VFix: Value-Flow-Guided Precise Program Repair for Null Pointer Dereferences

Xuezheng Xu†, Yulei Sui‡, Hua Yan* and Jingling Xue*

*School of Computer Science and Engineering, UNSW Sydney, Australia
{xuezhengxu, huayan, jingling}@cse.unsw.edu.au

†Faculty of Engineering and Information Technology, University of Technology Sydney, Australia
yulei.sui@uts.edu.au

Abstract—Automated Program Repair (APR) faces a key challenge in efficiently generating correct patches from a potentially infinite solution space. Existing approaches, which attempt to reason about the entire solution space, can be ineffective (by often producing no plausible patches at all) and imprecise (by often producing plausible but incorrect patches).

We present VFix, a new value-flow-guided APR approach, to fix null pointer exception (NPE) bugs by considering a substantially reduced solution space in order to greatly increase the number of correct patches generated. By reasoning about the data and control dependences in the program, VFix can identify bug-relevant repair statements more accurately and generate more correct repairs than before. VFix outperforms a set of 8 state-of-the-art APR tools in fixing the NPE bugs in Defects4J in terms of both precision (by correctly fixing 3 times as many bugs as the most precise one and 50% more than all the bugs correctly fixed by these 8 tools altogether) and efficiency (by producing a correct patch in minutes instead of hours).

Keywords—program repair, static analysis, null dereference

I. INTRODUCTION

To reduce the maintenance cost of large-scale software, Automated Program Repair (APR) provides a promising solution for automatically generating software patches to fix software bugs. Existing APR approaches proceed in two phases: (1) fault localization, which identifies a set of suspicious, i.e., repair statements, \( L \), that may trigger a bug after exercising at least one failing test case in a test suite, and (2) patch generation, which generates a repair operation \( o \in O \) by applying a predefined repair template (e.g., insertion and deletion) on a suspicious statement \( \ell \in L \) and then validates the candidate patch by running the repaired program against the test suite. This generate-and-validate process repeats until a plausible patch, i.e., one that passes all the test cases in the test suite is found. A plausible patch is only correct iff it results in the correct outputs for all possible program inputs.

Challenges. APR faces a key challenge in efficiently generating correct patches from a potentially infinite solution space. Given a set of predefined repair templates, existing search-based approaches usually generate an unbounded number of repair operations \( O \) (e.g., via brute-force mutations [38, 54, 78]) even for a single suspicious statement \( \ell \in L \). The underlying search-space-explosion problem is further exacerbated when all repair templates are instantiated over all the suspicious statements in a program. The resulting solution space \( S = L \times O = C \cup I \cup X \) may contain possibly infinitely many patches, where the ratio of the correct patches \( \in C \) over the incorrect patches (including implausible ones \( \in I \) and plausible but incorrect ones \( \in X \)) is extremely small. This can seriously impede the efficiency and effectiveness of existing APR approaches in finding correct patches.

Prior Work. To address this problem, existing efforts typically accelerate the repair process by ranking patches based on their probabilities of being correct, i.e., exercising only the high-priority patches that are likely to be correct. Given a user-specified time budget, the solution space \( L \times O \) is thus reduced to a subset \( L' \times O' \), where \( L' \) is often selected through a fault localization tool via stress testing with a limited number of test cases [4, 68, 88] and \( O' \) is selected by adopting a variety of heuristics, such as syntactic [24, 84] or semantic code search [32], statistical analysis [31, 37, 81, 85], symbolic execution [49, 53] and machine learning [44, 58].

Limitations. Without considering the data and control dependences in a program, these patch-ranking approaches that operate on the entire solution space are still inadequate, as they can be ineffective (by often producing no plausible patches at all) and imprecise (by often producing plausible but incorrect patches). In addition, increasing the time budget used for fixing a bug may increase the chances for obtaining a correct patch, but without any correctness guarantee in the presence of a huge solution space [86]. As a result, the per-bug time budget

```java
1 boolean removeDomainMarker(…) {
2 …
3 if (layer == Layer.FOREGROUND) {
4 markers = this.Markers.get(…);
5 // if (markers==null) return false;
6 } else {
7 // if (markers==null) return false;
8 removed = markers.remove(…);
9 // NPE!
10 if (removed & notify)
11 return removed;
12}
```

Fig. 1: An example with three patches, correct, plausible but incorrect, and incorrect, for fixing the NPE at \( i_8 \).

78]
is often set up as 3 – 5 hours [38, 54, 78, 81, 87] for heavy testing.

Most APR approaches [24, 31, 52, 57, 81, 84, 85] adopt a generic process to repair bugs without distinguishing their types or categories. Thus, many repair operations that are unable to repair a particular type of bugs are often generated, but to no avail. For example, existing APR tools are ineffective in fixing Null Pointer Dereference (aka NullPointerException (NPE)), one of the most common types of Java bugs [77], representing 37.2% of all memory bugs in Mozilla and Apache and over 40% of exceptions in Android [33, 41, 72]. However, an APR approach is expected to increase both precision and efficiency by being aware of the bug types, e.g., NPE, repaired during its fault localization and patch generation.

NPE, as a representative bug type, is very difficult to fix since choosing a right repair statement with a correct repair operation is challenging. For the 15 NPEs in Defects4j (a well-known benchmark suite often used for validating APR [30]), existing tools, HDRepair [37], ACS [56], CapGen [81] and SimFix [28], can only correctly repair 1, 2, 2 and 4 NPEs, respectively, after having tried tens of thousands of incorrect patches in hours per bug. This is because an NPE and its bug-fixing location can span across multiple functions in large codebases [9], due to a wide variety of programming mistakes, including missing null pointer checks [34, 58] and object initialization [34], resulting in possibly infinite many repair operations at a large number of suspicious statements [43].

Insights. Static analysis is relatively unexplored for automated program repair. This paper aims to make one step forward in investigating how to apply static value-flow analysis, which resolves both the data and control flow of a program, to help APR generate a precise solution space by increasing the number of correct patches generated for repairing NPEs. Figure 1 gives a real NPE from JFreeChart to demonstrate how we can avoid a plausible but incorrect patch (due to an imprecise $L'$) and an implausible patch (due to an imprecise $O'$) if the data and control flow information is considered.

When obtaining $L'$, the spectrum-based fault localization tools [41, 83] usually record the execution traces for successful and failed test cases and then compare the two types of traces by ranking the frequently executed statements that trigger a bug [56]. Due to limited test cases, this coarse-grained selection usually produces an imprecise $L'$ since some $l \in L'$ may not represent a correct repair location [53]. In Figure 1, markers may be null in both branches, $\ell_4$ and $\ell_6$. However, there is only one test case that triggers the bug at $\ell_6$, with $L' = \{ \ell_2, \ell_3, \ell_4, \ell_8 \}$ containing the executed statements along the if branch but without the unexecuted ones, $\ell_5 - \ell_7$ (along the else branch) and $\ell_9 - \ell_{12}$. All the statements in $L'$ are given the same priority to produce a fix. However, applying a fix at any statement other than $\ell_6$ will always generate an incorrect patch, since the unexecuted $\ell_6$ may also introduce an NPE at $\ell_8$. For example, placing a fix after $\ell_4$ will generate a plausible but incorrect patch as highlighted in yellow.

When obtaining $O'$, existing approaches [24, 31, 52, 57, 81, 84, 85], which ignore data and control dependences, often generate incorrect repair operations that either fail to pass a test case or introduce new bugs. For example, a skip operation is often applied, resulting in a null pointer check added (highlighted in gray in Figure 1) to bypass the crash point at $\ell_9$. However, removed, which is only initialized at $\ell_8$, is used later at $\ell_9$. Thus, this $\ell_9$-bypassing skip operation will cause an undefined behavior at $\ell_9$, failing on some test cases. By considering the static value-flow information, we can generate a correct fix (same as the manual fix in green) that never accesses such an undefined variable, by also avoiding hundreds of attempts made by, e.g., dependence-unaware repair [48].

Our Solution. This paper introduces VFIX, a new value-flow-guided APR approach to fixing NPEs by considering a substantially reduced solution space in order to increase the number of correct patches generated. VFIX enhances APR by incorporating with data and control dependence information, making it possible to identify bug-relevant repair statements more accurately and generate more correct repairs than before.

VFIX is complementary to existing APR approaches, which operate on an already reduced solution space $S' = L' \times O' = C' \cup I' \cup X'$, VFIX will operate on a different one $S_{vf} = S_{if} \times S_{of} = C_{vf} \cup I_{vf} \cup X_{vf}$ to improve precision and efficiency by including more correct patches and less incorrect ones from the entire solution space $S = L \times O$, such that $|S_{vf}| \ll |S'|$, $|C_{vf}| \gg |C'|$, and $|I_{vf} \cup X_{vf}| \ll |I' \cup X'|$.

Figure 2 gives an overview of VFIX. A test suite for a given NPE bug contains one failing test case (with several NPE-triggering test cases for the same bug conceptually treated as one). For the NPE-triggering test case given, VFIX first constructs an inter-procedural value-flow graph (VFG), a static (value-flow) slice of the program, to capture all the potential NPE-triggering sources and other related NPE crash sites. Then we formulate our fault localization problem by first identifying the set of suspicious statements, $L_{vf}$, as a portion of the static slice dynamically executed by the NPE-triggering test case, and then rank them by solving a graph congestion calculation problem on the static slice [15, 16]. Given a repair location, VFIX finally produces a precise set of value-flow-aware repair operations, $O_{vf}$, by filtering out repair templates or their sub-templates doomed to yield incorrect repair operations based on the dependence constraints established.

In summary, we make the following main contributions:

- We propose VFIX, a value-flow-guided APR approach to
We describe VFIX’s three steps for constructing static value-flow slices (Section 3A), selecting and ranking repair loca-
s ∈ Stmt ::= p = q | p = q.f | p.f = q | p = new T | p = null | return_m, p | p = q.m(‘T’)

r, p, q ∈ Ptr f ∈ Field T ∈ Class a, a’ ∈ Obj m ∈ Method ℓ, ℓ’, ℓ” ∈ Label

(a) A tiny Java-like language

Rule | Statement ℓ | Def-Use Information
--- | --- | ---
NullAsgn | p = null | Def ℓ ∈ Def p
NewAsgn | p = new T | Def ℓ ∈ Def p
Copy | p = q | Def ℓ ∈ Def p
Load | p = q.f; a ∈ pts(q) | Def ℓ ∈ Use a.f
Store | p.f = q; a ∈ pts(p) | Def ℓ ∈ Use a.f
Call | p = q.m(‘T’) | Def ℓ ∈ Use a.f
MtdEntry | m(‘T’){} | Def ℓ ∈ Use a.f
Ret | return_m, p | Def ℓ ∈ Use a.f

(b) Intra-procedural def-use information statement-wise (Def p, Use a.f)

denotes the set of definition (use) statements for a variable v.

Fig. 4: Value-flow graph construction.

A. Constructing Static Value-Flow Slices

Given a buggy program with one NPE crash site, VFIX builds an inter-procedural value-flow graph (VFG) \( G_{sta} = (L_{sta}, E_{sta}) \), a directed graph that captures all the potential NPE-triggering sources and other related NPE crash sites, where \( L_{sta} \) is the set of nodes representing statements and \( E_{sta} \) is the set of edges representing their def-use relations.

Figure 4(a) gives a tiny Java-like language. Global variables and static methods are handled in the standard manner.

Figure 4(b) lists the intra-procedural def-use information of a variable or field extracted directly from a statement. For a NullAsgn, NewAsgn or Copy statement, its def-use information is directly read off. For a Load or Store, the def-use information for the objects indirectly accessed must also be considered. Here, \( pts(v) \) denotes the points-to set of v obtained from a demand-driven context-sensitive pointer analysis \([66]\). At a Store, a weak update is assumed \([69]\).

For a method call (Call, MtdEntry and Ret), the def-use information for variables is directly available.

Figure 4(c) gives the rules for building the value-flow edges between two statements. To keep track of the inter-procedural value-flow for variables and fields, respectively, \([\text{INTER-POINTER}] \) and \([\text{INTER-OBJECT}] \) find the inter-procedural definition sites \( ℓ \) of a variable or field from its use site \( ℓ’ \) via \( \text{IntraDU} \) (Definition 1). \([\text{INTER-POINTER}] \) and \([\text{INTER-OBJECT}] \) handle the inter-procedural value-flow for variables and fields, respectively. In \([\text{INTER-POINTER}] \), the inter-procedural def-use relations for variables are obtained directly via parameter/return passing. In \([\text{INTER-OBJECT}] \) (illustrated in Figure 5), the inter-procedural def-use relations for fields are obtained via \( \text{InterDU} \) (Definition 2).

Definition 1 (Intra-Procedural Def-Use). \( \text{IntraDU}(ℓ, ℓ’, v) \), where \( v \) is a variable or a field, represents a def-use relation for \( v \) from \( ℓ \) to \( ℓ’ \) on an intra-procedural control flow path \( P \) with no redefinition of \( x \) in between, i.e., \( \exists ℓ'' ∈ P : ℓ ≺ ℓ'' ≺ ℓ’ \).

Definition 2 (Inter-Procedural Def-Use). \( \text{InterDU}(ℓ, ℓ’, a.f) \) represents a def-use relation for \( a.f \) from \( ℓ \) to \( ℓ’ \) (with both in two distinct methods) on an inter-procedural control-flow path \( P \), which is discovered context-sensitively \([66]\), with no redefinition of \( a.f \) in between, i.e., \( \exists ℓ'' ∈ P : ℓ ≺ ℓ'' ≺ ℓ’ \).

Example 1. Figure 5 shows a value-flow for \( a.f \) across \( \text{bar}() \), \( \text{zot}() \) and \( \text{foo}() \) via the two callsites \( ℓ_3 \) and \( ℓ_5 \). Here, \( a.f \) is defined at \( ℓ_2 : p.f = .. \) in \( \text{bar}() \) and used at \( ℓ_5 : .. = p.f \) in \( \text{foo}() \), where \( p.f \) and \( q.f \) are aliases since \( pts(p) \cap pts(q) = \{a\} \), on the inter-procedural control-flow path \( ℓ_2, ℓ_3, ℓ_4, ℓ_5, ℓ_6, ℓ_7 \), with no redefinition of \( a.f \).

Let \( p.use() \) be an NPE crash site, where \( p \) is null. We obtain its static value-flow slice \( G_{sta} = (L_{sta}, E_{sta}) \) by solving \( G_{i+1} = f_{vfg}(G_i) \) iteratively, starting with \( G_0 = \{p\} \), until a fixed point is reached, where \( f_{vfg} \) consists of applying the rules in Figure 4(c) to add first all the (direct and indirect) defs of the variables or fields in \( G_i \) to \( G_{i+1} \) and then all the (direct and indirect) uses of the variables or fields in \( G_{i+1} \) to \( G_{i+2} \). It is understood that \( f_{vfg} \) includes the statement for a variable or field under consideration in \( G_i \).

B. Selecting and Ranking Repair Locations

Let \( L_{dyn} \) be the set of statements executed by the NPE-triggering test case. To fix this bug, it suffices to consider only
the repair statements in the dynamic slice $L_{vf} = L_{dyn} \cap L_{sta}$, i.e., the portion of the static slice $L_{sta}$ dynamically executed.

VFIX then ranks the repair statements in $L_{vf}$ by solving a congestion calculation (Definition 3) problem on $G_{sta}$. The intuition behind is that a repair location with a higher congestion value has a better chance to avoid also the other related potential NPE bugs that are not discovered by the NPE-triggering test case, thus reducing more effectively the number of plausible but incorrect patches generated.

**Definition 3.** Given $G_{sta}$ for a NPE bug, an $\ell_s - \ell_t$ VF-path, denoted VF-Path($\ell_s, \ell_t$), is a simple path (with no repeated cycles), such that $\ell_s$ is a source and $\ell_t$ is a sink in $G_{sta}$. For a fixed source-sink pair ($\ell_s, \ell_t$), VF-Path-Set($\ell_s, \ell_t$) is the set of all $\ell_s - \ell_t$ VF-paths. The congestion value of a node $\ell$ in VF-Path-Set($\ell_s, \ell_t$) is given by $N_{\ell, s, t}^{vf}$, where $N_{\ell, s, t}^{vf}$ is the number of $\ell_s - \ell_t$ VF-paths that passes through $\ell$. The congestion value of a node $\ell$ in $G_{sta}$ is given by $\sum_{\ell_s \in SRC} \sum_{\ell_t \in SNK} N_{\ell, s, t}^{vf}$. The congestion value for a repair statement in $\ell$ is given in Figure 6(a), we assume that $L_{vf} = \{a, b, c, d, e\}$. Figure 6(b) displays its VF-Paths. Figure 6(c) ranks $a, b, c, d$ and $e$ based on their congestion values.

Finally, Algorithm 1 is used to rank the candidate statements in $L_{vf}$. In actuality, it is only necessary to compute the congestion value for a repair statement in $L_{vf}$ in line 5.

**C. Applying Value-Aware Repair Operations**

We first discuss our NPE bug model and then introduce the repair templates used for fixing NPE bugs.

---

**Algorithm 1: Ranking Repair Locations**

**Input:** $G_{sta} = (L_{sta}, E_{sta})$ (a static slice)  
$L_{vf}$ (a set of repair statements)

**Output:** a linear ordering of the repair statements in $L_{vf}$

1. **Function** RankingRepairLocations($G_{sta}$, $L_{vf}$)
2. $G_{sta}' = (L_{sta}', E_{sta}') \leftarrow G_{sta}$ with all SCCs collapsed;
3. Let $SRC'_{sta}$ be the set of source nodes in $G_{sta}'$;
4. Let $SNK'_{sta}$ be the set of sink nodes in $G_{sta}'$;
5. **foreach** $\ell \in L_{sta}'$ do
6. **foreach** $\ell_s \times \ell_t \in SRC'_{sta} \times SNK'_{sta}$ do
7. $N_{\ell_s, \ell_t}^{vf} \leftarrow$ number of $\ell_s - \ell_t$ VF-paths that passes through $\ell$ in $G_{sta}'$;
8. VF-Path-Set($\ell_s, \ell_t$) $\leftarrow$ set of $\ell_s - \ell_t$ VF-paths in $G_{sta}'$;
9. $C_{\ell}^{vf, \ell_s} \leftarrow |VF-Path-Set(\ell_s, \ell_t)|$;
10. $C_{\ell} \leftarrow \sum_{\ell_s \in SRC} \sum_{\ell_t \in SNK} C_{\ell}^{vf, \ell_s}$;
11. **end**
12. **end**
13. Define a linear ordering of the statements in $\ell' \in L_{sta}'$, $\gg_C$, in increasing order of their congestion values, $C_{\ell}$, by breaking ties with a reverse topological order;
14. Replace each SCC $\{\ell_1, \ldots, \ell_n\} \in L_{sta}'$ in $\gg_C$ by $\ell_1, \ldots, \ell_n$, where $\ell_i \in L_{sta}'$, in any arbitrary order;
15. **return** $\gg_C$ restricted to the statements in $L_{vf}$.

---

1) **NPE Bug Model:** The majority of the NPE bugs that happen at a crash site, $v.use()$, in real-world programs arise in two scenarios: (1) $v$ is not initialized on some path leading to $v.use()$, and (2) a “null check” for $v.use()$ is missing (e.g., when retrieving an element that may not exist in a map as in Figure 3). These two sources of NPEs are also acknowledged elsewhere in developing repair templates [34, 86, 87].

In this paper, we focus on repairing these two kinds of NPE bugs by using call-graph integrity (i.e., that API calls are invoked correctly) and type integrity (i.e., that variables are typed correctly in their declarations). In Section [4], we discuss some NPE bugs caused due to such integrity violations.

2) **Repair Operations:** For the NPE bugs considered, we summarize all candidate repair operations based on two repair templates, i.e., initialization and skip, which are also used by some existing APR tools [34, 86, 87]. The Initialization template aims to initialize a null pointer by assigning a newly created object while the skip template aims to avoid executing an NPE-triggering statements and other related ones. In this paper, we focus on how to use these two templates efficiently and precisely under VFIX, our value-flow analysis framework.

VFIX, as discussed in Section 5, operates on the solution space $L_{vf} \times O_{vf}$, where $L_{vf}$ is the set of repair statements identified in Section III-B and $O_{vf}$ is the set of repair operations defined in Figure 4. There are two sets of rules. The set of rules in Figure 4(a) is responsible for extracting the candidate variables or fields from a repair statement in $\ell \in L_{vf}$ that may participate in a repair operation for $\ell$. The set of
rules in Figure 7(b) generates the repair operations in $O_d$ by using six NPE-repairing templates, with three refined from the initialization template and the three from the skip template.

Let us first consider the rules in Figure 7(a). As $L_{vf}$ is a portion of $G_{src}$ restricted to those dynamically executed by the NPE-triggering test case, denoted $G_{vf}$, $SRC_{vf}$, $SNK_{vf}$ and $INT_{vf}$ represent the set of source, sink and intermediate nodes in $G_{vf}$, respectively. In particular, $SNK_{vf}$ is a singleton containing the NPE-triggering statement. For each repair statement $\ell \in L_{vf}$, we collect $(\ell, x, y) \in VF$, where $x$ and $y$ appear in $\ell$, to indicate that $y$ may be null on entry of $\ell$ and $x$ may be null (or undefined when $\ell \in SNK_{vf}$) on exit of $\ell$. $[V-ASSN-NULL], [V-ASSN], [V-RET-NULL], [V-RET]$ are self-explanatory, where $\ell \in SRC_{vf}$ and $\ell \notin SNK_{vf}$ are redundant. For a load ($\ell : p = q.f$), store ($\ell : p.f = q$), or call ($\ell : p = q.m(T)$), there are two rules each, depending on whether $\ell$ is the actual NPE crash site or not.

Let us now examine the six NPE-fixing templates given in Figure 7(b) in greater detail. Due to the value-flow information collected (Figure 7(b)), we can now apply a template when some value-flow constraints are satisfied, thereby filtering out a particular incorrect template, and consequently, all the incorrect repair operations instantiated from it.

(a) Initialization: There are three templates, [INIT-SIMP], [INIT-SRC] and [INIT-SNK], which serve to assign a new object to a potentially null pointer (a variable or field). Currently, when generating an object initialization statement $v = new T$, the public default constructor (if available) in class $T$, where $T$ is the declared type of $v$ or a subtype, is used as in $[\text{INIT-SIMP}]$. In future work, other non-default constructors can be considered similarly. According to the premises of these templates, [INIT-SIMP] applies to the statements handled by $[V-ASSN], [V-RET], [V-LD-SNK], [V-ST-SNK]$ and $[V-CALL-SNK]$; [INIT-SRC] applies to the statements handled by $[V-ASSN-NULL]$ and $[V-RET-NULL]$; and [INIT-SNK] applies to the statements handled by $[V-LD-SNK]$ and $[V-CALL-SNK]$.

In [INIT-SIMP], as $y$ may be null, we perform an initialization of $y$ guarded by a runtime null check for $y$.

In [INIT-SRC], where a repair statement is $\_ = null$ or return null, we simply add the missing initialization.

In [INIT-SNK], where a repair statement $\ell$ (a load or a call) suffers from a null dereference when $y = null$, causing the result in $x$ to be undefined, we add an initialization of $x$ guarded again by a runtime null check for $y$ and also skip $\ell$. However, we will only do so when $\exists \ell' \in USe_y : \ell' pDom \ell$ holds, i.e., when $y$ does not have any post-dominant use in $USe_y$ with respect to $\ell$. Otherwise, the repair operation for $\ell$ will likely be incorrect even plausible. If $x$ is of a primitive type $T$, then $T()$ returns the default value for $T$ as in C++.

(b) Skip: There are also three templates, [SKIP-ONE], [SKIP-MULTI] and [SKIP-RET], which each insert a null check at a repair location to skip the statements affected by the null dereference. These three rules apply to all the statements except for $p = null$ and return null, which are handled by [INIT-SRC]. In both [SKIP-ONE] and [SKIP-MULTI], $\text{FSL}(\ell)$ represents the traditional forward slice computed at $\ell$.

In [SKIP-ONE], as $y$ may be null at a repair statement $\ell$, we skip $\ell$ with a null check $y != null$ (when $\text{FSL}(\ell) = \{\ell\})$.

In [SKIP-MULTI], we skip multiple statements that depend on a repair statement $\ell$ (when $\text{FSL}(\ell) = \{\ell, \ell_1, \ldots, \ell_n\} \neq \{\ell\}$). If $\ell, \ell_1, \ldots, \ell_n$ are not consecutive, we choose to skip the smallest code region encompassing these statements.

In [SKIP-RET], we skip all the statements starting from $\ell$ in the method $m$ containing $\ell$, by returning nothing $v$ if $m$'s return type is $\textbf{void}$, or new $T$ if $T$ is primitive as discussed for [INIT-SNK], or one in $\{\text{new } T, \text{null}\}$ if $T$ is a class.

In the case of multiple repair operations available at a repair statement from both an initialization template and a skip template, the skip template will be tried first, reflecting how such templates are used in real-world bug-fixing scenarios.
Example 3. Let us revisit Figure 1 to select the repair operations for the statement at $\ell_{8}$, the NPE-triggering site. By \(\text{[V-CALL-SNK]}\), we obtain \(\{\ell_{8}, \text{removed, markers}\} \in \text{VF}\). The skip templates are considered first. As \(\text{FSL}(\ell_{8}) = \{\ell_{8}, \ell_{9}, \ell_{10}, \ell_{11}\}\) contains \(\ell_{11}\), a \text{return} statement, \(\text{[SKIP-RET]}\) applies, giving rise to the correct path shown in Figure 1. Without our value-flow analysis, \(\text{[SKIP-ONE]}\) may be tried, resulting in the implausible patch also shown in Figure 1. Thus, VFIX can successfully avoid many incorrect repair operations, thereby reducing the search space.

IV. Evaluation

Our objective is to show that VFIX can significantly outperform the state of the art for repairing NPE bugs in terms of both precision and efficiency (i.e., the time spent on generating a correct patch). By comparing VFIX with a set of eight representative APR tools in fixing all the 15 NPE bugs in Defects4j (version 1.0.1), we find that VFIX can correctly repair 3x as many NPEs as the most precise one, SimFix [28] (12 bugs by VFIX vs. 4 bugs by SimFix) and 50% more than all the NPEs correctly fixed by all the eight APR tools together. In addition, we have also validated the effectiveness of VFIX using another set of 15 NPEs in 4 open-source Apache projects. VFIX has successfully repaired 12 NPEs, by generating the fixes that are semantically equivalent to the developers’ fixes. VFIX repairs 24 out of these 30 NPEs in about 30 minutes, i.e., under 80 seconds per bug on average.

A. Implementation

We have implemented VFIX in SOOT [26] in about 11.5 KLOC of Java code, with our value-flow analysis performed in its Jimple IR. Given a program with a test suite consisting of one NPE-triggering test case, VFIX relies on SOOT’s built-in CFGs and call graph to construct its static value-flow slice \(L_{\text{sta}}\) by using the rules in Figures 4(b) and (c). The points-to information required is discovered by using a demand-driven context-sensitive pointer analysis [66]. GZoltar [12] is used to obtain an execution trace \(L_{\text{dyn}}\) for the program under the NPE-triggering test case. \(L_{\text{cf}} = L_{\text{sta}} \cap L_{\text{dyn}}\) then contains all the repair statements localized. To generate patches, we use JavaParser [8] to parse, analyze and transform the source code.

B. Experimental Setup

We use two sets of benchmarks. Table I lists the four out of the five projects in Defects4j (version 1.0.1) [30], a bug database widely used by the program repair community. Note that Closure Compiler is excluded due to the lack of a standard JUnit testing format. There are 15 NPE bugs, with 7 in Chart, 5 in Lang, 3 in Math and 0 in Time. We have selected this version since the experimental results from many existing tools for it are available. Table I lists the four large open-source Apache applications with also a total of 15 NPEs, Pdfbox (9), Felix (2), Collections (2) and Sling (2). We have selected these 15 NPEs from their bug repositories [2] since they have NPE-triggering test cases with the correct patches accepted by developers. For each of these 30 NPEs, we use the test suite provided for the class where the bug resides, which includes one NPE-triggering test case.

Our experiments were done on a machine with an Intel Core i5 3.20 GHz CPU and 4GB memory, running Ubuntu 16.04 operating system with JDK 1.6.0_45 with the maximum heap size of Java virtual machine set as 4 GB. Each program was run five times and the average is reported in our evaluation.

C. Precision and Efficiency

Following [81] [86], we adopt a relatively strict definition of correctness for a patch. A patch is correct iff it passes all the test cases in the test suite and is also semantically or syntactically equivalent to a human-written patch.

We compare VFIX with eight representative APR tools, seven general-purpose ones, jGenProg [38, 48], jKALI [48, 55], NOPOL [48], ACS [86], CAPGEN [81], HDRepair [37] and SimFix [28], and one specialized for NPEs, NPEFix [22]. The results of the first seven tools are taken from their papers. The time budget per bug is 90 minutes for CAPGEN and HDRepair, 180 minutes for jGenProg, jKALI and NOPOL, 300 minutes for SimFix, and 30 minutes for ACS.

Figure 8 compares VFIX with these APR tools in terms of each tool’s capability in generating correct patches for the 15 NPE bugs in Defects4j. VFIX outperforms these tools by correctly repairing 12 out of the 15 NPE bugs.

1) VFIX vs. General-Purpose APR Tools: Among the seven general-purpose APR tools, SimFix is the best in terms of the number of correct patches generated, repairing 4 out of the 15 NPEs, with one plausible but incorrect patch and 10 implausible patches produced. CAPGEN and ACS each fix two NPEs. NOPOL and jKALI produce some plausible patches, which are all incorrect. VFIX fails to generate correct patches for three NPEs, as discussed further in Section IV-F. In comparison with these general-purpose APR tools, VFIX has successfully repaired six out of the 15 NPE bugs exclusively.

2) VFIX vs. NPEFix: NPEFix, an APR tool developed to repair NPE bugs, can generate 12 plausible patches but with only two being correct. When fixing a NPE bug, NPEFix considers only the NPE-triggering test case while ignoring the
others in the test suite. This renders NPEFix vulnerable to over-fitting with simple plausible yet incorrect patches such as randomly replacing a null pointer with a non-null pointer. Thus, the majority of its patches (10/12 = 83%) are incorrect, confirmed by manual inspection, because they either introduce new bugs or significantly alter the program logic.

3) VFix vs. Runtime Recovery Tools: Instead of providing correct patches for bugs (including NPEs), some tools, such as APPEND [20], RCV [45], and Ares [27], aim to prevent crashes in order to continue program execution. These tools seek a temporary recovery solution in a short time rather than repeatedly generating and validating patches against a test suite. To compare with APPEND [20], we have written the patches for the 15 NPEs in Defects4j by following its recovery policy (i.e., calling a default constructor or skipping the null dereference if the default constructor is unavailable). Only two bugs, Lang-20 and Lang-47, are fixed correctly.

4) VFix’s Patch Generation: Table III analyzes VFix’s efficiency and precision in repairing the 30 NPEs across the 8 projects, with a 10-minute time budget per bug. VFix repairs a bug in three steps, Steps 1 – 3, as shown in Figure 2. For Chart-2, Math-79, Pdfbox-2812 and Math-2951, VFix times out in Step 3 (marked as OOB). If these four NPEs are included, VFix takes 4256 seconds (70.9 minutes) in repairing the 30 NPEs, with 22.9%, 7.4% and 69.7% in Steps 1 – 3, respectively. Otherwise, VFix takes 1856 seconds (30.9 minutes) in repairing the 26 remaining NPEs, with 44.4%, 13.4% and 42.2% in Steps 1 – 3, respectively.

For the 30 NPE bugs, VFix has successfully generated patches for 26 NPEs, in which 24 are correct and two are incorrect, giving a precision of 92.3%. The high precision confirms the effectiveness of our value-flow analysis. In addition, VFix is also efficient as the first patch is correct in 19 out of 24 correct fixes. This fast and precise patch generation is attributed to the value-flow-aware repair operations used. For the four NPE bugs mentioned above, however, VFix still fails to generate plausible patches within the time budget.

**TABLE II:** Analyzing VFix’s patch generation for repairing the 30 NPEs across the 8 projects (with a 10-minute time budget per bug). OOB denotes out-of-budget. A breakdown of the times for its three steps (Figure 2) is given. A green (red) box marks a correct (incorrect) patch, while a white box marks a bug for which no plausible patch is generated.

<table>
<thead>
<tr>
<th>Bug ID</th>
<th>Time (secs)</th>
<th>N-th patch</th>
<th>Correctness</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Step 1</td>
<td>Step 2</td>
<td>Step 3</td>
</tr>
<tr>
<td>Chart-2</td>
<td>37</td>
<td>16</td>
<td>OOB</td>
</tr>
<tr>
<td>Chart-4</td>
<td>45</td>
<td>14</td>
<td>34</td>
</tr>
<tr>
<td>Chart-14</td>
<td>41</td>
<td>14</td>
<td>32</td>
</tr>
<tr>
<td>Chart-15</td>
<td>48</td>
<td>13</td>
<td>46</td>
</tr>
<tr>
<td>Chart-16</td>
<td>13</td>
<td>8</td>
<td>10</td>
</tr>
<tr>
<td>Chart-25</td>
<td>41</td>
<td>13</td>
<td>37</td>
</tr>
<tr>
<td>Chart-26</td>
<td>43</td>
<td>3</td>
<td>49</td>
</tr>
<tr>
<td>Lang-20</td>
<td>14</td>
<td>3</td>
<td>13</td>
</tr>
<tr>
<td>Lang-33</td>
<td>15</td>
<td>4</td>
<td>17</td>
</tr>
<tr>
<td>Lang-39</td>
<td>14</td>
<td>3</td>
<td>16</td>
</tr>
<tr>
<td>Lang-47</td>
<td>13</td>
<td>3</td>
<td>16</td>
</tr>
<tr>
<td>Lang-57</td>
<td>12</td>
<td>3</td>
<td>12</td>
</tr>
<tr>
<td>Math-4</td>
<td>16</td>
<td>11</td>
<td>7</td>
</tr>
<tr>
<td>Math-70</td>
<td>13</td>
<td>8</td>
<td>9</td>
</tr>
<tr>
<td>Math-79</td>
<td>12</td>
<td>6</td>
<td>OOB</td>
</tr>
<tr>
<td>Collections-39</td>
<td>13</td>
<td>4</td>
<td>11</td>
</tr>
<tr>
<td>Collections-360</td>
<td>12</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>Pdfbox-2266</td>
<td>50</td>
<td>19</td>
<td>57</td>
</tr>
<tr>
<td>Pdfbox-2477</td>
<td>52</td>
<td>14</td>
<td>66</td>
</tr>
<tr>
<td>Pdfbox-2812</td>
<td>49</td>
<td>24</td>
<td>OOB</td>
</tr>
<tr>
<td>Pdfbox-2948</td>
<td>52</td>
<td>17</td>
<td>61</td>
</tr>
<tr>
<td>Pdfbox-2951</td>
<td>54</td>
<td>20</td>
<td>OOB</td>
</tr>
<tr>
<td>Pdfbox-2965</td>
<td>51</td>
<td>17</td>
<td>47</td>
</tr>
<tr>
<td>Pdfbox-2995</td>
<td>50</td>
<td>18</td>
<td>44</td>
</tr>
<tr>
<td>Pdfbox-3479</td>
<td>55</td>
<td>19</td>
<td>50</td>
</tr>
<tr>
<td>Pdfbox-3572</td>
<td>55</td>
<td>16</td>
<td>38</td>
</tr>
<tr>
<td>Sling-4992</td>
<td>12</td>
<td>1</td>
<td>16</td>
</tr>
<tr>
<td>Sling-6487</td>
<td>57</td>
<td>1</td>
<td>60</td>
</tr>
<tr>
<td>Total (with OOB)</td>
<td>976</td>
<td>314</td>
<td>2966</td>
</tr>
<tr>
<td>Total (w/o OOB)</td>
<td>824</td>
<td>248</td>
<td>784</td>
</tr>
</tbody>
</table>

**Fig. 9:** The percentages of the number of repair statements in $L_{\text{ref}}$ found by VFix over the total number of suspicious statements reported by the fault localization tool, GZoltar [12], for the 12 NPEs in Defects4j fixed by VFix (with Chart, Lang and Math abbreviated to C, L, and M, respectively).

**D. VFix’s Refined Solution Space**

We provide some insights on why and how VFix achieves high precision and efficiency by refining its solution space $L_{\text{ref}} \times C_{\text{ref}}$ using value-flow analysis. We consider the 12 NPEs in Defects4j that are fixed by VFix, as shown in Table III. Figure 9 shows that VFix has achieved an average reduction of 94.11% by moving away from the space of suspicious statements selected by a general-purpose fault localization tool, GZoltar [12], to $L_{\text{ref}}$. By leveraging the value-flow information for an NPE-triggering site, VFix avoids many irrelevant repair...
locations that would otherwise be selected by such bug-type-unaware spectrum-based fault localization techniques.

Figure 10 shows how VFIX has significantly reduced the number of repair operations in O_{df} by comparing the number of repair templates instantiated with and without our value-flow analysis at a statement where a correct fix is made. In the absence of value-flow information, all six templates (given in Figure 7) are used. By exploiting the value-flow information, VFIX has cut this down to 2.58, resulting in a reduction of 57.0%, on average, avoiding unnecessary repair operations tried and boosting the efficiency of patch generation.

E. Case Studies

We conduct two cases studies in Defects4j to show how VFIX repairs intra- and inter-procedural NPEs precisely and efficiently within a reduced solution space S_{df} = L_{df} \times O_{df}.

Figure 11(a) shows an NPE bug, Chart-4, and its patch generated by VFIX. The bug happens in \ell_{4494}, where x can be null. VFIX adds a null check in \ell_{4493} and \ell_{4452} to encompass \ell_{4494} - \ell_{4451} in an if branch by [SKIP-MULTI]. By manual inspection, we found that this patch is identical to the one committed by developers. The challenge here lies in determining the end of the scope for the if-branch. Closing it too early (e.g., right after \ell_{4494}) would leave some variables that are data-dependent on the skipped statements to be undefined (e.g., c in \ell_{4495} and a in \ell_{4496}). Closing it too late would introduce unnecessary control-dependencies on the if-statement added (in \ell_{4493}), potentially altering the program logic. Without value-flow analysis, one would have to blindly enumerate all possible mutations at a statement in order to find a correct fix, which is computationally impracticable. With value-flow analysis, resulting in a precise O_{df}, VFIX can correctly identify the scope of the inserted if branch by [SKIP-MULTI].

Figure 11(b) illustrates another NPE bug, Chart-16, repaired by VFIX. This bug happens in \ell_{574} as this.category is not initialized in \ell_{207} in the constructor DefaultIntervalCategoryDataset(...). Given \ell_{574}, G_{df} built by VFIX contains not only \ell_{574} but also \ell_{690}, another potential NPE crash site. For the given NPE-triggering test case, L_{df} contains \ell_{208} and \ell_{574} (but not \ell_{690} as it is not executed). VFIX finds the correct patch, which initializes this.category in \ell_{208} by [INIT-SRC]. This patch can be obtained only if the inter-procedural value-flow information is available. In addition, our ranking algorithm, which prioritizes the repair statements L_{df} precisely by their congestion values, is also instrumental here. As G_{df} contains both \ell_{574} and \ell_{690}, \ell_{208} is ranked ahead of \ell_{574}, enabling VFIX to generate the correct fix in \ell_{208}. Otherwise, adding the same fix just before \ell_{574} is only plausible but incorrect, as it fails to fix the other NPE crash site in \ell_{690}.

F. Discussions

In this paper, we investigate how to apply value-flow analysis to boost the precision and efficiency of APR. While we focus on fixing NPEs, the most common type of Java bugs, our approach can also be effective in fixing other types of bugs such as use-after-free and memory leaks in C/C++ programs.

In practice, a test suite provided for a bug report may not be comprehensive enough to enable the bug to be fixed. This paper shows that we can mitigate such deficiency by augmenting a test suite with static value-flow analysis.

Currently, VFIX focuses on repairing a class of commonly occurring NPE bugs under the assumptions that call-graph integrity and type integrity are preserved (Section III-C). However, an NPE can occur when a wrong API is called, resulting in a call-graph integrity violation. For example, fixing Math-70 in Defects4j would require return solve(min, max) to be replaced by return solve(min, max), so that ε becomes correctly initialized. In addition, an NPE can also occur due to an integer overflow, caused possibly by a type integrity violation. For example, fixing the NPE bug, Math-79 in Defects4j, would require the declared type of sum and dp in method distance() of class MathUtils to be changed from int to double.
Going beyond our current bug model will be an interesting research topic.

V. RELATED WORK

A. Automated Program Repair

Existing APR approaches can be broadly classified into two categories: general-purpose approaches, which can be theoretically applied to all kinds of bugs, and bug-specific approaches, which are designed for specific types of bugs. General-purpose approaches are based on search algorithms or driven by semantics-preserving transformations. Search-based approaches typically adopt a generate-and-validate process that generates candidate patches by exhaustively exploring the solution space, for example, genetic programming [38, 79] or random search [54], and then validates the patches with a test suite. In practice, the large search space hinders their efficiency and scalability. Therefore, much research has been devoted to developing effective heuristics and repair templates for narrowing down the scope and guiding the search to generate correct patches efficiently. For example, AE [78] reduces the search space by merging semantically equivalent patches. PROPHET [44] uses machine learning techniques to guide patch generation by learning from correct patches. CAPGen [81] leverages the context information extracted from a program’s AST to achieve fine-grained patch prioritization. PAR [34] summarizes common fix patterns from human-written patches and performs the generate-and-validate process within a domain confined by these patterns. ACS [86] focuses on synthesizing branch conditions, ranking patches by analyzing documents, predicate statistics and dependencies relations between variables. Semantics-driven approaches [35, 36, 49, 50, 53, 87] represent another class of program repair techniques, which view a repair task as a program synthesis problem and synthesize patches via constraint-solving. As evaluated in Section IV, existing general-purpose approaches are only marginally effective for repairing NPEs.

Bug-specific APR techniques restrict their scope to some types of bugs only. For example, FOOTPATCH [72] fixes bugs related to heap memory properties by employing separation logic to reason about pointer semantics. MEMFix [39] considers memory deallocation errors (e.g., memory leaks, double-free and use-after-free bugs) using static analysis. There are also others focusing on buffer overflows [60, 92], integer overflows [13, 18], memory leaks [23, 89], error-handling bugs [75] and concurrency errors [29, 29]. The work that is the most closely related to ours is NPEFix [22], which generates patches for NPEs by capturing runtime information with dynamic analysis. In contrast, VFIX represents a static approach for fixing NPEs by performing a systematic value-flow analysis to drastically reduce the search space in order to avoid implausible and plausible but incorrect patches. As evaluated in Section IV, VFIX outperforms NPEFix significantly in terms of both effectiveness and precision.

B. Value-Flow Analysis

Understanding the flow of values in a program is fundamental in program analysis [61, 70]. By explicitly modelling the definition-use relations among program variables, value-flow analysis enables or enhances a series of crucial tasks, including compiler optimization [7, 67], pointer analysis [40, 51, 62, 69, 70, 73, 74], bug detection [14, 71], software debugging [80, 82], and validation and verification [19, 21]. In recent years, for example, the potential for value-flow analysis has been widely explored in detecting a variety of critical bugs, including memory leaks [14, 71], uses of uninitialized variables [46, 91], use-after-free errors [61, 90], and information leaks [5, 26]. While many existing approaches track the flow of values iteratively at each program point along the control-flow [6, 57, 63, 64], VFIX uses a fully-sparse value-flow analysis for both variables and fields. This full-sparse leads to the efficiency of VFIX. To the best of our knowledge, VFIX is the first approach that exploits value-flow analysis to repair NPE bugs.

C. Mitigations Against Null Pointer Dereferences

The NPEs that cause program crashes can be detected by a variety of testing techniques such as fuzzing [11, 8, 10] and symbolic execution [11, 25, 59]. To increase coverage, static analysis has been investigated. XYLEM [52], for example, is a representative static detector that has been shown to be effective in the industry. There also exist research efforts focusing on verifying the absence of NPEs using static analysis, in both whole-program [42, 65] and demand-driven [47] settings. In addition, fault tolerance techniques for NPEs has also been studied [20, 27, 45]. RCV [45], for example, processes the interrupt signals triggered by NPEs at runtime with its own handlers to allow a buggy program to continue execution. VFIX, as an APR approach, can also benefit from a more precise static analysis for its fault localization.

VI. CONCLUSION

This paper presents VFIX, a new value-flow-guided APR approach for fixing NPE bugs by considering a substantially reduced solution space in order to increase the number of correct patches generated efficiently. We have formulated our fault localization problem as one of solving a congestion calculation problem based on static value-flow analysis and dynamic execution trace with respect to a given NPE-triggering test case. We have formulated our problem of generating repair operations as one of instantiating repair templates subject to certain value-flow constraints. VFIX is shown to generate more correct patches more efficiently than the state of the art.

In future work, we plan to extend VFIX by repairing the types of NPE bugs that are not currently covered by our bug model. We also plan to generalize our value-flow analysis approach to repair other non-NPE bugs.

VII. ACKNOWLEDGMENTS

We would like to thank the anonymous reviewers for their valuable comments. This research is supported by Australian Research Grants DP180104169 and DE170101081.


