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:
Junhee Lee, Seongjoon Hong, and Hakjoo Oh. 2022. NPEX: Repairing Java Null Pointer Exceptions without Tests. In *44th International Conference on Software Engineering (ICSE '22), May 21–29, 2022, Pittsburgh, PA, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3510003.3510186

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.

*The first and second authors contributed equally to this work.
†Corresponding author

ICSE '22, May 21–29, 2022, Pittsburgh, PA, USA
© 2022 Association for Computing Machinery.
ACM ISBN 978-1-4503-9221-1/22/05…$15.00
https://doi.org/10.1145/3510003.3510186

Generate-and-Validate APR Approaches. Over the last decade, automated program repair (APR) techniques have shown promise in addressing the challenge of bug fixing [21, 24, 29, 37, 38, 43, 53, 59, 60, 66, 68]. Most existing APR techniques follow the conventional *generate-and-validate* approach, which alternates the two phases: (1) patch generation and (2) patch validation. In the patch generation phase, a candidate patch is selected from a pre-defined search space, and in the patch validation phase, the correctness of the candidate patch is checked by running the patched program on a set of test cases. This process is repeated until a plausible patch that passes all test cases is found.

VFix [67] is the current state-of-the-art for repairing NPEs. Its novel feature is to use value-flow information of programs to accurately localize suspicious statements and reduce the search space of candidate patches appropriately for NPEs. The reduced patch space improves the efficiency of the generate-and-validate process and also increases the chances of finding correct patches. As a result, VFix has been shown to outperform existing APR techniques such as GenProg [59], ACS [66], CapGen [60], Nopol [68], SimFix [21], and NPEfix [13] when evaluated for NPEs [67].

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

(a) Buggy program

```
1 boolean compare(int row, Column<?> temp, Column<?> org) {
2     Object o1 = org.get(row);
3     Object o2 = temp.get(temp.size() - 1);
4     return o1.equals(o2); // NPE
5 }
```

(b) NPE-triggering input

```
int missing = IntColumnType.missingValueIndicator();
Table t1 = Table.create("T1",
    IntColumn.create("Id", 0, 0),
    IntColumn.create("ChildId", missing, missing));
t1.dropDuplicateRows(); // compare is invoked inside
```

(c) NPEX-generated patch (= developer's patch)

```
(-)   return o1.equals(o2);
(+)   return o1 == null ? o2 == null : o1.equals(o2);
```

Figure 1: An NPE bug (line 4) and developer patch

```
1 String m(Object obj, String valueIfNull) {
2     // if (obj==null) return valueIfNull; // developer patch
3     Class cls = obj.getClass(); // NPE
4     String name = cls.getCanonName();
5     if (name == null) { return valueIfNull; }
6     else { return name; }
7 }
```

Figure 2: A buggy program (simplified from Apache Commons Lang) and the developer patch (commented out).

```
1 if (obj == null)
2     return null;
3 Class cls = ...;
4 String name = ...;
```

P_1 (incorrect)

```
1 if (obj == null)
2     return valueIfNull;
3 Class cls = ...;
4 String name = ...;
```

P_2 (correct)

Figure 3: Candidate patches

program; given a candidate patch during the generate-and-validate process, we analyze the behavior of the patched program to check if it satisfies the inferred specification.

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 bug[2] 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 given[3] 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

[1]https://github.com/kupl/npex
[2]https://github.com/jtablesaw/tablesaw/commit/65596d8
[3]https://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, P_1 and P_2, in Figure 3 are generated during the repair process. While both patches succeed to eliminate the given NPE, P_1 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 P_1 while validating the correctness of P_2.

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 $\mathcal{M}$, 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.getCanonName()` at line 4 should be interpreted as `null`. In summary, when instantiated for the example program in Figure 2, the model $\mathcal{M}$ can be treated as the following function:

$$\mathcal{M} = \left\{ \begin{array}{rcl} \texttt{null.getClass()} & \mapsto & \texttt{null} \\ \texttt{null.getCanonName()} & \mapsto & \texttt{null} \end{array} \right\}. \quad (1)$$

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 `rapidoid`[4]:

```
(args[i] != null) ? args[i].getClass() : null
```

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

$$\texttt{null.getClass()} \mapsto \texttt{null} \quad (2)$$

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

```
obj != null ? obj.getClass() : Object.class
```

from which we infer the following:

$$\texttt{null.getClass()} \mapsto \texttt{Object.class}.$$

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 `getCanonName` can be inferred from the following code snippet in project `dozer`[5]:

```
destCls != null ? destCls.getCanonName() : null
```

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

[4]https://github.com/rapidoid/rapidoid
[5]https://github.com/DozerMapper/dozer

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 π_{init} is the initial path condition, i.e., $\pi_{init} = \emptyset$, and σ_{init} is the initial symbolic store, i.e., $\sigma_{init} = [\texttt{obj} \mapsto \alpha, \texttt{valueIfNull} \mapsto \beta]$, that maps formal parameters to fresh symbols α and β.

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:

$$\begin{array}{rcccl} s_1 & = & (\pi_1, \sigma_1) & = & (\alpha = \texttt{null}, \sigma_{init}[\texttt{cls} \mapsto \texttt{null}]) \\ s_2 & = & (\pi_2, \sigma_2) & = & (\alpha \neq \texttt{null}, \sigma_{init}[\texttt{cls} \mapsto f(\alpha)]) \end{array}$$

State s_1 represents the output of the NPE-triggering execution where `obj` is `null` (denoted by path condition $\alpha = \texttt{null}$). In this case, store σ_1 is obtained using the null-handling model as follows:

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

where we inferred the expected meaning of the method invocation using $\mathcal{M}$. State s_2 represents the normal execution where no NPE occurs. We use an uninterpreted function symbol to represent the return value of the external method; $f(\alpha)$ denotes the symbolic value that `obj.getClass()` evaluates to.

With s_1 and s_2 as input states, symbolic execution of line 4 results in the following output states:

$$\begin{array}{rcccl} s_3 & = & (\pi_3, \sigma_3) & = & (\alpha = \texttt{null}, \sigma_1[\texttt{name} \mapsto \texttt{null}]) \\ s_4 & = & (\pi_4, \sigma_4) & = & (\alpha \neq \texttt{null}, \sigma_2[\texttt{name} \mapsto g(f(\alpha))]) \end{array}$$

Note that the execution with s_1 encounters a new NPE because `cls` holds `null` in s_1 and method `getCanonName` is called on it. Thus, we use the null-handling model again to infer the expected behavior of `null.getCanonName` and obatin store σ_3 as follows:

$$\sigma_3 = \sigma_1[\texttt{name} \mapsto \mathcal{M}(\texttt{null.getCannonName})] = \sigma_1[\texttt{name} \mapsto \texttt{null}].$$

State s_4 is the result of executing line 4 with input state s_2, where $g(f(\alpha))$ is the symbolic value representing `cls.getCanonName()`.

We complete symbolic execution by analyzing the if statement at line 5 with states s_3 and s_4 as input, which produces the following three states as output:

$$\begin{array}{rcl} s_5 & = & (\alpha = \texttt{null}, \sigma_3[ret \mapsto \beta]) \\ s_6 & = & (\alpha \neq \texttt{null} \land g(f(\alpha)) = \texttt{null}, \sigma_4[ret \mapsto \beta]) \\ s_7 & = & (\alpha \neq \texttt{null} \land g(f(\alpha)) \neq \texttt{null}, \sigma_4[ret \mapsto g(f(\alpha))]) \end{array}$$

The input state s_3 results in s_5 taking only the true branch as the value of `name` is `null` in s_3. With s_4, we consider both true and false branches, producing s_6 and s_7, respectively. Finally, we obtain the symbolic summary $S_{\mathtt{m}}$ of method m by removing information about local variables in those states:

$$S_{\mathtt{m}} = \left\{ \begin{array}{l} (\alpha = \mathtt{null}, \sigma_{init}[ret \mapsto \beta]), \\ (\alpha \neq \mathtt{null} \land g(f(\alpha)) = \mathtt{null}, \sigma_{init}[ret \mapsto \beta]), \\ (\alpha \neq \mathtt{null} \land g(f(\alpha)) \neq \mathtt{null}, \sigma_{init}[ret \mapsto g(f(\alpha))]) \end{array} \right\}$$

We consider the symbolic summary $S_{\mathtt{m}}$ 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 $S_{\mathtt{m}}$. We conclude that a candidate patch is correct iff running a normal symbolic execution on it produces a summary equivalent to $S_{\mathtt{m}}$. For example, the summary S_1 of the incorrect patch P_1 in Figure 3 is computed as follows:

$$S_1 = \left\{ \begin{array}{l} (\alpha = \mathtt{null}, \sigma_{init}[ret \mapsto \mathtt{null}]) \\ (\alpha \neq \mathtt{null} \land g(f(\alpha)) = \mathtt{null}, \sigma_{init}[ret \mapsto \beta]), \\ (\alpha \neq \mathtt{null} \land g(f(\alpha)) \neq \mathtt{null}, \sigma_{init}[ret \mapsto g(f(\alpha))]) \end{array} \right\}$$

Note that S_1 and $S_{\mathtt{m}}$ do not agree with the return value, (ret) when α is null, i.e., $\sigma_{init}[ret \mapsto \mathtt{null}] \neq \sigma_{init}[ret \mapsto \beta]$, 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_m. 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 p_m and $\mathrm{body}(m)$ 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:

$$\begin{array}{l} S \rightarrow x = e \mid \mathtt{return}\ e \mid \mathtt{if}\ E\ S_1\ S_2 \mid \mathtt{while}\ E\ S \mid S_1; S_2 \mid \epsilon \\ E \rightarrow n \mid \mathtt{null} \mid x \mid x.m(y) \mid \mathtt{new}\ C() \mid E_1\ ?\ E_2 : E_3 \mid E_1 == E_2 \end{array}$$

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 $\mathrm{type}(e)$ 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_{NPE} for the set of expressions where NPEs may occur; in our language, E_{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

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

from a dataset of programs. Given a program $P \in Pgm$ and its NPE-triggering expression $e_{\mathsf{NPE}} \in E_{\mathsf{NPE}}$, $\mathcal{M}_P(e_{\mathsf{NPE}})$ predicts an alternative expression that can be used as a substitute for e_{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 $\mathbf{D}$ of null-handling patterns from a codebase. Let $\mathbf{P} = \{P_1, P_2, \ldots, P_m\}$ be a collection of programs. The dataset $\mathbf{D}$ is of the type $\mathbf{D} \subseteq Pgm \times E_{\mathsf{NPE}} \times E$. That is, $\mathbf{D}$ is a set of tuples (P, e_{NPE}, e) where $P \in \mathbf{P}$ is a program in the codebase, e_{NPE} is an NPE expression in P, and e is an expression that is alternatively used in P when e_{NPE} causes an NPE.

To collect the dataset $\mathbf{D}$, we traverse the abstract syntax tree of each program $P \in \mathbf{P}$ and observe how NPEs are handled. For example, from ternary expression of the form $x == \mathtt{null}\ ?\ e\ :\ x.m(y)$, we collect tuple $(P, x.m(y), e)$, 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 $x == \mathtt{null} \mid\mid x.m(y)$ is translated into $x == \mathtt{null}\ ?\ \mathtt{true}\ :\ x.m(y)$ and represented by tuple $(P, x.m(y), \mathtt{true})$.

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:

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

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

$$\widehat{E} = \mathbb{N}_{\mathbf{D}} \cup \{\mathtt{null}, \mathsf{ARG}, \mathsf{NEW}\} \cup \{\mathsf{ARG} == \widehat{e} \mid \widehat{e} \in \mathbb{N}_{\mathbf{D}} \cup \{\mathtt{null}\}\} \cup \{\top\}$$

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

$$\alpha(e_{\mathsf{NPE}}, e) = \left\{ \begin{array}{ll} e & \cdots\ e \in \mathbb{N}_{\mathbf{D}} \cup \{\mathtt{null}\} \\ \mathsf{NEW} & \cdots\ e = \mathtt{new}\ C() \\ \mathsf{ARG} & \cdots\ e_{\mathsf{NPE}} = x.m(y) \land e = y \\ \mathsf{ARG} == n & \cdots\ e_{\mathsf{NPE}} = x.m(y) \land e = (y == n) \land n \in \mathbb{N}_{\mathbf{D}} \cup \{\mathtt{null}\} \\ \top & \cdots\ otherwise \end{array} \right.$$

With α, the generalized dataset $\widehat{\mathbf{D}}$ is obtained as follows:

$$\widehat{\mathbf{D}} = \{(P, e_{\mathsf{NPE}}, \alpha(e_{\mathsf{NPE}}, e)) \mid (P, e_{\mathsf{NPE}}, e) \in \mathbf{D}\}.$$

Feature Representation. Next, we represent NPE expressions as feature vectors to generate the training data $\mathcal{D}$:

$$\mathcal{D} \subseteq \{0,1\}^n \times \widehat{E}.$$

To do so, we assume n boolean features: $\Phi = \{\phi_1, \phi_2, \ldots, \phi_n\}$, where each feature $\phi_i : Pgm \times E_{\mathsf{NPE}} \rightarrow \{0,1\}$ 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 $\Phi(P, e_{\mathsf{NPE}})$ for the feature vector of (P, e_{NPE}): $\Phi(P, e_{\mathsf{NPE}}) = \langle \phi_1(P, e_{\mathsf{NPE}}), \phi_2(P, e_{\mathsf{NPE}}), \ldots, \phi_n(P, e_{\mathsf{NPE}})\rangle$. With Φ, we can generate the training data $\mathcal{D}$ as follows:

$$\mathcal{D} = \{(\Phi(P, e_{\mathsf{NPE}}), \widehat{e}) \mid (P, e_{\mathsf{NPE}}, \widehat{e}) \in \widehat{\mathbf{D}}\}.$$

We use 31 features in Table 1. Here, each feature is a predicate on method calls, $x.m(y)$, since E_{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 whether a 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 $\widehat{E}$ for a given feature vector. Let Pr be the learned probability distribution; given a program P, an NPE expression e_{NPE}, and abstract expression $\widehat{e} \in \widehat{E}$, $\Pr(\widehat{e} \mid \Phi(P, e_{\mathsf{NPE}}))$ denotes the probability of the alternative expression of e_{NPE} being $\widehat{e}$. We computed Pr using an off-the-shelf learning algorithm for Random Forest Classifier.

We can construct the null-handling model $\mathcal{M}$ from Pr, with an additional process that concretizes an abstract expression into a type-compatible concrete expression. We define $\gamma : E_{\mathsf{NPE}} \times \widehat{E} \rightarrow E \cup \{\bot\}$ for concretization, which converts an abstract expression into a concrete expression as follows:

$$\gamma(x.m(y), \widehat{e}) = \begin{cases} n & \cdots \quad \widehat{e} = n \wedge T = \texttt{int} \\ \texttt{null} & \cdots \quad \widehat{e} = \texttt{null} \wedge T \neq \texttt{int} \\ \texttt{new}\, C() & \cdots \quad \widehat{e} = \mathsf{NEW} \wedge T = C \\ y & \cdots \quad \widehat{e} = \mathsf{ARG} \wedge \mathsf{type}(y) = T \\ y == e & \cdots \quad \widehat{e} = \mathsf{ARG} == e \wedge T = \texttt{int} \\ \bot & \cdots \quad \textit{otherwise} \end{cases}$$

where T denotes the type of expression $x.m(y)$. With γ, our null-handling model $\mathcal{M} : Pgm \times E_{\mathsf{NPE}} \rightarrow E$ is defined as follows:

$$\mathcal{M}_P(e_{\mathsf{NPE}}) = \gamma(e_{\mathsf{NPE}}, \underset{\widehat{e} \in C}{\operatorname{argmax}} \Pr(\widehat{e} \mid \Phi(P, e_{\mathsf{NPE}})))$$

where $C = \{\widehat{e} \in \widehat{E} \mid \gamma(e_{\mathsf{NPE}}, \widehat{e}) \neq \bot\}$ is the set of concretizable expressions and we pick one with the highest probability.

Table 1: Features for method invocations

Class	#	Description
Name Features	1 - 20	1."Code", 2."hash", 3."append", 4."equals", 5."on", 6."Error", 7."Success", 8."get", 9."set", 10."is", 11."add". 12."close", 13."Empty", 14."Value", 15."put" 16."String", 17."to", 18."remove", 19."write", 20."contains"
Body Features	21	return type is void
	22	method returns a literal
	23	thrown exceptions are annotated
	24	null check expression exists
	25	method returns a constructor call
	26	method is the base of another invocation
	27	method returns a field
Context Features	28	caller method is private
	29	null pointer is assigned to an array
	30	null pointer is assigned to a field
	31	null pointer is assigned to a public field

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 $P_{\mathsf{NPE}} \in Pgm$ is given and the buggy (NPE-triggering) expression in P_{NPE} is $x_{\mathsf{NPE}}.m(y)$, where x_{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 $\mathcal{M}$ 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 ($x_{\mathsf{NPE}}.m(y)$) 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 Σ from methods to summaries:

$$\Sigma \in \textit{SumTable} = \textit{Method} \rightarrow \textit{Summary}$$

where a summary is defined as follows:

$$\begin{array}{rcccl} S & \in & \textit{Summary} & = & \mathcal{P}(\textit{State}) \\ s & \in & \textit{State} & = & \textit{PC} \times \textit{Store} \\ \pi & \in & \textit{PC} & = & \mathcal{P}(\textit{SymVal} \times \{=, \neq\} \times \textit{SymVal}) \\ \sigma & \in & \textit{Store} & = & \textit{Var} \rightarrow \textit{SymVal} \\ v & \in & \textit{SymVal} & = & \mathbb{Z} + \textit{Class} + \{\texttt{null}\} + \textit{Symbol} \end{array}$$

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 $SymExec : Pgm \rightarrow SumTable$, to be compositional and bounded [11]. It analyzes each method of a program only once by calculating its summary in isolation using summaries of callee 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 `if` statements. With these assumptions, it is enough to define symbolic execution for the following subset of statements and expressions:

$$\begin{array}{lcl} S & \rightarrow & x := e \mid \texttt{return}\ x \mid \texttt{if}\ (x == y)\ S_1\ S_2 \mid S_1; S_2 \\ E & \rightarrow & n \mid \texttt{null} \mid x \mid x.f(y) \mid \texttt{new}\ C() \end{array}$$

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

$$SymExec(P) = [m_i \mapsto F(\text{body}(m_i), \Sigma^i, s^i_{init})]^n_{i=1}$$

where $m_1, \dots, m_n$ are a sequence of methods sorted according to the reverse topological order of the call-graph, Σ^i is the partial summary table for methods $m_1, \dots, m_{i-1}$, and the initial state s^i_{init} is defined by $(\emptyset, [p_{m_i} \mapsto \alpha_{m_i}])$ which indicates that the formal parameter p_{m_i} of method m_i is bound to a fresh symbolic value α_{m_i}. The semantic function $F : Stmt \times SumTable \times State \rightarrow \mathcal{P}(State)$ is defined in a standard manner as follows:

$$\begin{array}{l} F(x := n, \Sigma, (\pi, \sigma)) = \{(\pi, \sigma[x \mapsto n])\} \\ F(x := \texttt{null}, \Sigma, (\pi, \sigma)) = \{(\pi, \sigma[x \mapsto \texttt{null}])\} \\ F(x := y, \Sigma, (\pi, \sigma)) = \{(\pi, \sigma[x \mapsto \sigma(y)])\} \\ F(x := \texttt{new}\ C(), \Sigma, (\pi, \sigma)) = \{(\pi, \sigma[x \mapsto C])\} \\ F(x := y.m(z), \Sigma, (\pi, \sigma)) = \{(\pi \cup \pi^i_m, \sigma[x \mapsto \sigma^i_m(ret_m)])\}_i \\ F(S_1; S_2, \Sigma, s) = \bigcup\{F(S_2, \Sigma, s') \mid s' \in F(S_1, \Sigma, s)\} \\ F(\texttt{if}(x == y)\ S_1\ S_2, \Sigma, (\pi, \sigma)) = F(S_1, \Sigma, s_1) \cup F(S_2, \Sigma, s_2) \end{array}$$

where $s_1, s_2 = (\pi \cup \{\sigma(x) = \sigma(y)\}, \sigma), (\pi \cup \{\sigma(x) \neq \sigma(y)\}, \sigma)$ and $(\pi^i_m, \sigma^i_m) \in \Sigma(m)[\alpha_m \mapsto \sigma(z)]$.

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

$$SymExec^{x_{\text{NPE}}}_{\mathcal{M}}(P) = [m_i \mapsto F^{x_{\text{NPE}}}_{\mathcal{M}}(\text{body}(m_i), \Sigma^i, s^i_{init})]^n_{i=1}.$$

The extended semantic function $F^{x_{\text{NPE}}}_{\mathcal{M}}(S, \Sigma, (\pi, \sigma))$ 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., $x := y.m(z)$, where y is the fault variable x_{NPE}. In this case, $F^{x_{\text{NPE}}}_{\mathcal{M}}(S, \Sigma, (\pi, \sigma))$ produces

$$F(S, \Sigma, (\pi_{\texttt{nonnull}}, \sigma)) \cup F(x := \mathcal{M}_P(y.m(z)), \Sigma, (\pi_{\text{NPE}}, \sigma))$$

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

$$F^{x_{\text{NPE}}}_{\mathcal{M}}(S, \Sigma, (\pi, \sigma)) = F(x := \mathcal{M}_{P_{\text{NPE}}}(y.m(z)), \Sigma, (\pi, \sigma)).$$

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.

```
A foo(p) { return p.hoo(0); // NPE }
int goo(z) {
  A x = this.foo(z);
  return x.goo(); }
```

Suppose that NPE occurs at line 1 because `p` is `null` and the alternative expression inferred by the model is `null` for the NPE-triggering expression `p.hoo(0)`. Then, the inferred procedural summary of `foo` is $\{(\alpha_{\texttt{foo}} = \texttt{null}, [ret_{\texttt{foo}} \mapsto \texttt{null}])\}$, where $\alpha_{\texttt{foo}}$ and $ret_{\texttt{foo}}$ denote the symbolic parameter and return variable, respectively. Then, the return value `null` propagates to caller's variable `x`. So the model $\mathcal{M}$ is applied again at line 4, and we get the summary of `goo` as $\{(\alpha_{\texttt{goo}} = \texttt{null}, [ret_{\texttt{goo}} \mapsto v])\}$ where we assume v is the alternative value for `x.goo()` obtained by the model $\mathcal{M}$.

Specification Validation. The next step is to validate candidate patches against the inferred specification. Let P_{NPE} be a buggy program and P_{cand} be a patch candidate. We would like to determine whether P_{cand} is a correct patch of P_{NPE}. We do this by checking the equivalence: $SymExec^{x_{\text{NPE}}}_{\mathcal{M}}(P_{\text{NPE}}) \equiv SymExec(P_{\text{cand}})$, where the left-hand side denotes the inferred repair specification of the buggy program. We say that two summary tables Σ_1 and Σ_2 are equivalent, denoted $\Sigma_1 \equiv \Sigma_2$, if the following holds for all methods m:

$$\bigwedge_{(\pi_1,\sigma_1) \in \Sigma_1(m)} \bigwedge_{(\pi_2,\sigma_2) \in \Sigma_2(m)} \pi_1 \wedge \pi_2 \implies \sigma_1 = \sigma_2$$

where we assume the symbolic value for the formal parameter of method m is consistently named (e.g., α_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(0)` to `p == null ? null : p.hoo()` is correct for method `foo`, but it introduces a new NPE in `goo`. In this case, we can successfully reject this patch by checking the semantic equivalence for `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 Infer [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_{NPE}, an NPE stack trace

(x_{NPE}, τ), and a null-handling model $\mathcal{M}$ as input, the algorithm produces a set of validated patches to fix the bug. At line 1, it first runs the buggy program with the crashing input to get the fault variable x_{NPE} and stack trace τ. At line 2, the buggy program P_{NPE} is analyzed by our variant of symbolic execution, $SymExec^{x_{\mathsf{NPE}}}_{\mathcal{M}}$, which returns the repair specification, $\Sigma_{inferred}$, of the buggy program. At line 3, we compute a set X of candidate faults. We iterate each candidate fault x'_{NPE} in X, and accumulate validated patches in *Patches*. At line 6, we enumerate patches with given fault x'_{NPE}, and for each patch $P_{patched}$, we compute its summary table $\Sigma_{patched}$ by normal symbolic execution. If the summary table of $P_{patched}$ is equivalent to the inferred specification $\Sigma_{inferred}$, we add it to *Patches*.

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_{NPE} and stack trace τ only, FaultLocalization computes a set of null pointer expressions that may be alias to x_{NPE} for each method in the stack trace τ. 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_{NPE}, PatchEnumeration enumerates patches based on pre-defined templates. We used the following templates from prior work [13, 67]:

- SKIP: (i) `if` (x_{NPE} `!= null`) S, or (ii) `if` (x_{NPE} `== null`) fb
- REPLACE: x_{NPE} `== null?` e `:` e_{NPE}
- INIT: `if` (x_{NPE} `== null`) $x_{\mathsf{NPE}} = e$

SKIP skips a statement or a block (S) containing an NPE (SKIP-(i)), or inserts a control flow break (fb): `break`, `continue`, `return` e, or `throw` Exn (SKIP-(ii)). REPLACE substitutes an expression involving the fault expression (e_{NPE}) with a ternary with an alternative expression (e). INIT initializes a null pointer to a fresh object obtained by calling a constructor (e). For expressions (e), 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 (Exn) 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 unrolled each loop of programs twice.

Algorithm 1 The NPEX Algorithm

```
Input: Buggy program $P_{NPE}$, NPE stack trace $(x_{NPE}, \tau)$, model $\mathcal{M}$
Output: A set Patches of validated patches
1:  $x_{NPE}, \tau \leftarrow$ RunProgram($P_{NPE}, I$)        ▷ Fault variable and stack trace
2:  Patches $\leftarrow \emptyset$
3:  $\Sigma_{inferred} \leftarrow SymExec^{x_{NPE}}_{\mathcal{M}}(P_{NPE})$        ▷ Spec inference
4:  $X \leftarrow$ FaultLocalization($P_{NPE}, \tau$)
5:  for $x'_{NPE} \in X$ do
6:      for $P_{patched} \in$ PatchEnumeration($P_{NPE}, x'_{NPE}$) do
7:          $\Sigma_{patched} \leftarrow SymExec(P_{patched})$
8:          if ($\Sigma_{patched} \equiv \Sigma_{inferred}$) then        ▷ Spec validation
9:              Patches $\leftarrow$ Patches $\cup \{P_{patched}\}$
10:         end if
11:     end for
12: end for
13: return Patches
```

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; ... if (x_NPE != null) { y = x_NPE.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].

- GenBM: 16 NPE bugs from [36].
- BearsBM: 14 NPE bugs from Bears [41].
- OurBM: 59 NPE bugs from open-source repositories.

VFixBM, GenBM, and BearsBM came from prior work. VFixBM consists of 30 NPE bugs, of which 15 is from Defects4J [23] and another 15 from open-source repositories collected by the authors of VFix. We note that the structures of the projects in VFixBM are modified by the authors of VFix to easily run VFix. They removed the build system in the original projects, and wrote a compilable Main.java that acts as a test suite. We used VFixBM as they are provided without any modification. We also collected NPE bugs used for evaluating Genesis [36] and contained in Bears [41]. Both of these benchmarks consist of NPE bugs collected from open-source projects in GitHub and use the apache maven project management system[6]. From [36, 41], we only collected benchmarks that can be built in our environment and parsed by Spoon [49] and Infer [20]. Finally, 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[7]. Among them, we only considered projects that can be built by 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[8]. 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

[6]https://maven.apache.org/
[7]https://github.com/apache/
[8]http://www.cs.toronto.edu/~fanl/program_repair/genesis-rep/index.html

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 ($\frac{\#C}{\#R}$).

Benchmarks		NPEX					NPEX$_{base}$					Genesis [36]					VFix [67]				
Name	#Bug	#R	#G	#C	Prec	FixR	#R	#G	#C	Prec	FixR	#R	#G	#C	Prec	FixR	#R	#G	#C	Prec	FixR
VFixBM	30	30	28	19	68 %	63 %	30	30	6	20 %	20 %	0	n/a	n/a	n/a	n/a	30	26	20	77 %	67 %
GenBM	16	16	16	10	63 %	63 %	16	14	3	21 %	19 %	16	11	8	73 %	50 %	2	2	1	50 %	50 %
BearsBM	14	14	11	6	55 %	43 %	14	10	3	30 %	21 %	14	9	3	33 %	21 %	2	1	0	0 %	0 %
OurBM	59	59	44	26	59 %	44 %	59	47	16	34 %	27 %	59	33	9	27 %	15 %	23	15	3	20 %	13 %
Total	119	119	99	61	**62** %	**51** %	119	101	28	**28** %	**24** %	89	53	20	**38** %	**22** %	57	44	24	**55** %	**42** %

manually translated the generated IR to the source code with the same semantics, and then checked the correctness. We re-evaluated the patches labeled by existing works with the same criteria above. We found that 4 of VFix-generated patches labeled as correct by the authors are actually incorrect under the criteria (i.e., patches were not semantically equivalent to the developer's fix). Thus, we labeled them as incorrect in our evaluation.[9]

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% ($\frac{61}{119}$) and a precision (i.e., how precisely generated patches turned out to be correct) of 62% ($\frac{61}{99}$). On the other hand, NPEX$_{base}$, 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% ($\frac{20}{53}$). 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% ($\frac{24}{57}$) 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.

Scalability. The sizes of programs in our benchmarks range from 2K to 340KLoC (75KLoC on average). Excluding the build time by Infer, NPEX took 173.8 seconds to fix a bug on average. Specifically, it took 65 seconds for fault localization, 25.8 seconds for specification inference, and 84.0 seconds for validating patch candidates on average. Note that the time cost for running tests, 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.

(a) Buggy Program

```
1  public List getJpaAnnotated(Class c, ...) {
2     final List jpaAnnotated = new ArrayList<>();
3     while (c != Object.class) {
4        for (Field f: c.getDeclaredFields()) { // NPE
5           jpaAnnotated.add(...);
6        }
7        c = c.getSuperclass()
8     ...
9     return jpaAnnoated;
10 }
```

(b) Developer's test case

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

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

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`[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:

[9] We made these cases publicly available for verification.

[10] https://github.com/apache/aries-jpa/commit/771204

```
if (c.getSuper...() != null)
  while (c != ...) { ... }
```

Fix by Genesis

```
while (c != ...) {
  if (c == null)
    return new ArrayList();
```

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;
```

Fix by NPEX

```
while (c != Object.class
                  && c != null) {
```

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` were 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 assesment 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 decribed 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.

```
1       double distance(int[] p1, int[] p2) {
2   (-)     int sum = 0;
3   (+)     double sum = 0;
4           for (int i = 0; i < p1.length; i++) { sum += ...; }
5           return Math.sqrt(sum); // return NaN
6       }
7       void assignPointsToClusters(T point) {
8           Cluster cluster = null;
9           if (point.distance(...) < Double.MAX_VALUE) // false
10              cluster = ...;
11          cluster.addPoint(p) // NPE
12      }
```

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

```
1      public IteratorReader(Iterable<String> iterable) {
2   (+)    if (iterable == null)
3   (+)        throw new IllegalArgumentException("...");
4          this(iterable.iterator()); // NPE
5      }
6      public IteratorReader(Iterator<String> iterator) {
7          if (iterator == null)
8   (-)        throw new NullPointerException();
9   (+)        throw new IllegalArgumentException("...");
10         this.iterator = iterator;
11     }
```

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

NPEFix [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 SMT 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, 38, 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. Fix2Fit [15] focused on avoiding patches that cause crashes beyond the given input, by generating new test inputs using a greybox 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

[1] Christoffer Quist Adamsen, Anders Møller, Rezwana Karim, Manu Sridharan, Frank Tip, and Koushik Sen. 2017. Repairing Event Race Errors by Controlling Nondeterminism. In *Proceedings of the 39th International Conference on Software Engineering* (Buenos Aires, Argentina) *(ICSE '17)*. IEEE Press, Piscataway, NJ, USA, 289–299. https://doi.org/10.1109/ICSE.2017.34
[2] Nick Andrews. [n.d.]. We Crunched 1 Billion Java Logged Errors – Here's What Causes 97% of Them. https://www.overops.com/blog/we-crunched-1-billion-java-logged-errors-heres-what-causes-97-of-them-2/.
[3] Moumita Asad, Kishan Kumar Ganguly, and Kazi Sakib. 2019. Impact Analysis of Syntactic and Semantic Similarities on Patch Prioritization in Automated Program Repair. In *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 328–332. https://doi.org/10.1109/ICSME.2019.00050

[4] Johannes Bader, Andrew Scott, Michael Pradel, and Satish Chandra. 2019. Getafix: Learning to Fix Bugs Automatically. *Proc. ACM Program. Lang.* 3, OOPSLA, Article 159 (Oct. 2019), 27 pages. https://doi.org/10.1145/3360585

[5] A. Banerjee, L. K. Chong, C. Ballabriga, and A. Roychoudhury. 2018. EnergyPatch: Repairing Resource Leaks to Improve Energy-Efficiency of Android Apps. *IEEE Transactions on Software Engineering* 44, 5 (May 2018), 470–490. https://doi.org/10.1109/TSE.2017.2689012

[6] Subarno Banerjee, Lazaro Clapp, and Manu Sridharan. 2019. 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* (Tallinn, Estonia) *(ESEC/FSE 2019)*. ACM, New York, NY, USA, 740–750. https://doi.org/10.1145/3338906.3338919

[7] Xi Cheng, Min Zhou, Xiaoyu Song, Ming Gu, and Jiaguang Sun. 2017. IntPTI: Automatic Integer Error Repair with Proper-type Inference. In *Proceedings of the 32Nd IEEE/ACM International Conference on Automated Software Engineering* (Urbana-Champaign, IL, USA) *(ASE 2017)*. IEEE Press, Piscataway, NJ, USA, 996–1001. http://dl.acm.org/citation.cfm?id=3155562.3155693

[8] Maciej Cielecki, Jundefineddrzej Fulara, Krzysztof Jakubczyk, and Łukasz Jancewicz. 2006. Propagation of JML Non-Null Annotations in Java Programs. In *Proceedings of the 4th International Symposium on Principles and Practice of Programming in Java* (Mannheim, Germany) *(PPPJ '06)*. Association for Computing Machinery, New York, NY, USA, 135–140. https://doi.org/10.1145/1168054.1168073

[9] Roberta Coelho, Lucas Almeida, Georgios Gousios, and Arie van Deursen. 2015. 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*. 134–145. https://doi.org/10.1109/MSR.2015.20

[10] Elizabeth Dinella, Hanjun Dai, Ziyang Li, Mayur Naik, Le Song, and Ke Wang. 2020. Hoppity: Learning Graph Transformations to Detect and Fix Bugs in Programs. In *8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020*. OpenReview.net. https://openreview.net/forum?id=SJeqs6EFvB

[11] Dino Distefano, Manuel Fähndrich, Francesco Logozzo, and Peter W. O'Hearn. 2019. Scaling Static Analyses at Facebook. *Commun. ACM* 62, 8 (July 2019), 62–70. https://doi.org/10.1145/3338112

[12] Kinga Dobolyi and Westley Weimer. 2008. Changing Java's Semantics for Handling Null Pointer Exceptions. In *2008 19th International Symposium on Software Reliability Engineering (ISSRE)*. 47–56. https://doi.org/10.1109/ISSRE.2008.59

[13] T. Durieux, B. Cornu, L. Seinturier, and M. Monperrus. 2017. Dynamic patch generation for null pointer exceptions using metaprogramming. In *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 349–358. https://doi.org/10.1109/SANER.2017.7884635

[14] Qing Gao, Yingfei Xiong, Yaqing Mi, Lu Zhang, Weikun Yang, Zhaoping Zhou, Bing Xie, and Hong Mei. 2015. Safe Memory-leak Fixing for C Programs. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1* (Florence, Italy) *(ICSE '15)*. IEEE Press, Piscataway, NJ, USA, 459–470. http://dl.acm.org/citation.cfm?id=2818754.2818812

[15] Xiang Gao, Sergey Mechtaev, and Abhik Roychoudhury. 2019. Crash-avoiding program repair. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 8–18.

[16] L. Gazzola, D. Micucci, and L. Mariani. 2019. Automatic Software Repair: A Survey. *IEEE Transactions on Software Engineering* 45, 1 (Jan 2019), 34–67. https://doi.org/10.1109/TSE.2017.2755013

[17] Jacob Harer, Onur Ozdemir, Tomo Lazovich, Christopher P. Reale, Rebecca L. Russell, Louis Y. Kim, and Sang Peter Chin. 2018. Learning to Repair Software Vulnerabilities with Generative Adversarial Networks. In *Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018, NeurIPS 2018, December 3-8, 2018, Montréal, Canada*, Samy Bengio, Hanna M. Wallach, Hugo Larochelle, Kristen Grauman, Nicolò Cesa-Bianchi, and Roman Garnett (Eds.). 7944–7954. https://proceedings.neurips.cc/paper/2018/hash/68abef8ee1ac9b664a90b0bbaff4f770-Abstract.html

[18] Seongjoon Hong, Junhee Lee, Jeongsoo Lee, and Hakjoo Oh. 2020. SAVER: Scalable, Precise, and Safe Memory-Error Repair. In *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*. 271–283. https://doi.org/10.1145/3377811.3380323

[19] Zhen Huang, David Lie, Gang Tan, and Trent Jaeger. 2019. Using Safety Properties to Generate Vulnerability Patches. In *2019 IEEE Symposium on Security and Privacy (SP)*. 539–554. https://doi.org/10.1109/SP.2019.00071

[20] Facebook Inc. 2018. A tool to detect bugs in Java and C/C+++/Objective-C code before it ships. Available: https://fbinfer.com.

[21] Jiajun Jiang, Yingfei Xiong, Hongyu Zhang, Qing Gao, and Xiangqun Chen. 2018. Shaping Program Repair Space with Existing Patches and Similar Code. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Amsterdam, Netherlands) *(ISSTA 2018)*. Association for Computing Machinery, New York, NY, USA, 298–309. https://doi.org/10.1145/3213846.3213871

[22] Guoliang Jin, Linhai Song, Wei Zhang, Shan Lu, and Ben Liblit. 2011. Automated Atomicity-violation Fixing. In *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation* (San Jose, California, USA) *(PLDI '11)*. ACM, New York, NY, USA, 389–400. https://doi.org/10.1145/1993498.1993544

[23] René Just, Darioush Jalali, and Michael D. Ernst. 2014. Defects4J: A Database of Existing Faults to Enable Controlled Testing Studies for Java Programs. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis* (San Jose, CA, USA) *(ISSTA 2014)*. ACM, New York, NY, USA, 437–440. https://doi.org/10.1145/2610384.2628055

[24] Dongsun Kim, Jaechang Nam, Jaewoo Song, and Sunghun Kim. 2013. Automatic Patch Generation Learned from Human-written Patches. In *Proceedings of the 2013 International Conference on Software Engineering* (San Francisco, CA, USA) *(ICSE '13)*. IEEE Press, Piscataway, NJ, USA, 802–811. http://dl.acm.org/citation.cfm?id=2486788.2486893

[25] Anil Koyuncu, Kui Liu, Tegawendé F. Bissyandé, Dongsun Kim, Martin Monperrus, Jacques Klein, and Yves Le Traon. 2019. IFixR: Bug Report Driven Program Repair. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Tallinn, Estonia) *(ESEC/FSE 2019)*. Association for Computing Machinery, New York, NY, USA, 314–325. https://doi.org/10.1145/3338906.3338935

[26] Xuan-Bach D. Le, Duc-Hiep Chu, David Lo, Claire Le Goues, and Willem Visser. 2017. JFIX: Semantics-based Repair of Java Programs via Symbolic PathFinder. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Santa Barbara, CA, USA) *(ISSTA 2017)*. ACM, New York, NY, USA, 376–379. https://doi.org/10.1145/3092703.3098225

[27] Xuan-Bach D. Le, Duc-Hiep Chu, David Lo, Claire Le Goues, and Willem Visser. 2017. S3: Syntax- and Semantic-guided Repair Synthesis via Programming by Examples. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering* (Paderborn, Germany) *(ESEC/FSE 2017)*. ACM, New York, NY, USA, 593–604. https://doi.org/10.1145/3106237.3106309

[28] Xuan-Bach D. Le, Ferdian Thung, David Lo, and Claire Le Goues. 2018. Overfitting in Semantics-Based Automated Program Repair. In *Proceedings of the 40th International Conference on Software Engineering* (Gothenburg, Sweden) *(ICSE '18)*. Association for Computing Machinery, New York, NY, USA, 163. https://doi.org/10.1145/3180155.3182536

[29] Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, and Westley Weimer. 2012. GenProg: A Generic Method for Automatic Software Repair. *IEEE Transactions on Software Engineering* 38, 1 (2012), 54–72. https://doi.org/10.1109/TSE.2011.104

[30] Junhee Lee, Seongjoon Hong, and Hakjoo Oh. 2018. MemFix: Static Analysis-based Repair of Memory Deallocation Errors for C. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Lake Buena Vista, FL, USA) *(ESEC/FSE 2018)*. ACM, New York, NY, USA, 95–106. https://doi.org/10.1145/3236024.3236079

[31] Yi Li, Shaohua Wang, and Tien N. Nguyen. 2020. DLFix: Context-Based Code Transformation Learning for Automated Program Repair. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering* (Seoul, South Korea) *(ICSE '20)*. Association for Computing Machinery, New York, NY, USA, 602–614. https://doi.org/10.1145/3377811.3380345

[32] Zhenmin Li, Lin Tan, Xuanhui Wang, Shan Lu, Yuanyuan Zhou, and Chengxiang Zhai. 2006. Have Things Changed Now? An Empirical Study of Bug Characteristics in Modern Open Source Software. In *Proceedings of the 1st Workshop on Architectural and System Support for Improving Software Dependability* (San Jose, California) *(ASID '06)*. Association for Computing Machinery, New York, NY, USA, 25–33. https://doi.org/10.1145/1181309.1181314

[33] Huarui Lin, Zan Wang, Shuang Liu, Jun Sun, Dongdi Zhang, and Guangning Wei. 2018. PFix: Fixing Concurrency Bugs Based on Memory Access Patterns. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering* (Montpellier, France) *(ASE 2018)*. ACM, New York, NY, USA, 589–600. https://doi.org/10.1145/3238147.3238198

[34] Haopeng Liu, Yuxi Chen, and Shan Lu. 2016. Understanding and Generating High Quality Patches for Concurrency Bugs. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering* (Seattle, WA, USA) *(FSE 2016)*. Association for Computing Machinery, New York, NY, USA, 715–726. https://doi.org/10.1145/2950290.2950309

[35] Fan Long, Peter Amidon, and Martin Rinard. 2016. Automatic inference of code transforms and search spaces for automatic patch generation systems. (2016).

[36] Fan Long, Peter Amidon, and Martin Rinard. 2017. Automatic inference of code transforms for patch generation. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2017, Paderborn, Germany, September 4-8, 2017*. ACM, 727–739.

[37] Fan Long and Martin Rinard. 2015. Staged Program Repair with Condition Synthesis. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering* (Bergamo, Italy) *(ESEC/FSE 2015)*. ACM, New York, NY, USA, 166–178. https://doi.org/10.1145/2786805.2786811

[38] Fan Long and Martin Rinard. 2016. Automatic Patch Generation by Learning Correct Code. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (St. Petersburg, FL, USA) *(POPL '16)*. ACM, New York, NY, USA, 298–312. https://doi.org/10.1145/2837614.2837617

[39] Fan Long, Stelios Sidiroglou-Douskos, and Martin Rinard. 2014. Automatic Runtime Error Repair and Containment via Recovery Shepherding. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Edinburgh, United Kingdom) *(PLDI '14)*. Association for Computing Machinery, New York, NY, USA, 227–238. https://doi.org/10.1145/2594291.2594337

[40] Thibaud Lutellier, Hung Viet Pham, Lawrence Pang, Yitong Li, Moshi Wei, and Lin Tan. 2020. CoCoNuT: Combining Context-Aware Neural Translation Models Using Ensemble for Program Repair. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Virtual Event, USA) *(ISSTA 2020)*. Association for Computing Machinery, New York, NY, USA, 101–114. https://doi.org/10.1145/3395363.3397369

[41] Fernanda Madeiral, Simon Urli, Marcelo Maia, and Martin Monperrus. 2019. BEARS: An Extensible Java Bug Benchmark for Automatic Program Repair Studies. In *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 468–478. https://doi.org/10.1109/SANER.2019.8667991

[42] Ravichandhran Madhavan and Raghavan Komondoor. 2011. 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* (Portland, Oregon, USA) *(OOPSLA '11)*. Association for Computing Machinery, New York, NY, USA, 1033–1052. https://doi.org/10.1145/2048066.2048144

[43] Sergey Mechtaev, Jooyong Yi, and Abhik Roychoudhury. 2015. DirectFix: Looking for Simple Program Repairs. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1* (Florence, Italy) *(ICSE '15)*. IEEE Press, Piscataway, NJ, USA, 448–458. http://dl.acm.org/citation.cfm?id=2818754.2818811

[44] Sergey Mechtaev, Jooyong Yi, and Abhik Roychoudhury. 2016. Angelix: Scalable Multiline Program Patch Synthesis via Symbolic Analysis. In *Proceedings of the 38th International Conference on Software Engineering* (Austin, Texas) *(ICSE '16)*. ACM, New York, NY, USA, 691–701. https://doi.org/10.1145/2884781.2884807

[45] Martin Monperrus. 2018. Automatic Software Repair: A Bibliography. *ACM Comput. Surv.* 51, 1, Article 17 (Jan. 2018), 24 pages. https://doi.org/10.1145/3105906

[46] Paul Muntean, Martin Monperrus, Hao Sun, Jens Grossklags, and Claudia Eckert. 2019. IntRepair: Informed Repairing of Integer Overflows. *IEEE Transactions on Software Engineering* (2019), 1–1. https://doi.org/10.1109/TSE.2019.2946148

[47] Mangala Gowri Nanda and Saurabh Sinha. 2009. Accurate Interprocedural Null-Dereference Analysis for Java. In *2009 IEEE 31st International Conference on Software Engineering*. 133–143. https://doi.org/10.1109/ICSE.2009.5070515

[48] Hoang Duong Thien Nguyen, Dawei Qi, Abhik Roychoudhury, and Satish Chandra. 2013. SemFix: Program Repair via Semantic Analysis. In *Proceedings of the 2013 International Conference on Software Engineering* (San Francisco, CA, USA) *(ICSE '13)*. IEEE Press, Piscataway, NJ, USA, 772–781. http://dl.acm.org/citation.cfm?id=2486788.2486890

[49] Renaud Pawlak, Martin Monperrus, Nicolas Petitprez, Carlos Noguera, and Lionel Seinturier. 2016. Spoon: A library for implementing analyses and transformations of java source code. *Software: Practice and Experience* 46, 9 (2016), 1155–1179.

[50] Ripon K. Saha, Yingjun Lyu, Hiroaki Yoshida, and Mukul R. Prasad. 2017. Elixir: Effective object-oriented program repair. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 648–659. https://doi.org/10.1109/ASE.2017.8115675

[51] Marija Selakovic and Michael Pradel. 2015. Poster: Automatically Fixing Real-World JavaScript Performance Bugs. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, Vol. 2. 811–812. https://doi.org/10.1109/ICSE.2015.260

[52] Alex Shaw, Dusten Doggett, and Munawar Hafiz. 2014. Automatically Fixing C Buffer Overflows Using Program Transformations. In *Proceedings of the 2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN '14)*. IEEE Computer Society, Washington, DC, USA, 124–135. https://doi.org/10.1109/DSN.2014.25

[53] Stelios Sidiroglou-Douskos, Eric Lahtinen, Fan Long, and Martin Rinard. 2015. Automatic Error Elimination by Horizontal Code Transfer Across Multiple Applications. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Portland, OR, USA) *(PLDI '15)*. ACM, New York, NY, USA, 43–54. https://doi.org/10.1145/2737924.2737988

[54] Edward K. Smith, Earl T. Barr, Claire Le Goues, and Yuriy Brun. 2015. Is the Cure Worse Than the Disease? Overfitting in Automated Program Repair. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering* (Bergamo, Italy) *(ESEC/FSE 2015)*. ACM, New York, NY, USA, 532–543. https://doi.org/10.1145/2786805.2786825

[55] Shin Hwei Tan, Zhen Dong, Xiang Gao, and Abhik Roychoudhury. 2018. Repairing Crashes in Android Apps. In *Proceedings of the 40th International Conference on Software Engineering* (Gothenburg, Sweden) *(ICSE '18)*. Association for Computing Machinery, New York, NY, USA, 187–198. https://doi.org/10.1145/3180155.3180243

[56] Shin Hwei Tan, Hiroaki Yoshida, Mukul R Prasad, and Abhik Roychoudhury. 2016. Anti-patterns in search-based program repair. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. 727–738.

[57] Yuchi Tian and Baishakhi Ray. 2017. Automatically Diagnosing and Repairing Error Handling Bugs in C. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering* (Paderborn, Germany) *(ESEC/FSE 2017)*. ACM, New York, NY, USA, 752–762. https://doi.org/10.1145/3106237.3106300

[58] Rijnard van Tonder and Claire Le Goues. 2018. Static Automated Program Repair for Heap Properties. In *Proceedings of the 40th International Conference on Software Engineering* (Gothenburg, Sweden) *(ICSE '18)*. ACM, New York, NY, USA, 151–162. https://doi.org/10.1145/3180155.3180250

[59] Westley Weimer, ThanhVu Nguyen, Claire Le Goues, and Stephanie Forrest. 2009. Automatically Finding Patches Using Genetic Programming. In *Proceedings of the 31st International Conference on Software Engineering (ICSE '09)*. IEEE Computer Society, Washington, DC, USA, 364–374. https://doi.org/10.1109/ICSE.2009.5070536

[60] Ming Wen, Junjie Chen, Rongxin Wu, Dan Hao, and Shing-Chi Cheung. 2018. Context-Aware Patch Generation for Better Automated Program Repair. In *Proceedings of the 40th International Conference on Software Engineering* (Gothenburg, Sweden) *(ICSE '18)*. Association for Computing Machinery, New York, NY, USA, 1–11. https://doi.org/10.1145/3180155.3180233

[61] Martin White, Michele Tufano, Matías Martínez, Martin Monperrus, and Denys Poshyvanyk. 2019. Sorting and Transforming Program Repair Ingredients via Deep Learning Code Similarities. In *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 479–490. https://doi.org/10.1109/SANER.2019.8668043

[62] Qi Xin and Steven P Reiss. 2017. Identifying test-suite-overfitted patches through test case generation. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 226–236.

[63] Qi Xin and Steven P. Reiss. 2017. Leveraging syntax-related code for automated program repair. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 660–670. https://doi.org/10.1109/ASE.2017.8115676

[64] Yingfei Xiong, Xinyuan Liu, Muhan Zeng, Lu Zhang, and Gang Huang. 2018. Identifying Patch Correctness in Test-Based Program Repair. In *Proceedings of the 40th International Conference on Software Engineering* (Gothenburg, Sweden) *(ICSE '18)*. Association for Computing Machinery, New York, NY, USA, 789–799. https://doi.org/10.1145/3180155.3180182

[65] Yingfei Xiong, Xinyuan Liu, Muhan Zeng, Lu Zhang, and Gang Huang. 2018. Identifying patch correctness in test-based program repair. In *Proceedings of the 40th international conference on software engineering*. 789–799.

[66] Yingfei Xiong, Jie Wang, Runfa Yan, Jiachen Zhang, Shi Han, Gang Huang, and Lu Zhang. 2017. Precise Condition Synthesis for Program Repair. In *Proceedings of the 39th International Conference on Software Engineering* (Buenos Aires, Argentina) *(ICSE '17)*. IEEE Press, Piscataway, NJ, USA, 416–426. https://doi.org/10.1109/ICSE.2017.45

[67] Xuezheng Xu, Yulei Sui, Hua Yan, and Jingling Xue. 2019. VFix: Value-flow-guided Precise Program Repair for Null Pointer Dereferences. In *Proceedings of the 41st International Conference on Software Engineering* (Montreal, Quebec, Canada) *(ICSE '19)*. IEEE Press, Piscataway, NJ, USA, 512–523. https://doi.org/10.1109/ICSE.2019.00063

[68] Jifeng Xuan, Matias Martinez, Favio DeMarco, Maxime Clément, Sebastian Lamelas Marcote, Thomas Durieux, Daniel Le Berre, and Martin Monperrus. 2017. Nopol: Automatic Repair of Conditional Statement Bugs in Java Programs. *IEEE Transactions on Software Engineering* 43, 1 (2017), 34–55. https://doi.org/10.1109/TSE.2016.2560811

[69] Jinqiu Yang, Alexey Zhikhartsev, Yuefei Liu, and Lin Tan. 2017. Better Test Cases for Better Automated Program Repair. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering* (Paderborn, Germany) *(ESEC/FSE 2017)*. ACM, New York, NY, USA, 831–841. https://doi.org/10.1145/3106237.3106274

[70] Jinqiu Yang, Alexey Zhikhartsev, Yuefei Liu, and Lin Tan. 2017. Better test cases for better automated program repair. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. 831–841.

[71] He Ye, Jian Gu, Matias Martinez, Thomas Durieux, and Martin Monperrus. 2021. Automated classification of overfitting patches with statically extracted code features. *IEEE Transactions on Software Engineering* (2021).

[72] Alex Zhitnitsky. [n.d.]. The Top 10 Exception Types in Production Java Applications – Based on 1B Events. https://www.overops.com/blog/the-top-10-exceptions-types-in-production-java-applications-based-on-1b-events/.