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).**

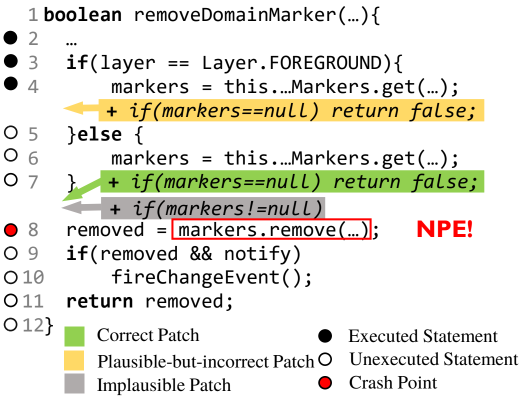
We present VFIX, a new value-flow-guided APR approach, to fix null pointer exception (NPE) bugs by considering a substantially reduced solution space in order to greatly increase the number of correct patches generated. By reasoning about the data and control dependences in the program, VFIX can identify 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.



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

Fig. 1: An example with three patches, correct, plausible but incorrect, and incorrect, for fixing the NPE at l_8.

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

1558-1225/19/$31.00 ©2019 IEEE
DOI 10.1109/ICSE.2019.00063

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

Most APR approaches [24, 31, 32, 37, 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 [17], 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 [86], 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 [4, 83] usually record the execution traces for successful and failed test cases and then contrast 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 [55]. In Figure 1, `markers` may be null in both branches, ℓ_4 and ℓ_6. However, there is only one test case that triggers the bug at ℓ_8, 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 ℓ_8 will always generate an incorrect patch, since the unexecuted ℓ_6 may also introduce an NPE at ℓ_8. For example, placing a fix after ℓ_4 will generate a plausible but incorrect patch as highlighted in yellow.

When obtaining O', existing approaches [24, 31, 32, 37, 58, 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 ℓ_8. However, `removed`, which is only initialized at ℓ_8, is used later at ℓ_9. Thus, this ℓ_8-bypassing skip operation will cause an undefined behavior at ℓ_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].

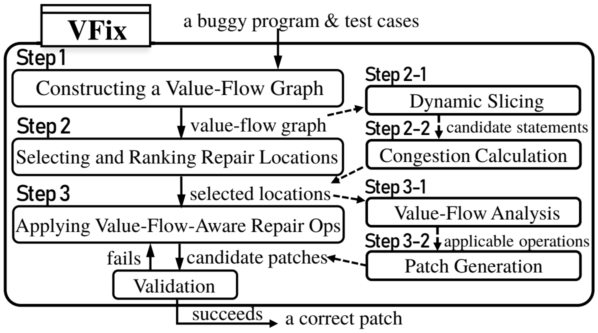


Fig. 2: The workflow of VFIX.

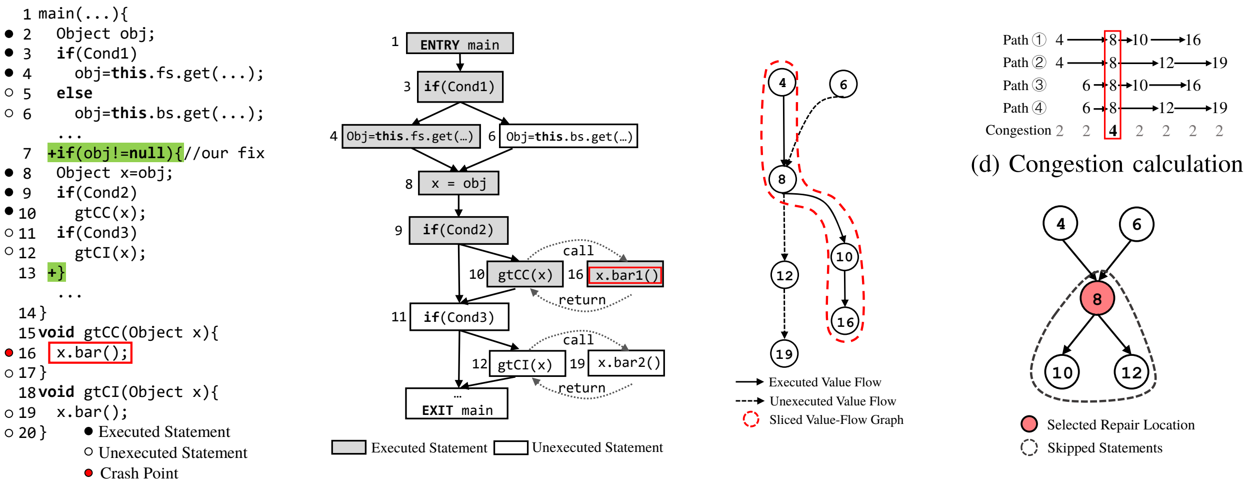
Our Solution. This paper introduces VFIX, a new value-flow-guided APR approach to fixing NPEs by considering a substantially reduced solution space in order to increase the number of correct patches generated. VFIX enhances APR by incorporating with data and control dependence information, making it 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 $S_{\text{vf}} = L_{\text{vf}} \times O_{\text{vf}} = C_{\text{vf}} \cup I_{\text{vf}} \cup X_{\text{vf}}$ 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 $|S_{\text{vf}}| \ll |S'|$, $|C_{\text{vf}}| \gg |C'|$, and $|I_{\text{vf}} \cup X_{\text{vf}}| \ll |I' \cup X'|$.

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



```
  1 main(...){
● 2   Object obj;
● 3   if(Cond1)
● 4     obj=this.fs.get(...);
○ 5   else
○ 6     obj=this.bs.get(...);
      ...
  7  +if(obj!=null){//our fix
● 8   Object x=obj;
● 9   if(Cond2)
●10     gtCC(x);
○11   if(Cond3)
○12     gtCI(x);
 13  +}
      ...
 14}
 15void gtCC(Object x){
●16   x.bar();
○17}
 18void gtCI(Object x){
○19   x.bar();
○20}
```

(a) A buggy program (b) Inter-procedural CFG (c) Value-flow graph (d) Congestion calculation (e) Repair operation

Fig. 3: A motivating example.

precisely and efficiently fixing NPE bugs by considering a substantially reduced solution space in order to increase the number of correct patches made.

- We formulate our fault localization problem by first identifying the suspicious problems based on static value-flow analysis and dynamic execution and then rank them by solving a graph congestion calculation problem.
- We have evaluated VFIX against a set of eight existing APR tools in terms of their ability in fixing the NPE bugs in `Defects4j`. VFIX is more effective by correctly fixing 3x as many bugs as the most precise one, SIMFIX [28] (12 bugs by VFIX vs. 4 bugs by SIMFIX), and 50% more than all the bugs correctly fixed by these eight tools altogether. In addition, VFIX is also more efficient by producing a correct patch in minutes instead of hours, on average. For 10 out of the 12 correctly fixed bugs, VFIX fixes each bug by generating only one patch in a single-pass validation. To further validate the effectiveness of VFIX, we also evaluate VFIX using another set of 15 NPEs in four open-source Apache projects. VFIX has successfully repaired 12 NPEs, with the fixes that are semantically equivalent to the developers' fixes.

II. A MOTIVATING EXAMPLE

We use a program in Figure 3 to explain the three steps of VFIX (Figure 2). In Figure 3(a), there is an NPE at ℓ_{16}, where x is `null`, adapted from the real NPEs in `JFreeChart`. The inter-procedural CFG in Figure 3(b) highlights the set of executed statements (in gray), $L_{\text{dyn}} = \{\ell_2, \ell_3, \ell_4, \ell_8, \ell_9, \ell_{10}, \ell_{16}\}$, by a test case that triggers the NPE, as `obj = null` after ℓ_4 on retrieving a non-existent element from a map. VFIX is able to generate exactly the same bug-fixing patch as the developer's fix (highlighted in ℓ_7 and ℓ_{13}), as follows:

Step 1: Constructing a Value-Flow Graph. For the crash site at ℓ_{16}, VFIX builds an inter-procedural value-flow graph (VFG), which is essentially a static (value-flow) slice, $G_{\text{sta}} = (L_{\text{sta}}, E_{\text{sta}})$, shown in Figure 3(c), where $L_{\text{sta}} = \{\ell_4, \ell_6, \ell_8, \ell_{10}, \ell_{12}, \ell_{16}, \ell_{19}\}$. The nodes (identified by line numbers) in L_{sta} represent variables and their edges capture their def-use relations. Apart from ℓ_4, the source for triggering the NPE at ℓ_{16}, L_{sta} also includes another potential source ℓ_6 for this NPE and a second potential crash point ℓ_{19}.

Step 2: Selecting and Ranking Repair Locations. This amounts to solving a fault localization problem. Given L_{dyn} and L_{sta}, VFIX collects the suspicious statements on the dynamic slice $L_{\text{vf}} = L_{\text{dyn}} \cap L_{\text{sta}} = \{\ell_4, \ell_8, \ell_{10}, \ell_{16}\}$, which is a portion of the static slice executed by the bug-triggering test case. VFIX then ranks these statements by solving a value-flow congestion calculation problem on G_{sta}. The intuition behind is to apply a repair operation to the most likely correct location in order to avoid also some NPEs that are not triggered by the given test suite, thus eliminating plausible but incorrect patches. As illustrated in Figure 3(d), VFIX calculates the congestion value of each $\ell \in L_{\text{vf}}$ on L_{sta} by enumerating all the paths from the sources in $\{\ell_4, \ell_6\}$ to the sinks in $\{\ell_{16}, \ell_{17}\}$ on the VFG (Figure 3(c)). Thus, ℓ_8, which has the highest congestion value, is selected as the first repair location.

Step 3: Applying Value-Flow-Aware Repair Operations. VFIX generates the repair operations for a repair statement based on the value-flow information (to increase their success rate). For ℓ_8 selected, VFIX concludes that ℓ_{10} and ℓ_{12} are dependent on ℓ_8 and thus adds a null pointer check that encompasses $\ell_8 - \ell_{12}$, as illustrated in Figure 3(e), delivering a very first patch that is identical to the manual fix.

VFIX fixes the NPE bug in this example by selecting precisely a repair location and a repair operation. Without our value-flow analysis, either ℓ_4 or ℓ_{16} will likely be selected as a repair location. Thus, another potential NPE-triggering source ℓ_6 is ignored, producing a plausible but incorrect patch.

III. APPROACH

We describe VFIX's three steps for constructing static value-flow slices (Section III-A), selecting and ranking repair loca-

$$s \in \textsc{Stmt} \;::=\; p = q \mid p = q.f \mid p.f = q \mid p = \textbf{new}\ T \mid p = \text{null} \mid \textbf{return}_m\, p \mid p = q.m(\overrightarrow{r})$$

$$r, p, q \in \textsc{Ptr} \quad f \in \textsc{Field} \quad T \in \textsc{Class} \quad a, a' \in \textsc{Obj} \quad m \in \textsc{Method} \quad \ell, \ell', \ell'' \in \textsc{Label}$$

(a) A tiny Java-like language

Rule	Statement ℓ	Def-Use Information
NullAsgn	$p = \text{null}$	$\ell \in Def_p$
NewAsgn	$p = \textbf{new}\ T$	$\ell \in Def_p$
Copy	$p = q$	$\ell \in Def_p \quad \ell \in Use_q$
Load	$p = q.f$	$\ell \in Def_p \quad \ell \in Use_q$
	$a \in pts(q)$	$\ell \in Use_{a.f}$
Store	$p.f = q$	$\ell \in Use_q \quad \ell \in Def_{a.f}$
	$a \in pts(p)$	$\ell \in Use_p \quad \ell \in Use_{a.f}$
Call	$p = q.m(\overrightarrow{r})$	$\ell \in Def_p \quad \ell \in Use_q$ $\forall\ r_i \in \overrightarrow{r} : \ell \in Use_{r_i}$
MtdEntry	$m(\overrightarrow{r})\{\}$	$\forall\ r_i \in \overrightarrow{r} : \ell \in Def_{r_i}$
Ret	$\textbf{return}_m\ p$	$\ell \in Use_p$

(b) Intra-procedural def-use information statement-wise (Def_v (Use_v) denotes the set of definition (use) statements for a variable v)

[INTRA-POINTER]

$$\frac{\ell \in Def_p \quad IntraDU(\ell, \ell', p) \quad \ell' \in Use_p}{\ell \xrightarrow{p} \ell'}$$

[INTRA-OBJECT]

$$\frac{\ell \in Def_{a.f} \quad IntraDU(\ell, \ell', a.f) \quad \ell' \in Use_{a.f}}{\ell \xrightarrow{a.f} \ell'}$$

[INTER-POINTER] $\quad m' = dispatch(a, m)$

$$\frac{\ell : p = q.m(\overrightarrow{r}) \quad a \in pts(q) \quad \ell' : m'(\overrightarrow{r'}) \quad \ell'' : ret_{m'}\ p'}{\ell \xrightarrow{q} \ell' \quad \forall\ r_i \in \overrightarrow{r} : \ell \xrightarrow{r_i} \ell' \quad \ell'' \xrightarrow{p'} \ell}$$

[INTER-OBJECT]

$$\frac{\ell \in Def_{a.f} \quad InterDU(\ell, \ell', a.f) \quad \ell' \in Use_{a.f}}{\ell \xrightarrow{a.f} \ell'}$$

(c) Value-flow edges

Fig. 4: Value-flow graph construction.

tions based on static value-flow slices (Section III-B), and determining value-flow-aware repair operations (Section III-C).

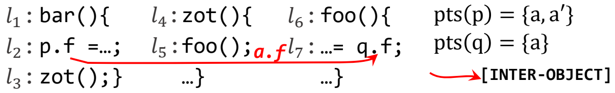
A. Constructing Static Value-Flow Slices

Given a buggy program with one NPE crash site, VFix builds an inter-procedural *value-flow graph* (VFG) $G_{\text{sta}} = (L_{\text{sta}}, E_{\text{sta}})$, 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 intra-procedural value-flow for variables and fields, respectively, [INTRA-POINTER] and [INTRA-OBJECT] find the intra-procedural definition sites ℓ of a variable or field from its use site ℓ' via $IntraDU$ (Definition 1). [INTER-POINTER] and [INTER-OBJECT] handle the inter-procedural value-flow for variables and fields, respectively. In [INTER-POINTER], the inter-procedural def-use relations for variables are obtained directly via parameter/return passing. In [INTER-OBJECT] (illustrated in Figure 5), the inter-procedural def-use relations for fields are obtained via $InterDU$ (Definition 2).



```
l1: bar(){        l4: zot(){       l6: foo(){
l2: p.f =...;     l5: foo();       l7: ...= q.f;
l3: zot();}          ...}             ...}
```

pts(p) = {a, a′}
pts(q) = {a}

Fig. 5: An inter-procedural value-flow for field $a.f$.

Definition 1 (Intra-Procedural Def-Use)**.** $IntraDU(\ell, \ell', v)$, 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., $\nexists\ \ell'' \in P : \ell'' \in Def_v$.

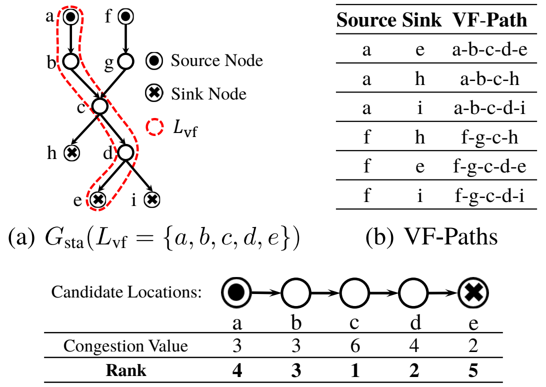
Definition 2 (Inter-Procedural Def-Use)**.** $InterDU(\ell, \ell', a.f)$ represents a def-use relation for $a.f$ from ℓ to ℓ' (with both in two distinct methods) on an inter-procedural control-flow path P, which is discovered context-sensitively [66], with no redefinition of $a.f$ in between, i.e., $\nexists\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

Example 1. Figure 5 shows a value-flow for $a.f$ across `bar()`, `zot()` and `foo()` via the two callsites ℓ_3 and ℓ_5. Here, $a.f$ is defined at $\ell_2 : p.f = ..$ in `bar()` and used at $\ell_7 : .. = q.f$ in `foo()`, where $p.f$ and $q.f$ are aliases since $pts(p) \cap pts(q) = \{a\}$, on the inter-procedural control-flow path $\ell_2, \ell_3, \ell_4, \ell_5, \ell_6, \ell_7$, 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 $G_{\text{sta}} = (L_{\text{sta}}, E_{\text{sta}})$ by solving $G_{i+1} = f_{\text{vfg}}(G_i)$ 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 and then all the (direct and indirect) uses of the variables or fields in G_i to G_i. 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



(a) $G_{\text{sta}}(L_{\text{vf}} = \{a, b, c, d, e\})$

Source	Sink	VF-Path
a	e	a-b-c-d-e
a	h	a-b-c-h
a	i	a-b-c-d-i
f	h	f-g-c-h
f	e	f-g-c-d-e
f	i	f-g-c-d-i

(b) VF-Paths

	a	b	c	d	e
Congestion Value	3	3	6	4	2
Rank	**4**	**3**	**1**	**2**	**5**

(c) Ranks

Fig. 6: An example for ranking repair locations.

the repair statements in the dynamic slice $L_{\text{vf}} = L_{\text{dyn}} \cap L_{\text{sta}}$, 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(ℓ_s, ℓ_t), is a simple path (with no repeated cycles), such that ℓ_s is a source and ℓ_t is a sink in G_{sta}. For a fixed source-sink pair (ℓ_s, ℓ_t), VF-Path-Set(ℓ_s, ℓ_t) is the set of all $\ell_s - \ell_t$ VF-paths. The *congestion* value of a node ℓ in VF-Path-Set(ℓ_s, ℓ_t) is given by $\frac{N_\ell^{\ell_s,\ell_t}}{|\text{VF-Path-Set}(\ell_s,\ell_t)|}$, where $N_\ell^{\ell_s,\ell_t}$ is the number of $\ell_s - \ell_t$ VF-paths that passes through ℓ. The *congestion* value of a node ℓ in G_{sta} is given by $\sum_{\ell_s \in Src} \sum_{\ell_t \in Snk} \frac{N_\ell^{\ell_s,\ell_t}}{|\text{VF-Path-Set}(\ell_s,\ell_t)|}$.

VFIX then ranks the statements in L_{vf} in the non-increasing order of their congestion values calculated for G_{sta} based on the definition above. A reverse topological order in G_{sta} is used as a tie-breaker (with the statements in the same strongly connected component (SCC) being ranked arbitrarily).

In G_{sta}, a source node can be p = null or **return** null, a potential source for causing a null dereference, and a sink node can be a potential point for triggering a null dereference.

Example 2. Figure 6 illustrates how to select and rank repair statements. For G_{sta} given in Figure 6(a), we assume that $L_{\text{vf}} = \{a, b, c, d, e\}$. 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-Flow-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: $G_{\text{sta}} = (L_{\text{sta}}, E_{\text{sta}})$ (a static slice)
       $L_{\text{vf}}$ (a set of repair statements)
Output: a linear ordering of the repair statements in $L_{\text{vf}}$
1  Function RankingRepairLocations($G_{\text{sta}}, L_{\text{vf}}$)
2      $G'_{\text{sta}} = (L'_{\text{sta}}, E'_{\text{sta}}) \leftarrow G_{\text{sta}}$ with all SCCs collapsed;
3      Let $SRC'_{sta}$ be the set of source nodes in $G'_{\text{sta}}$;
4      Let $SNK'_{sta}$ be the set of sink nodes in $G'_{\text{sta}}$;
5      foreach $\ell \in L'_{sta}$ do
6          foreach $\ell_s \times \ell_t \in SRC'_{sta} \times SNK'_{sta}$ do
7              $N_\ell^{\ell_s,\ell_t} \leftarrow$ number of $\ell_s - \ell_t$ VF-paths that
                 passes through $\ell$ in $G'_{\text{sta}}$;
8              VF-Path-Set$(\ell_s, \ell_t) \leftarrow$ set of $\ell_s - \ell_t$ VF-paths
                 in $G'_{\text{sta}}$;
9              $C_\ell^{\ell_s,\ell_t} \leftarrow \frac{N_\ell^{\ell_s,\ell_t}}{|\text{VF-Path-Set}(\ell_s,\ell_t)|}$;
10         end
11         $C_\ell \leftarrow \sum_{\ell_s \in SRC'_{sta}} \sum_{\ell_t \in SNK'_{sta}} C_\ell^{\ell_s,\ell_t}$;
12     end
13 Define a linear ordering of the statements in $\ell' \in L'_{\text{sta}}$,
     $\gg_C$, in increasing order of their congestion values, $C_\ell$,
     by breaking ties with a reverse topological order;
14 Replace each SCC $\{\ell_1, \dots \ell_n\} \in L'_{\text{sta}}$ in $\gg_C$ by
     $\ell_1, \dots \ell_n$, where $\ell_i \in L_{\text{sta}}$, in any arbitrary order;
15 return $\gg_C$ restricted to the statements in $L_{\text{vf}}$;
```

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 assuming *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 IV-F, 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 I, operates on the solution space $L_{\text{vf}} \times O_{\text{vf}}$, 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 7. There are two sets of rules. The set of rules in Figure 7(a) is responsible for extracting the candidate variables or fields from a repair statement in $\ell \in L_{\text{vf}}$ that may participate in a repair operation for ℓ. The set of

$$\texttt{[V-ASSN-NULL]}\ \frac{\ell : p = \text{null} \quad \ell \in \text{SRC}_{\text{vf}}}{(\ell, p, \bot) \in \texttt{VF}} \quad \texttt{[V-ASSN]}\ \frac{\ell : p = q \quad \ell \notin \text{SNK}_{\text{vf}}}{(\ell, p, q) \in \texttt{VF}} \quad \texttt{[V-RET-NULL]}\ \frac{\ell : \textbf{return}\ \text{null} \quad \ell \in \text{SRC}_{\text{vf}}}{(\ell, \bot, \bot) \in \texttt{VF}} \quad \texttt{[V-RET]}\ \frac{\ell : \textbf{return}\ \ p \quad \ell \notin \text{SNK}_{\text{vf}}}{(\ell, \bot, p) \in \texttt{VF}}$$

$$\texttt{[V-LD-SNK]}\ \frac{\ell : p = q.f \quad \ell \in \text{SNK}_{\text{vf}}}{(\ell, p, q) \in \texttt{VF}} \qquad \texttt{[V-ST-SNK]}\ \frac{\ell : p.f = q \quad \ell \in \text{SNK}_{\text{vf}}}{(\ell, \bot, p) \in \texttt{VF}} \qquad \texttt{[V-CALL-SNK]}\ \frac{\ell : p = q.m(\overrightarrow{r}) \quad \ell \in \text{SNK}_{\text{vf}}}{(\ell, p, q) \in \texttt{VF}}$$

$$\texttt{[V-LD-INT]}\ \frac{\ell : p = q.f \quad \ell \in \text{INT}_{\text{vf}}}{(\ell, p, q.f) \in \texttt{VF}} \qquad \texttt{[V-ST-INT]}\ \frac{\ell : p.f = q \quad \ell \in \text{INT}_{\text{vf}}}{(\ell, p.f, q) \in \texttt{VF}} \qquad \texttt{[V-CALL-INT]}\ \frac{\ell : p = q.m(\overrightarrow{r}) \quad \ell \in \text{INT}_{\text{vf}}}{(\ell, p, q.m(\overrightarrow{r})) \in \texttt{VF}}$$

(a) Identifying the candidate variables or fields in a repair statement $\ell \in L_{\text{vf}}$ for repairing ℓ

$$\texttt{[INIT-SIMP]}\quad \frac{(\ell, x, y) \in \texttt{VF} \quad y \neq \bot \quad c : y == \text{null} \quad \ell' : y = \textbf{new}\ T}{\ell \Rightarrow \textbf{if}\ (c)\ \{\ell'\}; \ell}$$

$$\texttt{[INIT-SRC]}\quad \frac{(\ell, x, \bot) \in \texttt{VF} \quad \ell : \_ = \text{null} \Longrightarrow \ell' : x = \textbf{new}\ T \quad \ell : \textbf{return}\ \text{null} \Longrightarrow \ell' : \textbf{return new}\ T}{\ell \Rightarrow \ell'}$$

$$\texttt{[INIT-SNK]}\quad \frac{(\ell, x, y) \in \texttt{VF} \quad \ell \in \text{SNK}_{\text{vf}} \quad x \neq \bot \quad \nexists\, \ell'' \in Use_y :\ \ell''\ pDom\ \ell \quad c : y == \text{null} \quad \ell' : x = \textbf{new}\ T}{\ell \Rightarrow \textbf{if}\ (c)\ \{\ell'\}\ \textbf{else}\ \{\ell\}}$$

$$\texttt{[SKIP-ONE]}\quad \texttt{FSL}(\ell) = \{\ell\} \quad \frac{(\ell, x, y) \in \texttt{VF} \quad y \neq \bot \quad \ell \neq \textbf{return}\ \_ \quad c : y\ != \text{null}}{\ell \Rightarrow \textbf{if}\ (c)\ \{\ell\}}$$

$$\texttt{[SKIP-MULTI]}\quad \texttt{FSL}(\ell) = \{\ell, \ell_1, \cdots, \ell_n\} \quad n \geq 1 \quad \frac{(\ell, x, y) \in \texttt{VF} \quad y \neq \bot \quad \nexists\, \ell' \in \texttt{FSL}(\ell) : \ell' = \textbf{return}\ \_ \quad c : y\ != \text{null}}{\ell \Rightarrow \textbf{if}\ (c)\ \{\ell, \ell_1, \cdots, \ell_n\}}$$

$$\texttt{[SKIP-RET]}\quad \frac{(\ell, x, y) \in \texttt{VF} \quad y \neq \bot \quad c : y == \text{null} \quad \ell' : \textbf{return}\ r \quad r \in \{\textbf{new}\ T, \text{null}, \epsilon\}}{\ell \Rightarrow \textbf{if}(c)\ \{\ell'\}; \ell}$$

(b) Generating a repair operation for $\ell \in L_{vf}$ ($\ell \Rightarrow \ell'$ denotes the code transformation from ℓ to ℓ' and $\texttt{FSL}(\ell)$ is a forward slice of ℓ)

Fig. 7: Rules for applying value-flow-aware repair operations.

rules in Figure 7(b) generates the repair operations in O_{vf} 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_{sta} 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 $\ell \in L_{\text{vf}}$, we collect $(\ell, x, y) \in \texttt{VF}$, where x and y appear in ℓ, to indicate that y may be null on entry of ℓ and x may be null (or undefined when $\ell \in \text{SNK}_{\text{vf}}$) on exit of ℓ. [V-ASSN-NULL], [V-ASSN], [V-RET-NULL], and [V-RET] are self-explanatory, where $\ell \in \text{SRC}_{vf}$ and $\ell \notin \text{SNK}_{vf}$ are redundant. For a load ($\ell : p = q.f$), store ($\ell : p.f = q$), or call ($\ell : p = q.m(\overrightarrow{r})$, there are two rules each, depending on whether ℓ 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 [20, 34]. 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-*], [V-ST-*] and [V-CALL-*]; [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 for 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 ℓ (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 ℓ. However, we will only do so when $\nexists\, \ell'' \in Use_y :\ \ell''\ pDom\ \ell$ holds, i.e., when y does not have any post-dominant use in Use_y with respect to ℓ. Otherwise, the repair operation for ℓ 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], $\texttt{FSL}(\ell)$ represents the traditional forward slice computed at ℓ.

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

In [SKIP-MULTI], we skip multiple statements that depend on a repair statement ℓ (when $\texttt{FSL}(\ell) = \{\ell, \ell_1, \ldots, \ell_n\} \neq \{\ell\}$). 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 ℓ in the method m containing ℓ, by returning nothing (ϵ) if m's return type is **void**, or **new** T if T is primitive as discussed for [INIT-SNK], or one in $\{$**new** T, 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.

TABLE I: NPEs in Defects4j

Project	Description	KLOC	Bug IDs	#NPEs
Chart	Plotting Software	96	2, 4, 14, 15, 16, 25, 26	7
Lang	Java Utility	22	20, 33, 39, 47, 57	5
Math	Mathematics Lib	85	4, 70, 79	3
Time	Calendar System	28	-	0
Total		231		**15**

TABLE II: NPEs in Apache Projects

Project	Description	KLOC	Bug IDs	#NPEs
Pdfbox	PDF Library	128	2266, 2477, 2812 2948, 2951, 2965 2995, 3479, 3572	9
Felix	OSGi Platform	25	4960, 5464	2
Collections	Collection Handling	69	39, 360	2
Sling	Web Framework	4	4982, 6487	2
Total		226	84	**15**

Example 3. Let us revisit Figure 1 to select the repair operations for the statement at ℓ_8, the NPE-triggering site. By [V-CALL-SNK], we obtain $(\ell_8, \texttt{removed}, \texttt{markers}) \in \texttt{VF}$. The skip templates are considered first. As $\texttt{FSL}(\ell_8) = \{\ell_8, \ell_9, \ell_{10}, \ell_{11}\}$ contains ℓ_{11}, a **return** statement, [SKIP-RET] applies, giving rise to the correct path shown in Figure 1. Without our value-flow analysis, [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 [76] 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_{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_{dyn} for the program under the NPE-triggering test case. $L_{\text{vf}} = L_{\text{sta}} \cap L_{\text{dyn}}$ then contains all the repair statements localized. To generate patches, we use JavaParser [3] 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 II 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

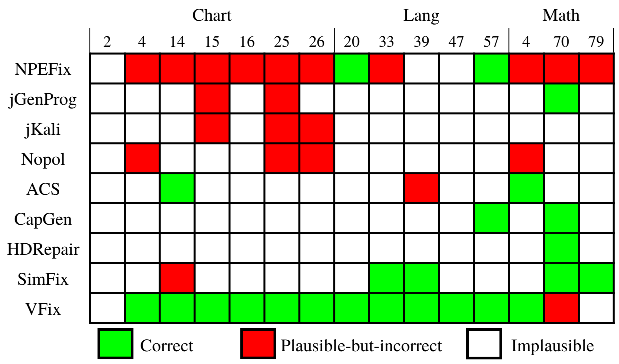


	Chart							Lang					Math		
	2	4	14	15	16	25	26	20	33	39	47	57	4	70	79
NPEFix	□	🟥	🟥	🟥	🟥	🟥	🟥	🟩	🟥	□	□	🟩	🟥	🟥	🟥
jGenProg	□	□	□	🟥	□	🟥	□	□	□	□	□	□	□	🟩	□
jKali	□	□	□	🟥	□	🟥	🟥	□	□	□	□	□	□	□	□
Nopol	□	🟥	□	□	□	🟥	🟥	□	□	□	□	□	🟥	□	□
ACS	□	□	🟩	□	□	□	□	□	□	🟥	□	□	🟩	□	□
CapGen	□	□	□	□	□	□	□	□	□	□	□	🟩	□	🟩	□
HDRepair	□	□	□	□	□	□	□	□	□	□	□	□	□	🟩	□
SimFix	□	□	🟥	□	□	□	□	□	🟩	🟩	□	□	□	🟩	🟩
VFix	□	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟥	□

🟩 Correct 🟥 Plausible-but-incorrect □ Implausible

Fig. 8: Comparing the correctness of the patches generated by VFix and eight existing representative APR tools for `Defects4j`. The numbers across are the bug IDs corresponding to Column 4 in Table I. Only the applications containing NPEs, i.e., `Chart`, `Lang` and `Math`, are included here.

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 III: 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.

Bug ID	Time (secs)				N-th patch passing test	Correctness
	Step 1	Step 2	Step 3	Total		
Chart-2	37	16	OOB	OOB	∞	□
Chart-4	45	14	34	93	1	🟩
Chart-14	41	14	32	87	1	🟩
Chart-15	48	13	46	107	1	🟩
Chart-16	13	8	10	31	1	🟩
Chart-25	41	13	37	91	1	🟩
Chart-26	43	3	49	95	1	🟩
Lang-20	14	3	13	30	2	🟩
Lang-33	15	4	17	36	1	🟩
Lang-39	14	3	16	33	1	🟩
Lang-47	13	3	16	32	3	🟩
Lang-57	12	3	12	27	1	🟩
Math-4	16	11	7	34	1	🟩
Math-70	13	8	9	30	2	🟥
Math-79	12	6	OOB	OOB	∞	□
Felix-4960	23	7	19	49	1	🟩
Felix-5464	14	8	9	31	1	🟩
Collections-39	13	4	11	28	2	🟩
Collections-360	12	7	8	27	1	🟩
Pdfbox-2266	50	19	57	126	1	🟩
Pdfbox-2477	52	14	66	132	2	🟩
Pdfbox-2812	49	24	OOB	OOB	∞	□
Pdfbox-2948	52	17	61	130	1	🟩
Pdfbox-2951	54	20	OOB	OOB	∞	□
Pdfbox-2965	51	17	47	115	1	🟩
Pdfbox-2995	50	18	44	112	1	🟩
Pdfbox-3479	55	19	50	124	2	🟩
Pdfbox-3572	55	16	38	109	1	🟩
Sling-4982	12	1	16	29	1	🟥
Sling-6487	57	1	60	118	1	🟩
Total (with OOB)	976	314	2966	4256		
(w/o OOB)	824	248	784	1856		

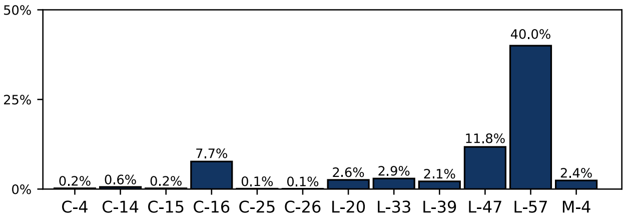


Fig. 9: The percentages of the number of repair statements in L_{vf} found by VFix over the total number of suspicious statements reported by the fault localization tool, GZoltar [12], for the 12 NPEs in `Defects4j` fixed by VFix (with `Chart`, `Lang` and `Math` abbreviated to `C`, `L`, and `M`, respectively).

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 $L_{\text{vf}} \times O_{\text{vf}}$ 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_{vf}. By leveraging the value-flow information for an NPE-triggering site, VFix avoids many irrelevant repair

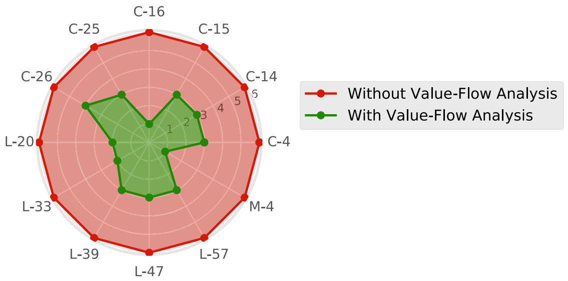


Fig. 10: The number of repair templates instantiated at each bug-fixing statement with and without value-flow analysis for the 12 NPEs in Defects4j fixed by VFix (with Chart, Lang and Math abbreviated to C, L, and M, respectively).

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_{vf} 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 $S_{\text{vf}} = L_{\text{vf}} \times O_{\text{vf}}$.

Figure 11(a) shows an NPE bug, Chart-4, and its patch generated by VFix. The bug happens in ℓ_{4494}, where r can be null. VFix adds a null check in ℓ_{4493} and ℓ_{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 ℓ_{4494}) would leave some variables that are data-dependent on the skipped statements to be undefined (e.g., c in ℓ_{4495} and a in ℓ_{4499}). Closing it too late would introduce unnecessary control-dependencies on the if-statement added (in ℓ_{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 impractical. With value-flow analysis, resulting in a precise O_{vf}, VFix can correctly identify the scope of the inserted if branch by [SKIP-MULTI].

```
4493 +if(r != null){
4494   Collection c = r.getAnnotations();
4495   Iterator I = c.iterator();
4496   while(i.hasNext()){
4497     XYAnnotation a = (XYAnnotation)i.next();
4498     if(a instanceof XYAnnotationBoundsInfo){
4499       includedAnnotations.add(a);
4450     }
4451   }
4452 +}
```

(a) An NPE, Chart-4, correctly repaired with a precise O_{vf}

```
142 DefaultIntervalCategoryDataset(…) {
    …
207 - this.categoryKeys = null;
208 + this.categoryKeys = new Comparable[0];
209 }
    …
573 public int getCategoryIndex(…){
574   for(int i=0;i<this.categoryKeys.length;i++)
      …
581 }
    …
689 public int getColumnCount(){
690   return this.categoryKeys.length;
691 }
```

(b) An NPE, Chart-16, correctly repaired with a precise L_{vf}

Fig. 11: Two case studies in Defects4j (with the patches generated by VFix shown by + anf -).

Figure 11(b) illustrates another NPE bug, Chart-16, repaired by VFix. This bug happens in ℓ_{574} as this.category is not initialized in ℓ_{207} in the constructor DefautlIntervalCategoryDataset(...). Given ℓ_{574}, G_{sta} built by VFix contains not only ℓ_{574} but also ℓ_{690}, another potential NPE crash site. For the given NPE-triggering test case, L_{vf} contains ℓ_{208} and ℓ_{574} (but not ℓ_{690} as it is not executed). VFix finds the correct patch, which initializes this.category in ℓ_{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_{vf} precisely by their congestion values, is also instrumental here. As G_{sta} contains both ℓ_{574} and ℓ_{690}, ℓ_{208} is ranked ahead of ℓ_{574}, enabling VFix to generate the correct fix in ℓ_{208}. Otherwise, adding the same fix just before ℓ_{574} is only plausible but incorrect, as it fails to fix the other NPE crash site in ℓ_{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 f 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 using, 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 [77] 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-sparsity 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 Againist Null Pointer Dereferences

The NPEs that cause program crashes can be detected by a variety of testing techniques such as fuzzing [1, 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 localiaztion 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.

References

[1] American fuzzy lop (afl) fuzzer. http://lcamtuf.coredump.cx/afl.

[2] Apache projects issues. https://issues.apache.org/jira/projects.

[3] Javaparser. https://javaparser.org/.

[4] R. Abreu, P. Zoeteweij, and A. J. Van Gemund. On the accuracy of spectrum-based fault localization. In *TAICPART-MUTATION '07*, pages 89–98, 2007.

[5] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. In *PLDI '14*, pages 259–269, 2014.

[6] D. Babic and A. J. Hu. Calysto: Scalable and precise extended static checking. In *ICSE '08*, pages 211–220, 2008.

[7] R. Bodík and S. Anik. Path-sensitive value-flow analysis. In *POPL '98*, pages 237–251, 1998.

[8] M. Böhme, V.-T. Pham, M.-D. Nguyen, and A. Roychoudhury. Directed greybox fuzzing. In *CCS '17*, pages 2329–2344, 2017.

[9] M. D. Bond, N. Nethercote, S. W. Kent, S. Z. Guyer, and K. S. McKinley. Tracking bad apples: reporting the origin of null and undefined value errors. In *OOPSLA 07*, pages 405–422, 2007.

[10] E. Bounimova, P. Godefroid, and D. Molnar. Billions and billions of constraints: Whitebox fuzz testing in production. In *ICSE '13*, pages 122–131, 2013.

[11] C. Cadar, D. Dunbar, D. R. Engler, et al. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI '08*, pages 209–224, 2008.

[12] J. Campos, A. Riboira, A. Perez, and R. Abreu. Gzoltar: an eclipse plug-in for testing and debugging. In *ASE '12*, pages 378–381, 2012.

[13] X. Cheng, M. Zhou, X. Song, M. Gu, and J. Sun. Intpti: Automatic integer error repair with proper-type inference. In *ASE '17*, pages 996–1001, 2017.

[14] S. Cherem, L. Princehouse, and R. Rugina. Practical memory leak detection using guarded value-flow analysis. In *PLDI '07*, pages 480–491, 2007.

[15] J. Chuzhoy. Routing in undirected graphs with constant congestion. In *STOC '12*, pages 855–874, 2012.

[16] J. Chuzhoy, V. Guruswami, S. Khanna, and K. Talwar. Hardness of routing with congestion in directed graphs. In *STOC '07*, pages 165–178, 2007.

[17] M. Cielecki, J. Fulara, K. Jakubczyk, and Ł. Jancewicz. Propagation of jml non-null annotations in java programs. In *PPPJ '06*, pages 135–140. ACM, 2006.

[18] Z. Coker and M. Hafiz. Program transformations to fix c integers. In *ICSE '13*, pages 792–801, 2013.

[19] M. Das, S. Lerner, and M. Seigle. Esp: Path-sensitive program verification in polynomial time. In *PLDI '02*, pages 57–68, 2002.

[20] K. Dobolyi and W. Weimer. Changing java's semantics for handling null pointer exceptions. In *ISSRE '08*, pages 47–56, 2008.

[21] N. Dor, S. Adams, M. Das, and Z. Yang. Software validation via scalable path-sensitive value flow analysis. In *ISSTA '04*, pages 12–22, 2004.

[22] T. Durieux, B. Cornu, L. Seinturier, and M. Monperrus. Dynamic patch generation for null pointer exceptions using metaprogramming. In *SANER '17*, pages 349–358, 2017.

[23] Q. Gao, Y. Xiong, Y. Mi, L. Zhang, W. Yang, Z. Zhou, B. Xie, and H. Mei. Safe memory-leak fixing for c programs. In *ICSE '15*, pages 459–470, 2015.

[24] Q. Gao, H. Zhang, J. Wang, Y. Xiong, L. Zhang, and H. Mei. Fixing recurring crash bugs via analyzing q&a sites (T). In *ASE '15*, pages 307–318, 2015.

[25] P. Godefroid, N. Klarlund, and K. Sen. Dart: Directed automated random testing. In *PLDI '05*, pages 213–223, 2005.

[26] M. I. Gordon, D. Kim, J. H. Perkins, L. Gilham, N. Nguyen, and M. C. Rinard. Information flow analysis of android applications in droidsafe. In *NDSS '15*, page 110, 2015.

[27] T. Gu, C. Sun, X. Ma, J. Lü, and Z. Su. Automatic runtime recovery via error handler synthesis. In *ASE'16*, pages 684–695, 2016.

[28] J. Jiang, Y. Xiong, H. Zhang, Q. Gao, and X. Chen. Shaping program repair space with existing patches and similar code. In *ISSTA '18*, pages 298–309, 2018.

[29] G. Jin, L. Song, W. Zhang, S. Lu, and B. Liblit. Automated atomicity-violation fixing. In *PLDI '11*, pages 389–400, 2011.

[30] R. Just, D. Jalali, and M. D. Ernst. Defects4j: A database of existing faults to enable controlled testing studies for java programs. In *ISSTA '14*, pages 437–440, 2014.

[31] S. Kaleeswaran, V. Tulsian, A. Kanade, and A. Orso. Minthint: automated synthesis of repair hints. In *ICSE '14*, pages 266–276, 2014.

[32] Y. Ke, K. T. Stolee, C. Le Goues, and Y. Brun. Repairing programs with semantic code search (T). In *ASE '15*, pages 295–306, 2015.

[33] S. W. Kent. Dynamic error remediation: A case study with null pointer exceptions. *University of Texas Masters thesis*, 2008.

[34] D. Kim, J. Nam, J. Song, and S. Kim. Automatic patch generation learned from human-written patches. In *ICSE '13*, pages 802–811, 2013.

[35] X.-B. D. Le, D.-H. Chu, D. Lo, C. Le Goues, and W. Visser. Jfix: Semantics-based repair of java programs via symbolic pathfinder. In *ISSTA '17*, pages 376–379, 2017.

[36] X.-B. D. Le, D.-H. Chu, D. Lo, C. Le Goues, and W. Visser. S3: Syntax-and semantic-guided repair synthesis via programming by examples. In *ESEC/FSE'17*, pages 593–604, 2017.

[37] X. D. Le, D. Lo, and C. Le Goues. History driven program repair. In *SANER '16*, pages 213–224, 2016.

[38] C. Le Goues, T. Nguyen, S. Forrest, and W. Weimer. Genprog: A generic method for automatic software repair. *TSE*, 38(1):54–72, 2012.

[39] J. Lee, S. Hong, and H. Oh. Memfix: static analysis-based repair of memory deallocation errors for c. In *ESEC/FSE'18*, pages 95–106, 2018.

[40] L. Li, C. Cifuentes, and N. Keynes. Boosting the performance of flow-sensitive points-to analysis using value flow. In *ESEC/FSE '11*, pages 343–353, 2011.

[41] Z. Li, L. Tan, X. Wang, S. Lu, Y. Zhou, and C. Zhai. Have things changed now?: an empirical study of bug characteristics in modern open source software. In *ASID '06*, pages 25–33, 2006.

[42] A. Loginov, E. Yahav, S. Chandra, S. Fink, N. Rinetzky, and M. Nanda. Verifying dereference safety via expanding-scope analysis. In *ISSTA '08*, pages 213–224, 2008.

[43] F. Logozzo and T. Ball. Modular and verified automatic program repair. In *OOPSLA '12*, pages 133–146, 2012.

[44] F. Long and M. Rinard. Automatic patch generation by learning correct code. In *POPL '16*, pages 298–312, 2016.

[45] F. Long, S. Sidiroglou-Douskos, and M. Rinard. Automatic runtime error repair and containment via recovery shepherding. In *PLDI '14*, pages 227–238, 2014.

[46] K. Lu, C. Song, T. Kim, and W. Lee. Unisan: Proactive kernel memory initialization to eliminate data leakages. In *CCS '16*, pages 920–932, 2016.

[47] R. Madhavan and R. Komondoor. Null dereference verification via over-approximated weakest pre-conditions analysis. In *OOPSLA '11*, pages 1033–1052, 2011.

[48] M. Martinez, T. Durieux, R. Sommerard, J. Xuan, and M. Monperrus. Automatic repair of real bugs in java: A large-scale experiment on the defects4j dataset. *Empirical Software Engineering*, 22(4):1936–1964, 2017.

[49] S. Mechtaev, J. Yi, and A. Roychoudhury. Directfix: Looking for simple program repairs. In *ICSE '15*, pages 448–458, 2015.

[50] S. Mechtaev, J. Yi, and A. Roychoudhury. Angelix: Scalable

multiline program patch synthesis via symbolic analysis. In *ICSE '16*, pages 691–701, 2016.

[51] A. Milanova, A. Rountev, and B. G. Ryder. Parameterized object sensitivity for points-to analysis for java. *TOSEM*, 14(1):1–41, 2005.

[52] M. G. Nanda and S. Sinha. Accurate interprocedural null-dereference analysis for java. In *ICSE '09*, pages 133–143, 2009.

[53] H. D. T. Nguyen, D. Qi, A. Roychoudhury, and S. Chandra. Semfix: Program repair via semantic analysis. In *ICSE '13*, pages 772–781, 2013.

[54] Y. Qi, X. Mao, Y. Lei, Z. Dai, and C. Wang. The strength of random search on automated program repair. In *ICSE '14*, pages 254–265, 2014.

[55] Z. Qi, F. Long, S. Achour, and M. C. Rinard. An analysis of patch plausibility and correctness for generate-and-validate patch generation systems. In *ISSTA '15*, pages 24–36, 2015.

[56] M. Renieres and S. P. Reiss. Fault localization with nearest neighbor queries. In *ASE '03*, pages 30–39, 2003.

[57] T. Reps, S. Horwitz, and M. Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *POPL '95*, pages 49–61, 1995.

[58] R. K. Saha, Y. Lyu, H. Yoshida, and M. R. Prasad. ELIXIR: effective object oriented program repair. In *ASE '17*, pages 648–659, 2017.

[59] K. Sen, D. Marinov, and G. Agha. Cute: A concolic unit testing engine for c. In *ESEC/FSE'05*, pages 263–272, 2005.

[60] A. Shaw, D. Doggett, and M. Hafiz. Automatically fixing c buffer overflows using program transformations. In *DSN '14*, pages 124–135, 2014.

[61] Q. Shi, X. Xiao, R. Wu, J. Zhou, G. Fan, and C. Zhang. Pinpoint: Fast and precise sparse value flow analysis for million lines of code. In *PLDI '18*, pages 693–706, 2018.

[62] Y. Smaragdakis, M. Bravenboer, and O. Lhoták. Pick your contexts well: understanding object-sensitivity. In *POPL '11*, pages 17–30, 2011.

[63] J. Späth, K. Ali, and E. Bodden. Ide al: Efficient and precise alias-aware dataflow analysis. In *OOPSLA '17*, page 99, 2017.

[64] J. Späth, L. Nguyen Quang Do, K. Ali, and E. Bodden. Boomerang: Demand-driven flow-and context-sensitive pointer analysis for java. In *ECOOP '16*, pages 22:1–22:26, 2016.

[65] F. Spoto. Precise null-pointer analysis. *Software & Systems Modeling*, 10(2):219–252, 2011.

[66] M. Sridharan and R. Bodík. Refinement-based context-sensitive points-to analysis for java. In *PLDI '06*, pages 387–400, 2006.

[67] B. Steffen, J. Knoop, and O. Rüthing. The value flow graph: A program representation for optimal program transformations. In *ESOP '90*, pages 389–405, 1990.

[68] F. Steimann, M. Frenkel, and R. Abreu. Threats to the validity and value of empirical assessments of the accuracy of coverage-based fault locators. In *ISSTA '13*, pages 314–324, 2013.

[69] Y. Sui and J. Xue. On-demand strong update analysis via value-flow refinement. In *FSE '16*, pages 460–473, 2016.

[70] Y. Sui and J. Xue. SVF: interprocedural static value-flow analysis in llvm. In *CC '16*, pages 265–266, 2016.

[71] Y. Sui, D. Ye, and J. Xue. Static memory leak detection using full-sparse value-flow analysis. In *ISSTA '12*, pages 254–264, 2012.

[72] S. H. Tan, Z. Dong, X. Gao, and A. Roychoudhury. Repairing crashes in android apps. In *ICSE '18*, 2018.

[73] T. Tan, Y. Li, and J. Xue. Making k-object-sensitive pointer analysis more precise with still k-limiting. In *SAS '16*, pages 489–510, 2016.

[74] T. Tan, Y. Li, and J. Xue. Efficient and precise points-to analysis: modeling the heap by merging equivalent automata. *PLDI '17*, 52(6):278–291, 2017.

[75] Y. Tian and B. Ray. Automatically diagnosing and repairing error handling bugs in c. In *ESEC/FSE'17*, pages 752–762, 2017.

[76] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan. Soot: A java bytecode optimization framework. In *CASCON '99*, pages 13–, 1999.

[77] R. van Tonder and C. Le Goues. Static automated program repair for heap properties. In *ICSE '18*, pages 151–162, 2018.

[78] W. Weimer, Z. P. Fry, and S. Forrest. Leveraging program equivalence for adaptive program repair: Models and first results. In *ASE '13*, pages 356–366, 2013.

[79] W. Weimer, T. Nguyen, C. Le Goues, and S. Forrest. Automatically finding patches using genetic programming. In *ICSE'09*, pages 364–374, 2009.

[80] M. Weiser. Programmers use slices when debugging. *Communications of the ACM*, 25(7):446–452, 1982.

[81] M. Wen, J. Chen, R. Wu, D. Hao, and S.-C. Cheung. Context-aware patch generation for better automated program repair. In *ICSE '18*, 2018.

[82] R. Wismüller. Debugging of globally optimized programs using data flow analysis. In *PLDI '94*, pages 278–289, 1994.

[83] W. E. Wong, R. Gao, Y. Li, R. Abreu, and F. Wotawa. A survey on software fault localization. *TSE*, 42(8):707–740, 2016.

[84] Q. Xin and S. P. Reiss. Leveraging syntax-related code for automated program repair. In *ASE '17*, pages 660–670, 2017.

[85] Y. Xiong, X. Liu, M. Zeng, L. Zhang, and G. Huang. Identifying patch correctness in test-based program repair. In *ICSE '18*, pages 789–799, 2018.

[86] Y. Xiong, J. Wang, R. Yan, J. Zhang, S. Han, G. Huang, and L. Zhang. Precise condition synthesis for program repair. In *ICSE '17*, pages 416–426, 2017.

[87] J. Xuan, M. Martinez, F. Demarco, M. Clement, S. R. L. Marcote, T. Durieux, D. L. Berre, and M. Monperrus. Nopol: Automatic repair of conditional statement bugs in java programs. *TSE*, 43(1):34–55, 2017.

[88] J. Xuan and M. Monperrus. Learning to combine multiple ranking metrics for fault localization. In *ICSME '14*, pages 191–200, 2014.

[89] H. Yan, Y. Sui, S. Chen, and J. Xue. Automated memory leak fixing on value-flow slices for c programs. In *SAC '16*, pages 1386 – 1393, 2016.

[90] H. Yan, Y. Sui, S. Chen, and J. Xue. Spatio-temporal context reduction: A pointer-analysis-based static approach for detecting use-after-free vulnerabilities. In *ICSE '18*, pages 327–337, 2018.

[91] D. Ye, Y. Sui, and J. Xue. Accelerating dynamic detection of uses of undefined values with static value-flow analysis. In *CGO '14*, page 154, 2014.

[92] C. Zhang, T. Wang, T. Wei, Y. Chen, and W. Zou. Intpatch: Automatically fix integer-overflow-to-buffer-overflow vulnerability at compile-time. In *ESORICS '10*, pages 71–86, 2010.