{body}(m_i), s_i', s'_{\text{init}}) \right\}_{i=1}^{n}
\]

The extended semantic function \( F_{\text{MPE}}^{\text{MPE}}(S, \Sigma, (\pi, \sigma)) \) is defined in the same way as \( F \) except for the following two cases. The first case is when \( S \) is an NPE-triggering statement, i.e., \( x := y.m(z) \) or \( y \) is the fault variable \( x_{\text{fault}} \). In this case, \( F_{\text{MPE}}^{\text{MPE}}(S, \Sigma, (\pi, \sigma)) \) produces

\[
F(S, \Sigma, (\pi_{\text{nonnull}}, \sigma)) \cup \{ (x := M_{\text{MPE}}(y.m(z)), \Sigma, (\pi_{\text{nonnull}}, \sigma)) \}
\]

where \( \pi_{\text{nonnull}} = \pi \cup \{ \sigma(x_{\text{null}}) \neq \text{null} \} \) and \( M_{\text{MPE}} = \pi \cup \{ \sigma(x_{\text{nonnull}}) = \text{null} \} \). The former describes states where the fault variable is not \( \text{null} \), hence they normally execute the invocation. The latter describes states where the fault variable is \( \text{null} \). In this case, we interpret the NPE-triggering expression using the output of the model \( \text{M} \). The second case is when \( S \) is \( y \) \text{null}(z) and the base variable \( y \) is not the fault variable but evaluates to \( \text{null} \), i.e., \( (\sigma(y) = \text{null}) \in \pi \). This happens when the prediction of the model in the former case returns \( \text{null} \), i.e., \( M_{\text{MPE}}(y.m(z)) = \text{null} \). If a new NPE is introduced by the model, we apply the model again:

\[
F_{\text{MPE}}^{\text{MPE}}(S, \Sigma, (\pi, \sigma)) = F(x := M_{\text{MPE}}(y.m(z)), \Sigma, (\pi, \sigma)).
\]

Example 3.1. Note that we infer procedural summaries not only for the faulty method, but also for its callers. Let us consider the following code that uses null pointers across procedure boundaries.

```java
A foo(p) { return p.hoo(); // NPE }
int goo(z) {
    A x = this.foo(z);
    return x.goo();
}
```

Suppose that NPE occurs at line 1 because \( p \) is \text{null} and the alternative expression inferred by the model is \text{null} for the NPE-triggering expression \( p.hoo() \). Then, the inferred procedural summary of \( \text{foo} \) is \{ \( \sigma_{\text{foo}} = \text{null}, \sigma_{\text{retfoo}} \mapsto \text{null} \} \), where \( \sigma_{\text{foo}} \) and \( \sigma_{\text{retfoo}} \) denote the symbolic parameter and return variable, respectively. Then, the return value \text{null} propagates to caller’s variable \( x \). So the model \( \text{M} \) is applied again at line 4, and we get the summary of \( \text{goo} \) as \{ \( \sigma_{\text{goo}} = \text{null}, \sigma_{\text{retfoo}} \mapsto \text{null} \} \} where we assume \( v \) is the alternative value for \( x.\text{goo}() \) obtained by the model \( \text{M} \).

**Specification Validation.** The next step is to validate candidate patches against the inferred specification. Let \( P_{\text{bug}} \) be a buggy program and \( P_{\text{cand}} \) be a patch candidate. We would like to determine whether \( P_{\text{cand}} \) is a correct patch of \( P_{\text{bug}} \). We do this by checking the equivalence: \( \text{SymExec}_{\text{M}}^{\text{MPE}}(P_{\text{bug}}) \equiv \text{SymExec}(P_{\text{cand}}) \), where the left-hand side denotes the inferred repair specification of the buggy program. We say that two summary tables \( \Sigma_1 \) and \( \Sigma_2 \) are equivalent, \( \Sigma_1 \equiv \Sigma_2 \), if the following holds for all methods \( m \):

\[
\bigwedge (\pi, \sigma) \in \Sigma_1(m) \bigwedge (\pi, \sigma) \in \Sigma_2(m) \Rightarrow \pi_1 \cap \pi_2 \Rightarrow \sigma_1 = \sigma_2
\]

where we assume the symbolic value for the formal parameter of method \( m \) is consistently named (e.g., \( \sigma_m \)). Intuitively, the formula checks if all output states are equal in the inferred specification and the summary of the candidate patch.

Note that we check the equivalence not only for the patched method but also for its callers. A patch could implement correct semantics for the patched method, but incorrect semantics for its callers. For example, consider the program in Example 3.1. A patch that modifies \( p.hoo() \) to \( p == \text{null} \) : \( p.hoo() \) is correct for method \( \text{foo} \), but it introduces a new NPE in \( \text{goo} \). In this case, we can successfully reject this patch by checking the semantic equivalence for \( \text{goo} \) as well.

4 NPEX

NPEX is an end-to-end repair tool based on our idea in Section 3. In this section, we describe other details of NPEX.

**Implementation.** We implemented NPEX in 4,200 lines of Java and 7,400 lines of OCaml codes. Our fault localization algorithm and symbolic execution are implemented on top of the Isabela [20] framework. We also used the Spoon [49] library to parse Java programs and transform source codes.

**Overall Algorithm.** Algorithm 1 describes the overall algorithm of NPEX. Given a buggy program \( P_{\text{bug}} \), an NPE stack trace
we compute a set of programs twice. At line 2, the buggy program \( P_{\text{BPE}} \) is analyzed by our variant of symbolic execution, \( \text{SymExec}_{\text{NPE}}^\mathcal{M} \), which returns the repair specifications, \( \Sigma_{\text{inferred}} \), of the buggy program. At line 3, we compute a set \( \mathcal{X}_{\text{VFix}} \) of candidate faults. We iterate each candidate fault \( x_{\text{prev}} \) in \( \mathcal{X}_{\text{VFix}} \), and accumulate validated patches in \( \mathcal{P}_{\text{Fix}} \). At line 5, we enumerate patches with given fault \( x_{\text{prev}} \), and for each patch \( P_{\text{patched}} \), we compute its summary table \( \Sigma_{\text{patched}} \) by normal symbolic execution. If the summary table of \( P_{\text{patched}} \) is equivalent to the inferred specification \( \Sigma_{\text{inferred}} \), we add it to \( \mathcal{P}_{\text{Fix}} \).

**Fault Localization.** Our fault localization is similar to that of VFix [67] in that it tracks the dataflow of a given null pointer. Given the fault expression \( x_{\text{prev}} \) and stack trace \( \tau \) only, FaultLocalization computes a set of null pointer expressions that may be alias to \( x_{\text{prev}} \) for each method in the stack trace \( \tau \). We do not rank each fault and returns the set of all the computed faults, as we validate patches by inferred specifications rather than relying on a patch ranking heuristic. We implemented a light-weight pointer analysis on top of Infer to compute alias information.

**Patch Enumeration.** Given the localized fault expression \( x_{\text{prev}} \), PatchEnumeration enumerates patches based on pre-defined templates. We used the following templates from prior work [13, 67]:

- **SKIP**: (i) if \( x_{\text{prev}} \neq \text{null} \) \( \delta \), or (ii) if \( x_{\text{prev}} = \text{null} \) \( \beta \)
- **REPLACE**: \( x_{\text{prev}} = \text{null} \) \( \epsilon : \text{null} \)
- **INIT**: if \( x_{\text{prev}} = \text{null} \) \( \varphi \)

SKIP skips a statement or a block (\( \beta \)) containing an NPE (\( \text{SKIP} \)-(i)), or inserts a control flow break (\( \beta \)): break, continue, return \( \epsilon \), or throw \( E \) (\( \text{SKIP} \)-(ii)). REPLACE substitutes an expression involving the fault expression \( x_{\text{prev}} \) with a ternary with an alternative expression \( \epsilon \). INIT initializes a null pointer to a fresh object obtained by calling a constructor \( e \). For expressions \( \epsilon \), we used frequent expressions in null-handling patterns collected from our training database. We collected the top-3 frequent expressions for each of primitive types and common class types (e.g., java.util.ArrayList). We synthesize exceptions (\( E \)) from exceptions thrown around the fault expression within its class. We implemented PatchEnumeration using the Spoon library’s source code transformation.

**Scalable Symbolic Execution.** We implemented symbolic execution on top of the Infer’s bottom-up analysis framework. We took advantage of the bottom-up analysis to analyze only the parts of programs related to each patch in the validation phase. Although the analysis is bottom-up, it was nontrivial to implement a scalable symbolic executor that works for real-world applications while supporting the full Java language including dynamic dispatch, exception handling, field accesses, etc. Thus, we designed a path merging heuristic to further accelerate the analysis; we only distinguished error states where an NPE occurs while merging other NPE-irrelevant states. We treated results of an invocation for an external function as an uninterpreted symbol assuming it has no side-effect. For the Java Library Class methods (e.g., the String methods), we modeled the effect of each method. We unraveled each loop of programs twice.

### Algorithm 1: The NPEX Algorithm

**Input:** Buggy program \( P_{\text{BPE}} \), NPE stack trace \( (x_{\text{prev}}, \tau) \), model \( \mathcal{M} \)

**Output:** A set \( \mathcal{P}_{\text{Fix}} \) of validated patches

1. \( x_{\text{prev}}, \tau \leftarrow \text{RunProgram}(P_{\text{BPE}}, I) \) \( \rightarrow \) Fault variable and stack trace
2. \( \mathcal{P} \leftarrow \emptyset \)
3. \( \Sigma_{\text{inferred}} \leftarrow \text{SymExec}_{\text{NPE}}^\mathcal{M}(P_{\text{BPE}}) \) \( \rightarrow \) Spec inference
4. \( x \leftarrow \text{FaultLocalization}(P_{\text{BPE}}, \tau) \)
5. for \( x_{\text{prev}} \in \mathcal{X} \) do
6.   for \( P_{\text{patched}} \in \text{PatchEnumeration}(P_{\text{BPE}}, x_{\text{prev}}) \) do
7.     \( \Sigma_{\text{patched}} \leftarrow \text{SymExec}(P_{\text{patched}}) \)
8.     if \( (\Sigma_{\text{patched}} \equiv \Sigma_{\text{inferred}}) \) then
9.       \( \mathcal{P} \leftarrow \mathcal{P} \cup \{ P_{\text{patched}} \} \)
10. end if
11. end for
12. end for
13. return \( \mathcal{P}_{\text{Fix}} \)

**Null-handling Model.** We implemented null-handling code mining and feature extraction using Spoon. We collected null-handling patterns of more various syntactic forms than ternary described in Section 3.1. For example, we additionally collected null-handles of the following form:

\[
y = e; \ldots \text{if } (x_{\text{prev}} \neq \text{null}) \{ y = x_{\text{prev}}.\text{foo}(); \}
\]

where we interpreted e as an alternative value for foo.

**Use Cases of NPEX.** NPEX can be used in many application scenarios. First of all, Note that NPEX does not require a failing “test”; instead, NPEX only requires a stack trace (or a crashing input to obtain the stack trace). In the context of NPEs, a failing “test” is a pair of a crashing, NPE-triggering input and the corresponding expected output. What NPEX requires is the crashing input, which is available when NPEs are detected and confirmed. On the other hand, we do not require the expected output, which is typically unavailable at the time NPEs are reported; it is provided by a developer later when the reported NPE is fixed.

Because NPEX only requires a stack trace (crashing input), it can be used in many practical scenarios to automatically fix NPEs as soon as they are detected. For example, we can use NPEX to fix NPEs detected by automatic testing tools or static bug-finders. Also, NPEX can be used to fix NPEs reported in issue tracking systems.

Stack traces are usually available when NPEs are detected and reported. For example, when NPEs are found by testing tools, stack traces are immediately available from the crashes. Also, static bug-finders usually provide error traces (stack traces) for reported bugs. When NPE bugs are reported in issue tracking systems, it is typical that users include a crashing input or stack trace in the bug report.

### 5 Evaluation

In this section, we evaluate NPEX in comparison with state-of-the-art techniques for repairing NPEs.

#### 5.1 Setup

**Benchmark Selection.** We used four different benchmark sets:

- **VFixBM:** 30 NPE bugs from [67].
We note that the structures of the projects in VFix with NPEFix APR techniques such as (i.e., the parent revision of the collected commit) and the error is 15 from open-source repositories collected by the authors of 6 included NPEX of our new approach; NPEX prior work [67]. We also consider Genesis and ACS [66].

Specific and known to be significantly more effective than existing VFix repairing NPEs.

OurBM are already contained in the three benchmark sets above, which led to check whether an NPE is reproducible in the buggy version. Also, we excluded benchmarks that already exist in different benchmark sets, leading to 16 and 14 NPE bugs in GENBM and BEARSBM, respectively.

In addition, we tried to collect more diverse NPE bugs from open-source projects and constructed OURBM as the result. We used two sources: (1) the top-200 Java repositories in GitHub sorted by the number of stars and (2) repositories under the Apache project page. Among them, we only considered projects that can be built from the maven system. From those repositories, we collected 4472 commits whose messages contain keywords 'NPE' or 'Null Pointer Exception', where we only considered recent commits up to 5 years ago. We then searched for reproducible NPE bugs in a similar way as BEARS was collected [41]; we only ran NPE-triggering test cases to check whether an NPE is reproducible in the buggy version (i.e., the parent revision of the collected commit) and the error is removed in the fixed version. Also, we excluded benchmarks that Spoon or Infer fail to handle. Finally, we excluded benchmarks that are already contained in the three benchmark sets above, which led to a total of 59 NPE bugs in OURBM.

In total, we used 119 unique NPE bugs and all of them come with test suites written by developers to check patch correctness. All experiments were done on a machine running Ubuntu 18.04 with 20 CPUs and 128GB memory, powered by Intel Xeon Gold 6230 processor.

**Tool Selection and Setup.** We evaluated NPEX in comparison with VFix [67] and Genesis [36], two state-of-the-art techniques for repairing NPEs. VFix is the most recent technique, which is NPE-specific and known to be significantly more effective than existing APR techniques such as NPEFix [13], Nopol [68], CapGen [60], and ACS [66]. Genesis is a data-driven technique that can effectively fix NPEs using an NPE-specific patch space learned from human patches. We included Genesis as it was not evaluated in prior work [67]. We also consider NPEX_base, which is the baseline of NPEX that uses the conventional test-based patch validation instead of our new approach; NPEX_base uses exactly the same techniques for fault localization and patch generation as NPEX (Section 4). We included NPEX_base to see the net effect of our key contribution (automatic specification inference and validation). In the evaluation, we excluded NPEFix [13], another recent technique to fix NPEs, because it was reported that VFix significantly outperforms NPEFix on VFixBM [67]. Also, the results of NPEX_base hint at the performance of NPEFix for other benchmarks, as they use similar patch templates and the same patch validation method (i.e., test cases). In summary, we used the following tools in evaluation:

- **NPEX**: our technique (without test cases)
- **NPEX_base**: the baseline of NPEX (with test cases)
- **VFix**: a state-of-the-art for fixing NPEs (with test cases)
- **Genesis**: a data-driven technique for NPEs (with test cases)

For NPEX, we only used the single NPE-triggering test contained in each benchmark, and did not use other test cases. Instead, NPEX used a null-handling model learned from 571 Java projects. We collected these projects starting from the top-1000 Java projects based on the number of stars and excluding ones that are not built with maven or Spoon. We also excluded projects contained in the four benchmark sets in order to ensure that the training and test sets do not overlap. For NPEX_base, we used the same setting as NPEX except that NPEX_base uses test cases.

We obtained Genesis from the replication package released by the authors. When running Genesis, we used the search space learned for NPE, which is also provided in the replication package. Specifically, we used the space named npe-space-vo because it has been reported as the best among others [35]. Genesis takes as input a list of passing and failing test cases, which is used for fault localization and patch validation. Because several benchmarks contain multiple failing tests other than the NPE-triggering test, we only used tests in the NPE-triggering test case’s class as input so that the Genesis can precisely localize the target NPE. Although Genesis used bugs with more than 50 test cases in [36], we observed that no performance degradation occurred due to this setting, compared to the original numbers reported in [36].

We obtained the executable binary (JAR) of VFix via personal communication with the authors. Running VFix was nontrivial as it requires not only a stack trace and null pointer (fault) expression but also a runnable main class with an entry point which acts as a test suite. Therefore, running VFix on GENBM, BEARSBM, and OURBM was particularly challenging. We had to manually write the Main class for each bug, had to resolve the classpath for dependencies and set up testing environments by hand without aids of build systems and testing frameworks. We also encountered several internal errors of VFix running on those benchmarks but we could not debug them as source code is unavailable. As a result, though we did our best, we ended up with 27 out of 89 benchmarks in GENBM, BEARSBM, and OURBM. For those 27 benchmarks, we prepared stack traces and null pointer expressions by running NPE-triggering test cases.

**Correctness Criteria.** We say a patch is correct if it is semantically equivalent to the developer’s patch. We manually investigated each of the generated patches to check the correctness. Following Genesis [36], we ignored log messages and error messages of exceptions in the judgement. VFix often failed to convert an IR (intermediate representation) to source code. In this case, we

---

1. https://maven.apache.org/
2. https://github.com/apache/
Table 2: Evaluation results. #R: the number of bugs for which each tool was successfully ran. #G: the number of patchess successfully generated and validated by each tool. #C: the number of correct patches. Prec: precision (\(\frac{#C}{#G}\)). FixR: fix rate (\(\frac{#R}{#G}\)).

<table>
<thead>
<tr>
<th>Benchmarks</th>
<th>NPEX</th>
<th>NPEX\textsubscript{base}</th>
<th>GENESIS \textsuperscript{[56]}</th>
<th>VFix \textsuperscript{[67]}</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>#Bug</td>
<td>#R</td>
<td>#G</td>
<td>#C</td>
</tr>
<tr>
<td>VFixBM</td>
<td>30</td>
<td>30</td>
<td>28</td>
<td>19</td>
</tr>
<tr>
<td>GenBM</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>10</td>
</tr>
<tr>
<td>BearsBM</td>
<td>14</td>
<td>14</td>
<td>11</td>
<td>6</td>
</tr>
<tr>
<td>OurBM</td>
<td>59</td>
<td>59</td>
<td>44</td>
<td>26</td>
</tr>
<tr>
<td>Total</td>
<td>119</td>
<td>119</td>
<td>99</td>
<td>61</td>
</tr>
</tbody>
</table>

5.2 Results

Table 2 shows the evaluation results. Out of 119 bugs, NPEX generated and validated patches for 99 bugs (#G), and among them 61 were correct, leading to a fix rate of 51% (\(\frac{61}{99}\)) and a precision (i.e., how precisely generated patches turned out to be correct) of 62% (\(\frac{61}{99}\)). On the other hand, NPEX\textsubscript{base}, which does not use our patch validation but relies on a test suite, resulted in a fix rate of 24% and a precision of 28%, which shows that our patch validation is much more effective than conventional test-based validation.

Meanwhile, GENESIS generated 53 patches out of 89 bugs and 20 were correct among those, leading to a fix rate of 22% (\(\frac{20}{89}\)) and a precision of 38% (\(\frac{20}{89}\)). We could not run GENESIS on VFixBM because it has no build system and testing framework which are required to run GENESIS. We could successfully run VFix for 57 benchmarks and VFix generated 44 patches in total. The number of correct patches out of 44 were 24, leading to a fix rate of 42% (\(\frac{24}{44}\)) and a precision of 55% (\(\frac{24}{44}\)). Most of the correct patches produced by VFix were from VFixBM (20 out of 24). For other benchmarks (GenBM, BearsBM, OurBM), VFix generated 18 patches and 4 among them were correct, leading to a fix rate of 15% and a precision of 22%, which are substantially lower than the fix rate of 67% and the precision of 77% for VFixBM. This is because, most of the correct patches in VFixBM are similar in that they simply skip statements or blocks that contain NPEs, for which the VFix’s ranking heuristic works well. However, other benchmark sets contain NPE bugs that require more diverse fix strategies (e.g., returning a non-default value or replacing an existing expression). By contrast, NPEX consistently shows good performance over the four benchmark sets. Our tool validated 99 bugs, and among them 61 were correct, leading to a fix rate of 67% and a precision of 38% (\(\frac{61}{99}\)).

Figure 4: A simplified code snippet containing an NPE (line 4) from the project Apache aries-jpa’s revision 7712046

(a) Buggy Program

```java
public List getJpaAnnotated(Class c, ...) {
    final List jpaAnnotated = new ArrayList<>();
    while (c != Object.class) {
        for (Field f: c.getDeclaredFields()) { // NPE
            if (f.getType() == String.class) {
                String str = f.get(c);
            }
        }
        c = c.getSuperclass();
    }
    return jpaAnnotated;
}
```

(b) Developer’s test case

```java
List members = ...getJpaAnnotated(TestInterface.class, ...);
Assert.assertEquals(0, members.size());
```

A frequent performance bottleneck in generate-and-validate approaches, is zero for NPEX because it does not use test cases at all. Running test cases was very expensive for some benchmarks. For example, project commons-pool_41f4e41 took 3 minutes for a single run of the test suite, which must be repeated many times during the repair process and hence caused timeout (1 hour) for GENESIS.

## Case Study

We observed that test cases written by developers are often incomplete in practice, and VFix and GENESIS easily produce incorrect patches in such cases. For example, consider Figure 4, which describes an NPE found in project aries-jpa\textsuperscript{[10]} and the test case written by the developer. Method getJpaAnnotated (Figure 4(a)) collects all the declared fields of the input class object c retrieving its super classes. An NPE occurs at line 4 because c.getSuperClass() at line 7 may return null in case the super class is an interface. In this case, the developer wrote a single test case (the assertion in Figure 4(b)) for the purpose of patch validation, which first exposes the NPE and then checks if the length of the returned list equals to 0. This test case is incomplete as it only checks the execution where the loop iterates only once, while the fully expected behavior of the method is that it retrieves all the super classes until c.getSuperClass() returns Object.class or null. With this incomplete specification, GENESIS and VFix generated the following patches that pass the test case but are incorrect:

---
\textsuperscript{9}We made these cases publicly available for verification.

\textsuperscript{10}https://github.com/apache/aries-jpa/commit/771204
if (c.getSuper...() != null) while (c != ...) {
  if (c == null)
    return new ArrayList();
Fix by Genesis
  if (c == null)
    throw new NullPointerException();
Fix by VFix

By contrast, NPEX generated the following patch that is semantically equivalent to the developer’s fix, which was possible because NPEX can automatically infer the expected specification from the buggy code rather than relying on test cases.

while (c != ...) {
  if (c == null) break;
  & & c != null) {
Fix by NPEX
  throw new IllegalArgumentException("...");
Fix by developer

Limitations of NPEX. Next we discuss limitations of NPEX identified from the evaluation. First, NPEX failed because of unsupported fix patterns. In Figure 5, for example, the developer fixed an NPE by changing the type of a local variable from int to double. This is because the fault of this NPE is due to unsafe type conversion from double to int. Because of this unsafe conversion, the method distance returns NaN value, which causes the condition at line 9 to produce false, and therefore cluster is null at line 11.

We also identified a limitation of our specification inference. This happened when another fault exists in the buggy program other than the NPE to be fixed. For example, the program in Figure 6 has an NPE at line 4 since iterable can be null. NPEX inferred the incorrect specification for this program: “if iterable is null, then, throw NullPointerException”, which was computed by interpreting the NPE expression iterable.iterator() as null using the learned model and symbolically executing the constructor at line 6, which is through this(...) at line 4. However, there was another bug in the constructor at line 6. On the other hand, NPEX could infer a correct specification if IllegalArgumentException was thrown at line 8.

Validated Patches. We observed that NPEX can successfully validate various patches beyond simple ternary forms. Interestingly, all 21 patches fixed and validated correctly by NPEX were in the forms of SKIP or INIT. In other words, NPEX rejected all ternary patches even though we used ternary null handling code for specification mining. These bugs require SKIP or INIT patches to fix NPEs, since simply replacing an NPE expression to a ternary expression introduced a new NPE. NPEX inferred correct specification which is semantically equivalent to SKIP or INIT patches (e.g., Section 2.2).

Falsely Validated Patches. We manually investigated 38 (99-61, Table 2) false positive cases (i.e., incorrect patches accepted by the patch validator) during manual assessment of validated patches. Those cases were classified into the following cases:
- (26 cases) Inference of correct specification failed due to the existence of faults other than the target NPE (as described in Limitations of NPEX).
- (2 cases) Correct specification is inferred, but an incorrect patch is validated due to the imprecision of static symbolic execution.
- (10 cases) The learnt null-handling model returns wrong alternative expressions.

We discuss prior work closely related to ours. We focus on automated program repair (APR) approaches [16, 45], rather than techniques for detecting and mitigating NPEs (e.g., [6, 12, 39, 42, 47]). APR techniques are broadly classified into general-purpose and special-purpose techniques. Special-purpose techniques are applicable to particular kinds of bugs. For example, FootPatch [58] can fix heap-related bugs such as resource leaks, memory leaks, and null dereferences. SAVER [18], MemFix [30], and LeakFix [14] are techniques for fixing memory leaks, use-after-frees, and double-frees in C programs. Other special-purpose techniques have been proposed to fix common and important classes of bugs, e.g., concurrency bugs [1, 22, 33, 34], buffer/integer overflows [7, 19, 46, 52], error-handling bugs [57], and performance bugs [5, 51]. VFix [67] and

Figure 5: An NPE bug (line 11) and the developer patch (simplified code snippet Math-79 in Defects4J)

Figure 6: An NPE bug (line 4) and the developer patch (simplified from opengrok-6a95adb)

While the first case is the limitation of NPEX, we expect that the other cases can be resolved. The second case could be resolved by standard techniques to improve symbolic execution such as more advanced state merging heuristics. The third case could be resolved by refining the null-handling model with more features and training dataset.

6 RELATED WORK

We discuss prior work closely related to ours. We focus on automated program repair (APR) approaches [16, 45], rather than techniques for detecting and mitigating NPEs (e.g., [6, 12, 39, 42, 47]). APR techniques are broadly classified into general-purpose and special-purpose techniques. Special-purpose techniques are applicable to particular kinds of bugs. For example, FootPatch [58] can fix heap-related bugs such as resource leaks, memory leaks, and null dereferences. SAVER [18], MemFix [30], and LeakFix [14] are techniques for fixing memory leaks, use-after-frees, and double-frees in C programs. Other special-purpose techniques have been proposed to fix common and important classes of bugs, e.g., concurrency bugs [1, 22, 33, 34], buffer/integer overflows [7, 19, 46, 52], error-handling bugs [57], and performance bugs [5, 51]. VFix [67] and
NPEX [13] are NPE-specific techniques. NPEX is also specialized for fixing NPEs with its novel patch validation approach.

General-purpose approaches [21, 24, 26, 27, 29, 37, 38, 43, 44, 48, 53, 59, 60, 66] are applicable to any kinds of bugs, where most techniques rely on test cases to validate patches. General-purpose approaches are further classified into generate-and-validate [21, 24, 37, 38, 59] and semantics-based approaches [26, 27, 43, 44, 48]. Generate-and-validate techniques use search algorithms (e.g., genetic programming [59]) to iteratively generate candidate patches from a patch space until plausible patches that pass the given test suite are found. Semantics-based approaches explore the search space implicitly by generating constraints on correct patches and using SAT solvers to synthesize satisfying patches. Although these techniques are general, they are less effective for fixing specific types of bugs such as NPEs as demonstrated by Xu et al. [67].

To mitigate overfitting [28, 54, 69], existing APR techniques are often combined with patch prioritization [3, 21, 31, 50, 60, 63, 64]. For example, CapGen [60] uses context information of code (AST nodes) to rank correct patches before merely plausible ones. Prophet [38] learns a probabilistic model of correct code from a dataset of human-written patches collected from open-source software repositories, and uses the model to rank candidate patches based on the probability of being correct. VFix uses a heuristic that ranks NPE patches by solving a graph congestion problem. We believe these ranking techniques can be combined with our approach to better identify correct NPE patches.

Recently, various data-driven techniques have emerged to enhance program repair [10, 17, 31, 40, 61]. NPEX lies in this line of research, where we use data to learn a null-handling model. Notable existing data-driven techniques related to fixing NPEs are GENESIS and Getafix. GENESIS [36] uses data, a set of human patches, for search space inference. In particular, GENESIS can fix NPEs using a specialized search space learned from existing data. Getafix [4] aims to quickly generate human-like fixes for bugs detected by static analyzers. To do so, Getafix uses repair templates learned from past human patches and suggests the most appropriate fix for a given bug. Compared to NPEX, Getafix is more focused on the patch generation phase while relying on a simple ranking heuristic to select correct patches. As a result, one limitation of Getafix is that it cannot precisely infer the expected behavior of buggy code [4], which is particularly important for repairing NPEs. Our patch validation technique could be used with Getafix to better suggest NPE fixes.

Our work also lies in the line of work on identifying test-overfitted patches [15, 56, 62, 65, 70, 71]. Tan et al. [56] proposed a set of common syntactic patterns for incorrect patches, which can be used to prevent specific classes of patches that are likely to be incorrect. ODS [71] trains a statistical model to predict overfitted patches based on features that describe syntactic characteristics of correct patches. FixFit [15] focused on avoiding patches that cause crashes beyond the given input, by generating new test inputs using a grey-box fuzzing technique. Xiong et al. [65] proposed a technique to validate patches by measuring similarity or dissimilarity between a buggy program and a patched program for newly generated test inputs. Compared to these work, the goal of NPEX is focused on NPEs and presents a new approach based on learning and symbolic execution.

7 CONCLUSION

While NPEs are recurring and critical bugs in Java applications, automatically repairing NPEs still remains a significant challenge. The main difficulty is in identifying correct fixes out of a wide range of plausible patches that pass but overfitted to test cases, a central open problem in automated program repair.

In this paper, we presented a new approach to address this challenge. Instead of relying on test cases, our approach infers the expected behavior of a buggy program by combining learning and symbolic execution, and validates candidate patches against the inferred repair specification. We implemented our approach in a tool, NPEX, and showed that NPEX can fix diverse real-world NPEs more effectively than state-of-the-art test-based techniques.

Future Work. Although we focused on NPEs in this paper, our approach could be generalized to other faults. Note that the core idea of NPEX consists of two general components: (1) learning of error-handling model from codebases, and (2) validating patches using symbolic execution. The second component (symbolic execution) is already reusable for other types of faults once appropriate “error-handling model” is given. Instantiating the first component (learning of error-handling model) for each different fault is less obvious and will be interesting future work.

A good starting point for generalization would be the class of faults whose alternative semantics can be easily captured from error-handling code. For example, Class Cast Exceptions (CCEs), yet another common runtime error in Java, are such a case. In Java projects, developers handle CCEs in a way similar to NPEs, i.e., using ternary expressions with type checking guard and alternative expression. In this case, the idea of NPEX can be reused without significant changes.

ACKNOWLEDGMENTS

This work was partly supported by 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 and No.2021-0-00758, Development of Automated Program Repair Technology by Combining Code Analysis and Mining) and the MSIT(Ministry of Science and ICT), Korea, under the ICT Creative Consilience program (IITP-2022-2020-0-01819) supervised by the IITP(Institute for Information & communications Technology Planning & Evaluation), and the National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIT) (No.2021R1A5A1021944).

REFERENCES


