Detecting, Creating, Repairing, and Understanding Indivisible Multi-Hunk Bugs

QI XIN, School of Computer Science, Wuhan University; Hubei Luojia Laboratory, China
HAOJUN WU, School of Computer Science, Wuhan University, China
JINRAN TANG, School of Computer Science, Wuhan University, China
XINYU LIU, School of Computer Science, Wuhan University, China
STEVEN P. REISS, Department of Computer Science, Brown University, USA
JIFENG XUAN*, School of Computer Science, Wuhan University, China

This paper presents our approach proposed to detect and create indivisible multi-hunk bugs, an evaluation of existing repair techniques based on these bugs, and a study of the patches of these bugs constructed by the developers and existing tools. Multi-hunk bug repair aims to deal with complex bugs by fixing multiple locations of the program. Previous research on multi-hunk bug repair is severely misguided, as the evaluation of previous techniques is predominantly based on the Defects4J dataset containing a great deal of *divisible* multi-hunk bugs. A divisible multi-hunk bug is essentially a combination of multiple bugs triggering different failures and is uncommon while debugging, as the developer typically deals with one failure at a time. To address this problem and provide a better basis for multi-hunk bug repair, we propose an enumeration-based approach IBugFinder, which given a bug dataset can automatically detect divisible and indivisible bugs in the dataset and further isolate the divisible bugs into new indivisible bugs. We applied IBugFinder to 281 multi-hunk bugs from the Defects4J dataset. IBugFinder identified 139 divisible bugs and created 249 new bugs among which 105 are multi-hunk.

We evaluated existing repair techniques with the indivisible multi-hunk bugs detected and created by IBugFinder and found that these techniques repaired only a small number of bugs suggesting weak multi-hunk repair abilities. We further studied the patches of indivisible multi-hunk bugs constructed by the developers and the various tools with a focus on understanding the relationships of the partial patches made at different locations. The study has led to the identification of 8 partial patch relationships, which suggest different strategies for multi-hunk patch generation and provide important implication for multi-hunk bug repair.

CCS Concepts: • **Software and its engineering → Software creation and management**.

Additional Key Words and Phrases: Automated program repair, indivisible multi-hunk bugs, partial patch relationship

ACM Reference Format:
Qi Xin, Haojun Wu, Jinran Tang, Xinyu Liu, Steven P. Reiss, and Jifeng Xuan. 2024. Detecting, Creating, Repairing, and Understanding Indivisible Multi-Hunk Bugs. *Proc. ACM Softw. Eng.* 1, FSE, Article 121 (July 2024), 24 pages. https://doi.org/10.1145/3660828

*Corresponding author.

Authors' Contact Information: Qi Xin, qxin@whu.edu.cn, School of Computer Science, Wuhan University; Hubei Luojia Laboratory, China; Haojun Wu, haojunwu@whu.edu.cn, School of Computer Science, Wuhan University, China; Jinran Tang, jinrantang@whu.edu.cn, School of Computer Science, Wuhan University, China; Xinyu Liu, xinyuliu4@whu.edu.cn, School of Computer Science, Wuhan University, China; Steven P. Reiss, spr@cs.brown.edu, Department of Computer Science, Brown University, USA; Jifeng Xuan, jxuan@whu.edu.cn, School of Computer Science, Wuhan University, China.

ACM 2994-970X/2024/7-ART121
https://doi.org/10.1145/3660828

1 INTRODUCTION

Debugging is laborious. Automated program repair (APR) holds the promise of automatically fixing software bugs to significantly reduce the cost of debugging. Over the years, many APR techniques [26, 49, 88] have been proposed. They adopt various strategies to achieve the goal and have been generally classified as pattern-based (e.g., [35, 42]), search-based (e.g., [31, 69]), constraint-based (e.g., [46, 77]), and learning-based [88] techniques. With the rise of Large Language Models (LLMs), recent APR techniques [32, 52, 59, 71, 73] have shown great potential in fixing a single location of the program for bug correction.

Despite the remarkable progress made, APR is still far from being practically adopted for real-world debugging. A key reason has to do with its weakness in addressing complex *multi-hunk* bugs whose repair requires correcting multiple, non-contiguous code sections generally referred to as *code hunks* [57]. There has been evidence [34, 89] showing that multi-hunk bugs are common. In particular, Zhong and Su [89] found that at least 40% of real bug fixes require changes of more than one source file. The percentage of multi-hunk fixes should be higher, as their result does not account for changes affecting multiple locations in one source file. This result implies that APR techniques that do not support multi-hunk repair can only handle a limited fraction of real bugs and have limited usefulness. To further extend the repair scope of APR, various multi-hunk-oriented techniques have been proposed [39, 41, 46, 57, 70, 80, 83]. They achieve multi-hunk repair via strategies such as evolutionary search [39, 83], detection and update of evolutionary siblings (where similar fixes can be applied) [57], variational execution [70], deep learning [36, 41], and iterative self-supervised training [80].

Current research on multi-hunk bug repair suffers from two weaknesses. First, existing techniques were often evaluated on the Defects4J dataset [34], which is deeply flawed for multi-hunk repair evaluation, as about half (or 49.5% according to our result) of the multi-hunk bugs contained in the dataset are *divisible*. A divisible bug is essentially a combination of multiple independent bugs triggering different failures. Repairing a divisible bug by handling multiple failures at the same time is uncommon for debugging [37, 51], as a developer typically deals with one failure at a time [37, 51]. The effectiveness of divisible bug repair also does not reflect the core abilities of APR in patching multiple locations to address complex bugs. This is because for a divisible bug an APR technique can easily detect a promising patch that it generates by checking whether the patch can resolve any of the failures. Once identifying a promising patch, the APR technique can further build on the patch for bug correction, which ensures a significantly reduced search space. In general, however, promising patches are never easy to detect, as they may not indicate any obvious repair progress (consider a patch that only adds the definition of a variable needed for repair).

Second, while there have been efforts made towards addressing multi-hunk bugs [39, 41, 46, 57, 70, 80, 83], the very key questions related to the generation of multi-hunk patches are still left unanswered. Most importantly, it remains unclear why repairing a multi-hunk bug needs to address multiple locations, what are the characteristics of the fixes made at different locations, and furthermore what strategies one should consider for multi-hunk patch generation based on the various characteristics. Answers to these questions are important, as they can provide insights into effective multi-hunk patch generation to advance the state of the art.

In this paper, we propose our solutions to address the two weaknesses. Our first effort is dedicated to detecting and creating indivisible multi-hunk bugs to provide a better basis for multi-hunk repair research. We propose an enumeration-based approach IBugFinder (Indivisible Bug Finder). Given a bug dataset, which includes for each bug the original program, the test cases, and the developer patch (ground-truth fix), IBugFinder identifies multi-hunk bugs, determines the divisibility of each, and further isolates the bugs detected as divisible to create new indivisible bugs. We implemented

IBugFinder and applied it to all the 281 multi-hunk bugs from six projects in the Defects4J-v2.0 dataset used to evaluate most existing APR techniques. IBugFinder determined 139 (49.5%) multi-hunk bugs as divisible, 118 bugs as indivisible, and the remaining as unknown due to bug complexity and deprecation. Furthermore, it isolated the divisible bugs and created 249 new indivisible bugs, among which 105 are multi-hunk. We released these 249 new bugs in the CatenaD4J dataset we created, which is accessible at [10]. For each bug, we created a Defects4J-style command-line interface to download, compile, and test these bugs and obtain the bug information and metadata. We believe that CatenaD4J not only serves as a benchmark for tool evaluation but also provides an important basis to motivate the design and implementation of new multi-hunk repair techniques. While we have applied IBugFinder to Defects4J for indivisible bug detection and creation, it is worth noting that the approach is also applicable to other datasets [19, 43, 55], which we plan to investigate in future work.

Since no previous assessment of APR techniques has focused on indivisible bug repair, we evaluated existing techniques on all the indivisible multi-hunk bugs identified and created. For the original 118 Defects4J indivisible bugs, we compared the repair results of 14 existing techniques including 5 multi-hunk techniques [41, 57, 70, 80, 83] and 9 SOTA single-hunk techniques. For the other 105 indivisible bugs newly created, we ran 7 of the techniques to repair them. Our results showed that current APR techniques repaired only a small number of indivisible bugs. The best multi-hunk technique can repair at most 5 bugs and is outperformed by advanced single-hunk techniques, which can produce single-hunk and single-hunk-alike patches that are semantically equivalent to the multi-hunk patches provided by developers. We discussed the limitations of existing techniques and showed that current techniques repaired half as many indivisible multi-hunk bugs as divisible multi-hunk bugs.

To gain insights into effective multi-hunk patch generation, we conducted a study that comes in two parts. For the first part, to understand why an indivisible bug needs multiple fixes done at different locations and the role each fix plays to tackle the failure, we sampled 75 indivisible bugs from those found and created by IBugFinder. For each bug, we identified the hunks to repair, analyzed the fixes created for the hunks, and characterized the behavioral (semantic) relationships of the fixes. The result is a total of 8 behavioral relationships. We gave examples to describe the relationships and showed their frequencies.

A key implication is that the patch behavioral relationships suggest different repair strategies. We sketched the strategies, which we believe provide important insights into effective multi-hunk patch generation. For example, one relationship *original-and-new-problem-fix* suggests performing iterative patch generation to fix the original problem first, identify new problems raised (e.g., new exceptions thrown), and produce new patches to address them. This strategy can be used to tackle the Chart_15 bug that none of the multi-hunk approaches [39, 41, 46, 57, 70, 80, 83] have correctly repaired. For this bug, the failure is a null-pointer exception. By addressing the exception with a null-checker and having a problem detector that checks program execution, an APR technique would tell that a partial patch resolves the exception and allows the intended execution to continue. Later, it could build on the promising partial patch to deal with the new problem manifested as another null-pointer exception. Existing iterative approaches [80, 83] do not repair the bug, as they use weak guidance by checking for example the number of failing tests, which in this case does not change even with the promising (correct) partial repair.

For the second part of the study, we analyzed the behavioral relationships of patches generated for bugs correctly repaired by tools from our previous evaluation. We found that current tools can only handle simple multi-hunk bugs by generating single-hunk or single-hunk-alike patches, confirming their immaturity in patching different locations for complex bug repair.

The main contributions of this paper are as follows:

- An enumeration-based approach for detecting and creating indivisible multi-hunk bugs.
- An augmented dataset of indivisible bugs.
- An evaluation of existing repair techniques with the indivisible multi-hunk bugs.
- A study of patch behavioral relationships that provides guidance for multi-hunk bug repair.
- An analysis of the patch relationships for bugs correctly repaired by various tools.
- An artifact including the source code of IBugFinder, the new dataset, the study result, and the repair tools and test scripts used in the evaluation available at [6].

The rest of the paper is structured as follows. Section 2 presents IBugFinder along with the indivisible bug detection and creation experiment and the result. Section 3 presents our evaluation of existing APR techniques. Section 4 shows the study of multi-hunk fixes. Sections 5 and 6 discuss the threats to validity and the related work. Finally, Section 7 presents our conclusions and potential directions for future work.

2 INDIVISIBLE BUG DETECTION AND CREATION

We first use an example to intuitively describe what a divisible bug is and why it is not interesting. Next we give the definitions of divisible and indivisible bugs and other related terms. Then we show the algorithm that IBugFinder uses to determine bug divisibility and create indivisible bugs. Finally, we present the new bug dataset.

(a) First partial patch.

```
1 @Override
2 public OpenMapRealVector ebeDivide(RealVector v) {
3     checkVectorDimensions(v.getDimension());
4     OpenMapRealVector res = ...;
5   - Iterator iter = entries.iterator();
6   - while (iter.hasNext()) {
7   -     iter.advance();
8   -     res.setEntry(...);
9   - }
10  + final int n = getDimension();
11  + for (int i = 0; i < n; i++) {
12  +     res.setEntry(...);
13  + }
14    return res;
15 }
```

(b) Second partial patch.

```
1 @Override
2 public OpenMapRealVector ebeMultiply(RealVector v) {
3     checkVectorDimensions(v.getDimension());
4     OpenMapRealVector res = ...;
5     Iterator iter = entries.iterator();
6     while (iter.hasNext()) {
7         iter.advance();
8         res.setEntry(...);
9     }
10  + if (v.isNaN() || v.isInfinite()) {
11  +     final int n = getDimension();
12  +     for (int i = 0; i < n; i++) { ... }
13  + }
14    return res;
15 }
```

Fig. 1. Developer patch for the divisible multi-hunk bug Math_29.

2.1 Motivation

We take the divisible Math_29 bug from the Defects4J dataset for example. Figure 1 shows for this bug the developer patch consisting of two partial patches made at two locations, one (left figure) at method *ebeDivide* for performing vector-based element-by-element division and the other (right figure) at method *ebeMultiply* for element-by-element multiplication. The first partial patch ($ppat_1$) created for *ebeDivide* rewrites the original iterator-based loop (lines 6–9) scanning non-zero entries (enforced by line 7) into a new loop (lines 11–13) that allows the division of two zeros and uses `NaN` as the result. By skipping the zero entries, the original loop implicitly sets the division result of two zeros as 0, which is incorrect. The second partial patch ($ppat_2$) made at *ebeMultiply* is needed to give special treatment for elements whose values are `NaN` and positive and negative

infinities (lines 10–13). To fix Math_29, one has to tackle two failures (f_1 and f_2) exposed by two sets of failing test cases, one for testing *ebeDivide* (f_1) and the other for *ebeMultiply* (f_2). The two failures are independent. As long as $ppat_1$ is applied to *ebeDivide*, f_1 will be resolved, regardless of whether $ppat_2$ is made or not. Likewise, as long as $ppat_2$ is made for *ebeMultiply*, f_2 will disappear, irrespective of $ppat_1$.

The above analysis shows that Math_29 is actually a multi-failure bug. It is *divisible* and can be decomposed into two single-hunk bugs, which trigger $f1$ and $f2$ and can be addressed by two single-hunk patches $ppat_1$ and $ppat_2$ respectively. Repairing a divisible bug by addressing multiple failures simultaneously is uncommon, as in real debugging scenarios, a developer typically deals with one failure at a time [37, 51]. To repair Math_29, one would just look at one of f_1 and f_2, tackle it, and then work on the other. In this case, bug repair is performed as multiple single-hunk patch generation tasks. Repairing a divisible bug is not very interesting. As long as one location is correctly patched, one can see a failure resolved and continue fixing the other locations. This is not what one typically encounters for complex bug repair where a correct partial patch may not show any obvious progress.

2.2 Definitions

We next introduce the terms and give the definition of divisible and indivisible bugs.

Single-hunk and multi-hunk bugs. The repair of a *single-hunk* bug requires code edits done at a single location represented as one or a sequence of contiguous code lines referred to as a code hunk. Repairing a *multi-hunk* bug requires edits done at multiple locations that are not contiguous.

Patch and partial patches. A *patch* represents a set of code edits needed for bug repair. For a multi-hunk bug, a patch consists of multiple *partial patch*es, each done at a single location. For a single-hunk bug, because there is only one location to repair, there is only one partial patch, which is equivalent to the patch.

In practice, given the buggy and the fixed programs, we used the git diff utility [25] to perform a line-by-line comparison of two programs' source files to identify all the locations repaired and the corresponding partial patches.

Partial program and sub-program. Given a buggy program p and its fixed program p', one can obtain a set of partial patches $PAT = \{pat_1, pat_2, \ldots, pat_n\}$ by comparing the source files of p and p' (via for example a diff algorithm). We define a partially repaired program, or a *partial program*, p'' to be a program that one can get by applying to p the partial patches of PAT', which is a subset of PAT and is non-empty. When p'' is a partial program, we also say that p'' is a *sub-program* of p', the fully repaired program. Note that in our definition p and p' are not sub-programs of p'.

Positive and negative test cases. Given a buggy program p, the fixed program p', and a set of test cases T, we use PT to denote the subset of T that p passes and refer to PT as the *positive* test cases. Similarly, we use FT to denote the subset of T that p fails to pass and refer to FT as the *negative* test cases. When FT is not empty, we know p has a bug. Typically, PT is also not empty to allow the detection of invalid patches introducing regressions.

Divisible, indivisible, and isolated bugs. Given a buggy program p, the fixed program p', and a set of test cases T, which is a union of PT and FT (the positive and negative test cases), we say that p (or the bug) is *divisible*, or $div(p, p', PT, FT)$, if there is a partially repaired program p'', a sub-program of p', that can effectively resolve at least one of the failures by passing not only all the positive test cases but at least one negative test case. Formally, we have

$$div(p, p', PT, FT) \text{ iff } \exists p'' \in SUB(p').\ SomePass(p'', FT) \wedge AllPass(p'', PT),$$

where $SUB(p')$ is the set of all sub-programs of p', $SomePass(p'', FT)$ denotes that p'' passes some of the test cases in FT, and $AllPass(p'', PT)$ denotes that p'' passes all of the test cases in PT, Note

that in our definition, the divisibility of a bug is not determined by any partial program that fails for any of the positive test cases, indicating regression, regardless of whether it passes negative test cases or not. We say that p (or the bug) is *indivisible* if it is determined as not divisible.

When p is divisible, one may obtain an *isolated* bug exposed by the resolved failures. Let us say that p'' is a partial program that is found to have effectively resolved some failures by passing a subset of negative test cases $FT' \subseteq FT$ and all the positive test cases PT. In this case, one can identify an isolated bug of p exposed by FT'. For this bug, the fixed program is p'', and one can use T', a union of FT' and PT, as the test cases for bug-exposing and regression-testing. Note that an isolated bug can still be divisible. IBugFinder however only generates indivisible isolated bugs.

2.3 Algorithm

This section shows the algorithm of IBugFinder for divisible bug detection and isolation. The same algorithm can also detect indivisible bugs and create new indivisible bugs by isolating divisible bugs. We first provide the bug isolation principle, then describe the algorithm in detail, and finally discuss test minimization, a component of the algorithm that increases the opportunities for detecting divisible bugs.

Principle. As discussed in Section 2.2, a bug is divisible if there is a partial program that can effectively resolve some failures by passing all the positive test cases and at least one negative test case. Given a buggy program p and its fixed program p', IBugFinder finds and enumerates all partial programs of p' to determine if any of the programs can effectively resolve any of the failures. If there is a partial program p'' that effectively resolves a failure, the bug is determined as divisible. To further decide whether a new bug can be created based on p'', IBugFinder has to determine if p'' is still divisible, and if not, creates a new bug based on p, p'', and the failures resolved.

To understand this, consider Chart_18 for example. For this bug, we generated a partial program p_6 by having two of the partial patches pat_1 and pat_2 applied to the original program p. We found that p_6 passed two of the negative test cases t_4 and t_5 while also passing all the positive test cases, and thus determined Chart_18 as divisible. We however did not create a new indivisible bug based on p_6, as we found that two of its sub-programs p_2 (derived by having pat_1 applied to p) and p_3 (generated by having pat_2 to p) can pass t_5 and t_4 respectively. This means that p_6 is still divisible and can be further isolated into p_2 and p_3. For this reason, p_6 is omitted for new bug creation.

IBugFinder's approach could have be applied to the *original* test cases for indivisible bug detection and creation. This however would not be ideal, as a test case often contains more than one assertion, and IBugFinder would miss the opportunities to identify partial programs that fail to pass a negative test case as a whole but are able to pass an increasing number of assertions. To further increase the opportunity of divisible bug detection, IBugFinder performs test minimization to split a test case having multiple assertions into multiple test cases each with one assertion. We will detail the process at the end of this section.

Description. Algorithm 1 presents the detailed algorithm for divisible bug detection and isolation. IBugFinder is designed to be applied to an existing bug dataset. The main procedure *detectAndIsolate* takes as input the original buggy program pb, the fixed program pf (based on the developer patch), the set of test cases T, and two thresholds max_t and max_p used to avoid running for too long and processing too many partial programs raising memory issues. As output, the procedure generates a set of newly isolated bugs, each represented as a tuple $< pb, pat, FT_{pat}, PT >$ where pb is the original buggy program, FT_{pat} is a subset of negative tests proving that pb has a bug, pat is the patch for bug repair, and PT is the set of original positive test cases for regression testing. By applying pat to pb, one can get a fixed program pf' for this bug. Note that for an indivisible bug, FT_{pat} can contain multiple failures. They are however *homogeneous* in that none of the sub-programs of pf' can effectively resolve any of these failures by passing some tests in FT_{pat} without failing any

Algorithm 1: The algorithm of divisible multi-hunk bug detection and isolation.

```
1  Procedure detectAndIsolate(pb,pf,T,max_t,max_p):
2      S ← ∅;
3      PAT ← identifyPartialPatches(pb, pf) ;                    // Identified by the diff utility
4      if PAT.size() ≤ 1 then return S;
5      T ← minimize(T);
6      < FT, PT >← runTests(pb, T);
7      < PPATS_SET, exhaust >← getPowerSetOfPartialPatches(PAT, max_p);
8      PPATS ← rankIntoList(PPATS_SET);
9      C ← { };                                                                  // A working set
10     C.append(< ∅, "unused", FT >);
11     stime ← getCurrentTime();
12     for PPAT in PPATS do
13         if getCurrentTime() − stime ≥ max_t then
14             exhaust ← false;
15             Break;
16         end
17         pf′ ← applyPatchToProgram(pb, PPAT) ;                // Apply the partial patches to pb
18         if pf′ does not compile then
19             C.append(< PPAT, "unused", {} >);
20         else
21             < FT_PF′, PT_PF′ >← runTests(pf′, T);
22             if !isSubsetOf(FT_PF′, FT) then
23                 C.append(< PPAT, "unused", FT_PF′ >);
24             else
25                 FT_PASSED ← FT \ FT_PF′;
26                 FT_PASSED ← getFailingTestsPassedSolelyByTargetProgram(FT_PASSED, pf′, pb, C, FT);
27                 if FT_PASSED ≠ ∅ then
28                     S ← S ⋃{< pb, PPAT, FT_PASSED, PT >} ;                    // New bug found
29                     C.append(< PPAT, "used", FT_PF′ >);
30                 else
31                     C.append(< PPAT, "unused", FT_PF′ >;
32                 end
33             end
34         end
35     end
36     if S == ∅ and !exhaust then S ← { dummy_tuple} ;
37     return S;
38 end
39 Procedure getFailingTestsPassedSolelyByTargetProgram(FT_PASSED, pf′, pb, C, FT):
40     SUB_PF′ ← getSubPrograms(pf′) ;                           // Get all the sub-programs of pf′
41     for sub_pf′ in SUB_PF′ do
42         SUB_PPAT ← identifyPartialPatches(pb, sub_pf′) ;      // Identified by the diff utility
43         FT_SUBPF′ ← ∅;
44         for elem in C do
45             if elem.getLabel() == "used" and elem.getParPatches() == SUB_PPAT then
46                 FT_SUBPF′ ← FT_SUBPF′ ⋃ elem.getFailingTests();
47             end
48         end
49         FT_SUB_PASSED ← FT \ FT_SUBPF′;
50         FT_PASSED ← FT_PASSED \ FT_SUB_PASSED;
51     end
52     return FT_PASSED;
53 end
```

tests in PT. If the set of isolated bugs as output is empty, the original bug is indivisible. If the bug divisibility cannot be determined due to non-exhaustive search, the output is a set containing a dummy tuple.

IBugFinder starts with comparing pb and pf to identify the partial patches (line 3), minimizing the test cases (line 5), which we will detail later in this section, and identifying the negative and positive test cases (line 6). It simply skips single-hunk bugs which are by definition indivisible

(line 4). Based on the partial patches *PAT* identified, IBugFinder computes a powerset *PPATS_SET* where each element is an non-empty subset of *PAT* that can be used to generate a partial program, and it checks whether *PPATS_SET* exhaustively includes all subsets and saves the result in *exhaust* (line 7). IBugFinder ranks the elements of *PPATS_SET* by the number of partial patches contained in each element from low to high into a list *PPATS* (line 8). By examining the elements in *PPATS* from the beginning to the end, IBugFinder is guaranteed to look at s_1 before s_2, if the number of partial patches contained in s_1 is not greater. This ensures that IBugFinder always checks the sub-programs of p before p itself.

IBugFinder uses a working set *C* (line 9) to save some properties of partial programs that have been processed. Specifically, for each partial program pp, a tuple $< PPAT, l, FT\_PP >$ is saved in *C*, where *PPAT* is the partial patches used to generate pp, l is a label (either "used" or "unused") showing whether pp has been used to create a new bug or not, and *FT_PP* is the set of test cases that pp fails to pass. IBugFinder uses *C* to examine, for a partial program, each of its sub-programs that have been processed to determine if the partial program is divisible or not. *C* is initialized with a tuple of the original buggy program's properties (line 10).

A loop (lines 12–35) is used to enumerate the elements in *PPATS*, where each element *PPAT* is a subset of partial patches in *PAT*. The loop breaks if the time budget is used up, in which case, the search is not exhaustive (lines 13–16). For each *PPAT*, IBugFinder obtains the corresponding partial program pf' by applying *PPAT* to pb (line 17). It next compiles and tests pf' (lines 18–20). If pf' fails for all negative tests in *FT* or any positive tests not in *FT* (line 22), pf' does not reveal divisibility and is not used for new bug creation (line 23). Otherwise, pf' represents a partial program that effectively resolves some of the original failures indicating the bug is divisible. To further determine the eligibility of using pf' to create a new bug, IBugFinder enumerates all the sub-programs of pf' to compute *FT_PASSED*, a set of negative tests solely passed by pf' and not by any of its sub-programs (lines 25–26). If *FT_PASSED* is not empty, which means there are failures that can only be effectively resolved by pf' and not by its sub-programs, IBugFinder creates a new bug with the partial patches *PPAT* that result in pf' and the failures exposed by *FT_PASSED* (line 28). Otherwise, because there is no failure that cannot be addressed by its sub-programs, IBugFinder can use the sub-programs to create new bugs and thus skips pf'.

To compute *FT_PASSED*, IBugFinder initializes it as all the negative tests passed by pf' (line 25), and then invokes the procedure shown in lines 39–53 to enumerate the sub-programs and update *FT_PASSED*. IBugFinder gets all the sub-programs of pf' by generating the powerset of *PPAT*, partial patches that result in pf', applying each element in the powerset to pb to get a partially repaired program, and finally collecting all these programs. It enumerates each sub-program sub_pf', gets the partial patches by comparing it with pb via the diff utility, checks whether it has been used for bug creation, and if so, excludes from *FT_PASSED* the negative tests *FT_SUB_PASSED* passed by sub_pf' (lines 41–51).

Tests minimization. The idea of test minimization is to split a test case containing multiple assertions into multiple minimized test cases, each containing one assertion. One possible approach to achieving minimization would be (1) finding the assertions in the test case; (2) identifying, for each assertion, all the test code (from within the test case and from all the other test methods); and (3) creating new minimized test cases, each containing one of the assertions and all the test code that can affect the assertion. It is however challenging to automate and implement this approach, as for (2), the approach needs to deal with interprocedural analysis (maybe via a dynamic interprocedural-based slicing method) to effectively identify test code that can affect the assertion result while accounting for possibly complex language features. Moreover, step (3) can be complicated by the fact that the test code affecting an assertion can be from another assertion (that has side effect), in which case, one cannot easily keep the two assertions apart for creating minimized tests.

Table 1. Statistics of the original Defects4J multi-hunk bugs and the newly isolated bugs.

Project	Original (Multi-Hunk)				Isolated		
	#Total	#Div	#InDiv	#Ukn	#Total	#Multi	#Single
Chart	13	10	3	0	29	7	22
Closure	113	52	48	13	79	41	38
Lang	40	22	17	1	40	20	20
Math	70	38	26	6	58	20	38
Mockito	25	5	17	3	21	2	19
Time	20	12	7	1	22	15	7
All	281	139	118	24	249	105	144

What IBugFinder used for test minimization is an activation-based approach. It works by (1) identifying all the assertions in a test case and then (2) creating minimized test cases, each of which is identical to the original test case, except that only one assertion is kept active and the others are made inactive by the try-catch exception handling mechanism. Specifically, for (1), IBugFinder looks for statements in the test case that are calls to the JUnit assertion methods (such as *assertEquals*) that are defined in the JUnit assert package (*org.junit.Assert* and *junit.Assert*) and calls to other test methods that directly or indirectly (via nested method calls) use the JUnit assertion methods. It can also identify other assertion methods directly used in the test case but are not from JUnit library (e.g., *assertThat* from Hamcrest library) via string pattern-matching. For (2), to create a minimized test case, IBugFinder chooses one of the assertions from the original test case to be active and keeps the assertion as it is. For each of the other assertions s, IBugFinder replaces s with a new statement s', which wraps s with a try-catch structure as `try {s} catch(Throwable __SHOULD_BE_IGNORED){}`. In this way, when a minimized test case is run, an inactive assertion s can still be executed (so that the testing work done in s, which may affect the following assertions, can still be performed), but any failure triggered by the execution of s will be caught and ignored to ensure that the test case execution is not killed by the failure.

Note that IBugFinder does not have to minimize positive test cases. This is because a partial program that can be used to prove divisibility has to pass all the positive test cases including all the assertions. Minimized positive tests containing "separated" assertions do not create any new opportunities for divisible bug detection and isolation.

2.4 The Experiment of Indivisible Bug Detection and Creation and the Result

We implemented IBugFinder and applied it to all the multi-hunk bugs from six of the projects (Chart, Closure, Lang, Math, Mockito, and Time) in the Defects4J-v2.0 dataset that has been widely used for evaluating various APR techniques [26, 44, 88]. The detection-and-isolation process is automated. We note again that because IBugFinder has to enumerate and test all possible partial programs, the process can be costly. To ensure feasible bug detection and isolation, IBugFinder considered for each bug at most 2048 (*max_p* in Algorithm 1) partial programs to test to avoid out-of-memory errors and ran the isolation process for at most one hour (*max_t*). We found that a large fraction (~81%) of the bugs have no more than 5 hunks to repair, and IBugFinder has successfully determined the divisibility of 257 (over 91%) bugs. The experiment was performed on machines with 16 Intel i9-11900K 3.5 GHz CPUs, NVIDIA RTX A4000/5000 GPUs of 16/24 GB, and 128 GB RAM and run Ubuntu-20.04.

Table 1 presents the result. It shows for each project and for all the numbers of multi-hunk bugs (#Total under Original), the numbers of divisible (#Div) and indivisible (#InDiv) bugs detected, and the number of bugs whose divisibility is unknown (#Ukn) due to the incapability of partial program enumeration and bug deprecation (Time_21). It also shows the numbers of isolated bugs (#Total under Isolated) and multi-hunk (#Multi) and single-hunk (#Single) bugs. One can see that about a half (139/281=49.5%) of the original Defects4J bugs are divisible (column #Div). IBugFinder created

249 new bugs, among which 105 bugs are multi-hunk (column #Multi). It failed to determine the divisibility of 23 (or 8.1%) bugs. 22 of them are from three projects (Closure, Math, and Mockito) for which testing runs slow. These are the bugs with many hunks to repair (median is 12).

3 TOOL EVALUATION WITH INDIVISIBLE BUGS

We evaluated state-of-the-art APR techniques with the 118 original and 105 isolated bugs to investigate the effectiveness of existing techniques in repairing indivisible multi-hunk bugs. The section presents the experiment and the results.

3.1 Experiment Setup

For the 118 original Defects4J bugs, we looked for the repair results reported in the evaluation of 14 existing APR techniques including the 5 multi-hunk-oriented techniques ARJA-e [84], DEAR [41], Hercules [57], ITER [80], and VarFix [70] and 9 other SOTA techniques that are LLM-based (AlphaRepair [72]), NMT-based (e.g., Recoder [91]), search-based (e.g., SimFix [31]), constraint-based (e.g., Nopol [77]), and pattern-based (e.g., TBar [42]). We did not include other tools for example ChatRepair [73], Repilot [68], KNOD [33], TENURE [48], and MCRepair [36], as for these tools, we did not find the ids of the bugs reported as correctly or plausibly repaired with realistic fault localization from either the paper or the released artifact. We also excluded Angelix [46], as the tool [4] was implemented to repair C programs and cannot be directly applied to a Java benchmark.

For the 105 isolated bugs, because they were not used to evaluate any APR techniques, we conducted a repair experiment by applying 7 SOTA techniques, which are ARJA-e, Hercules, ITER, Recoder, SimFix, TBar, and AlphaRepair to these bugs for repair. Among these techniques, ITER, ARJA-e, and Hercules are SOTA learning-based and search-based multi-hunk-oriented techniques. Recoder, SimFix, and TBar are SOTA NMT-based, search-based, and pattern-based single-hunk techniques. AlphaRepair is a SOTA LLM-based technique that performs cloze-style single-hunk repair. We note that we did include single-hunk tools because they may produce single-hunk patches that are semantically equivalent to the developer patch addressing multiple hunks.

We contacted the authors of ITER to get the tool (which was not publicly available by our experiment time but is now released at [29]). For AlphaRepair, ARJA-e, Recoder, SimFix, and TBar, we used the tools provided by the authors at [2, 5, 53, 58, 64]. Because the Hercules tool is not available, we implemented it according to the technical description provided in [57]. The tool uses an Ochiai-based SBFL method [28] to identify faulty locations, the Soot framework [62] to identify code semantic context via data-flow analysis, and the ODS tool [15, 50] (rather than ELIXIR's ranking tool [56], which is not available) for patch prioritization. For a fair comparison with others, the tool does not leverage the revision history for repair.

Note that we did not consider all the 14 tools to repair the 105 isolated bugs. We excluded DEAR [16] and VarFix [66] due to usability issues. We omitted MCRepair, as the tool is not available [45]. For single-hunk tools, we used Recoder and excluded DLFix [17, 40] and RewardRepair [54, 79], as they are all NMT-based approaches and the latter two techniques were not more effective than Recoder based on both our result (Table 2) and the previous evaluation [79, 91]. We also excluded Nopol, which underperformed the other techniques by a large margin. We did not include Tare and TransplantFix as we failed to use the tools due to technical problems.

For the experiment of repairing 105 isolated bugs, what we provided for a tool is the bug isolation result, which is a four-element tuple for a bug (Algorithm 1, line 25), except the patch. Specifically, we provided the original program, the original passing tests, and the bug-exposing failing tests identified and minimized by IBugFinder. The test minimization, because of using an activation mechanism to create new tests, can incur additional overhead. The overhead is however empirically insignificant, as the minimization only affects a small number of test cases. For an isolated bug used

Table 2. Repair results for the 118 indivisible multi-hunk bugs presented as x/y where x and y are the numbers of bugs for which correct patches and plausible patches were generated respectively. Hyphen symbols denote that the results are not made available. We used the results from the previous evaluation of the tools.

Project	Multi-Hunk-Oriented Techniques					Single-Hunk-Oriented Techniques								
	ARJA-e	DEAR	Hercules	ITER	VarFix	AlphaRepair	DLFix	Nopol*	Recoder	RewardRepair	SimFix	Tare	TBar	TransplantFix
Chart	1/3	0/-	0/0	0/1	0/0	1/-	0/-	0/2	1/-	0/-	0/0	2/3	1/2	0/-
Closure	0/0	1/-	0/0	0/1	0/4	3/-	0/-	0/17	3/-	0/-	0/0	2/8	2/2	1/-
Lang	1/6	0/-	0/0	2/2	0/0	1/-	0/-	0/0	1/-	0/-	0/0	2/2	0/0	0/-
Math	2/9	1/-	2/2	1/2	0/4	1/-	0/-	0/4	0/-	0/-	0/2	1/9	1/6	1/-
Mockito	0/0	0/-	0/0	0/0	0/0	0/-	0/-	0/0	0/-	0/-	0/0	0/0	0/0	1/-
Time	1/3	1/-	0/0	0/0	0/0	1/-	1/-	0/3	1/-	0/-	1/1	1/1	1/1	0/-
Total	5/21	3/-	2/2	3/6	0/8	7/-	1/-	0/26	6/-	0/-	1/3	8/23	5/11	3/-

*We referred to [18] to find the plausible patches and determined the correctness (not shown in [18]) for relevant bugs.

Table 3. Repair results for the 105 isolated indivisible multi-hunk bugs presented as x/y where x and y are the numbers of bugs for which correct patches and plausible patches were generated respectively. We ran the tools to get the results.

Project	Multi-Hunk-Oriented Techniques			Single-Hunk-Oriented Techniques			
	ARJA-e	Hercules	ITER	AlphaRepair	Recoder	SimFix	TBar
Chart	0/5	1/3	0/1	1/3	1/3	1/2	1/3
Closure	0/0	1/3	1/4	2/5	2/5	1/1	3/5
Lang	0/2	0/7	0/1	1/4	0/2	0/0	0/6
Math	0/6	0/4	0/3	0/0	0/5	0/0	0/4
Mockito	0/0	0/0	0/0	0/0	0/0	0/0	0/0
Time	0/0	0/2	1/4	3/6	0/2	0/0	0/2
Total	0/13	2/19	2/13	7/18	3/17	2/3	4/20

in the experiment, there are only a few minimized tests – the average and median numbers are 7.1 and 2. We did not see any noticeable slowdown or validity issues when using the minimized tests.

The experiment was performed on the same machines we used to do bug isolation. Following the standard experiment settings [41, 57, 80], we ran each tool to repair each bug for a maximum of 5 hours and reported the numbers of bugs for which plausible and correct patches were generated. We adjusted the default settings and configurations of the tools when necessary to ensure a fair comparison and an optimized use of GPUs (for AlphaRepair and Recoder). More details can be found at the tool usage repositories accessible from [7]. A patch is plausible if the patched program passes all the test cases. Following the common practice for evaluation [41, 57, 80], we considered a patch as correct if it is identical or semantically equivalent to the developer patch. To determine correctness, two of the authors have independently examined the patches and held discussions when necessary to reach a consensus. We failed to use ITER and ARJA-e to repair the Mockito bugs, as according to [80, 83], the tools do not support repairing these bugs. We also excluded the Closure bugs for testing ARJA-e, as according to [83], the tool does not support repairing them due to the customized non-JUnit testing format used for these bugs. The machine time used to run the repair experiment is over 2K hours (83 days). The experiment actually ran for more than 23 days.

3.2 Result

Tables 2 and 3 present the results for the 118 original and the 105 isolated multi-hunk bugs respectively. According to our results, the best techniques correctly repaired no more than 8 original bugs and 7 isolated bugs. This shows that the repair abilities of existing techniques in handing indivisible bugs are not strong. Table 2 (column Multi-Hunk-Oriented Techniques, last row) shows that a multi-hunk tool correctly repaired at most 5 bugs. Together, these tools repaired 9 bugs, among which 6 bugs were done with single-hunk-alike patches. This implies that their ability in tackling more complex bugs is weak. We had similar observations for the 105 bugs.

Current multi-hunk techniques have various limitations. The search space for multi-hunk bug repair is often enormous, and ARJA-e's search strategy, which uses a fitness function that measures

patch size and failure rate, cannot effectively direct the exploration to find correct patches. We note that patch size does not necessarily imply correctness and that for a typical indivisible bug triggering a single failure, partial patches cannot lead to any failure resolved. For a similar reason, ITER, which detects failure rate decrease to guide iterative repair is also not good at dealing with indivisible bugs. Because again the search space is huge, DEAR, which uses a learning method to perform simultaneous fault localization and code editing, has limited effectiveness. Hercules identifies evolutionary siblings to narrow the search space. However, as we will show in Section 4.2, for indivisible bugs, correct partial patches identified as evolutionary siblings are not common, which weakens the applicability of Hercules. Finally, VarFix is not shown effective, as we believe its variational-execution-based exploration has scalability issues.

Tables 2 and 3 also present the repair results of the single-hunk tools. AlphaRepair and Tare, repaired more bugs than the multi-hunk tools. We note that this is possible. As we will show in Section 4.2, many multi-hunk bugs can be repaired with single-hunk-alike patches that, for example, add an if-check for a block of code in two hunks. It can also be the case that for some bugs there is a single-hunk patch that is semantically equivalent to the developer multi-hunk patch.

For the 118 bugs, we investigated how existing tools performed with perfect fault localization (PFL), which assumes that all the repair locations are known. With PFL, single-hunk tools achieved only slightly better result, and the best tool AlphaRepair correctly repaired 10 bugs. PFL is a strong assumption for multi-hunk repair, and for only one multi-hunk tool MCRepair, we were able to find the repair result for the 118 bugs. The tool correctly repaired 9 bugs. Because it does not support non-PFL, we did not include the tool in Table 2.

We also compared how a repair tool deals with divisible and indivisible multi-hunk bugs. We did this by collecting the repair result for all the 139 bugs identified as divisible and finding the numbers of the bugs correctly repaired by the 14 tools from the previous evaluation. The average numbers of divisible and indivisible Defects4J bugs repaired by the tools are 6.29 and 3.14. This shows that existing tools can repair twice as many divisible bugs as indivisible bugs.

4 STUDYING AND UNDERSTANDING MULTI-HUNK FIXES

This section presents our study of multi-hunk fixes. It comes in two parts. For the first part, we show the 8 behavioral relationships identified for the developer patches for a set of sampled bugs. For the second part, we show the relationships for patches of bugs correctly repaired by existing tools. In this section, we first explain how we did the study, then present the results and findings of both parts, and finally discuss the implications.

4.1 Methodology

For the first part of the study, to identify partial-patch behavioral relationships, we analyzed the developer patches for a variety of multi-hunk indivisible bugs sampled from the Defects4J dataset. The sampling process is as follows. Because different Defects4J projects have different numbers of bugs, to ensure diversity, we randomly selected, for each project except Chart, 10 original Defects4J bugs that are either (a) indivisible or (b) divisible but have isolated indivisible multi-hunk bugs. For (a), we directly used the bug for study; and for (b), we included all the isolated indivisible multi-hunk bugs. The Chart project has only 7 original bugs that are within our scope (specified by a and b), and we included all the indivisible bugs. In this way, we obtained a total of 75 indivisible multi-hunk bugs.

For each sampled bug, we used the following process to identify the behavioral relationships contained in the developer patch. We first understood the buggy and expected behaviors of the program and then analyzed the various behaviors with partial patches constructed in different combinations. This analysis often involved looking into the program states and execution. It was

largely manual, and the only tool we used is the Eclipse IDE debugger [20]. To make sure that we had a deep understanding of the program behavior, it was often the case that we used the debugging mode of the IDE with different breakpoints set up and ran the program stepwise in multiple rounds. It was based on the behavioral analyses of different bugs we identified the types of behavioral relationships. One of the authors was responsible for the relationship identification and another author checked the result.

For the second part of the study, we identified all the 28 bugs correctly repaired by all the tools reported in Tables 2 and 3. Then for each bug, we followed the above process to analyze its developer patch and identify the behavioral relationships of the patch. We also gathered from either the previous evaluation result or our experiment the correct patches generated by the tools and identified the relationships that exist in those tool-made patches. We did not analyze the patches for the 118 bugs generated by three techniques, DEAR, Hercules, and VarFix, since we did not find the correct patches from these tools.

4.2 Analysis of the Developer Patches for Sampled Bugs

As a main result of the study, we identified 8 behavioral relationships of partial patches. Below we first describe the relationships with examples and then provide our analysis and other results.

Def-and-use (DU). Partial patches with this relationship add the definition of variables, fields, packages, or methods and later use what has been defined for bug repair. A typical example is the bug Time_12b1 (isolated from Time_12) whose developer patch consists of two partial patches used to deal with a failure of incorrectly computing the year to initialize a local date time object, as shown in Figure 2a. The failure was due to the ignorance of the era (either AD or BC) to decide how to compute the year. For repair, one partial patch (line 1) adds the definition of era, and the other uses it to compute different year values.

One-action (OA). Partial patches with this relationship can be done in one repair action or operation. Figure 2b presents an example showing the patch for Chart_26. This patch consists of two partial patches (lines 1 and 4). They represent one repair action, which is to add an if-statement wrapping a hunk of code adding labels for chart drawing. Note that a wrapping action is widely used as a repair pattern for patch generation [42]. All partial patches that we identified as having this relationship can be generated with a wrapping action (by wrapping a hunk of code with for example an if, while, or try-catch statement). Depending on the repair actions used, however, one can have different notions of one-action relationships. Note that a wrapping OA patch can become single-hunk if any code indentation changes for the wrapped code are also included and considered.

Related-issue-fix (RIF). This relationship indicates that the partial patches are used to address related issues that arise in different locations. The issues are related due to similar problems (e.g., missing null checkers for the same variable) or related concept (e.g., overriding the *equals* and *hashCode* methods). Figure 2c shows three partial patches having the RIF relationship for repairing Chart_15. The partial patches are used to address related issues having the same problem, which is not checking the nulless of `dataset` (the pie chart dataset). Note that in this case the last two partial patches (lines 6 and 8) also have an OA relationship.

Different-issue-fix (DIF). This relationship indicates that the partial patches address different issues arising from different program parts that may implement the same functionality. Figure 2d shows as an example the isolated indivisible bug Lang_62b2 and two partial patches having this relationship. These partial patches are created to address different issues. The first ensures that one can get the correct entity value (e.g., *12345678*) from the entity content (e.g., *#x12345678*) by inserting a break at line 5 to avoid a mis-assignment at line 7. The second partial patch ensures that the entity value is not too big to process. These issues arise from a run that tests the *unescape* functionality of HTML/XML string manipulation.

(a) DU example.

```
+int era = calendar.get(Calendar.ERA);
int yearOfEra = calendar.get(Calendar.YEAR);
return new LocalDateTime(
-   yearOfEra,
+   (era == GregorianCalendar.AD ? yearOfEra : 1 -
↪  yearOfEra), ...);
```

(b) OA example.

```
+if (owner != null) {
+    EntityCollection entities =
↪   owner.getEntityCollection();
+    if (entities != null) { ... }
+}
```

(c) RIF example.

```
public double getMaximumExplodePercent() {
  +if (this.dataset == null) { return 0.0; }
  double result = 0.0;  ...
}
public PiePlotState initialise(...) { ...
  +if (this.dataset != null) {
  +   state.setTotal(...);
  +} ...
}
```

(d) DIF example.

```
public void unescape(...) { ...
  try { switch (isHexChar) {
      case 'X' : case 'x' : {
          entityValue = Integer.parseInt(...);
          +break; }
      default : {
          entityValue = Integer.parseInt(...);
      }}
      +if (entityValue > 0xFFFF)  entityValue = -1;
  } catch (NumberFormatException e) {} ...
}
```

(e) EOH example.

```
-if (searchChars.indexOf(ch) < 0) {
+boolean chFound = ...;
+if (i + 1 < strLen && ...) {
+   char ch2 = str.charAt(i + 1);
+   if (chFound && searchChars.indexOf(ch2) < 0) {
        return i;
+   }} else {
+      if (!chFound) { return i; }
}
```

(f) SU example.

```
public ClassLoaderAwareObjectInputStream(...) {
    +primitiveTypes.put(``byte'', byte.class); ...
}
protected Class<?> resolveClass(...) {
    +try {
        return Class.forName(name, false, ...);
    +catch (ClassNotFoundException cnfe) {
    +Class cls = primitiveTypes.get(name); ... }}
}
```

(g) ONPF example.

```
-if (minutesOffset < 0 || minutesOffset > 59) {
+if (minutesOffset < -59 || minutesOffset > 59) {
        throw new IllegalArgumentException(...); }
+if (hoursOffset > 0 && minutesOffset < 0) {
+   throw new IllegalArgumentException(...); }
```

(h) FU example.

```
+if (this.allowDuplicateXValues) {
+   add(x, y); return null; }
XYDataItem overwritten = null;
int index = indexOf(x);
-if (index >= 0 && !this.allowDuplicateXValues) {
+if (index >= 0) {
    ... }
```

Fig. 2. Examples of the fix relationships.

Essentially-one-hunk (EOH). Partial patches with this relationship can be considered as a single-hunk patch. The reasons can be that there is only one partial patch that is semantically needed and the others are created only to improve readability, that the locations addressed by the partial patches are physically close and semantically dependent and any partially repaired program derived from these patches cannot compile, or that the locations are semantically contiguous (connected with for example comments). Figure 2e shows an EOH patch for an indivisible bug derived from Lang_30. The repair locations for the two partial patches (lines 1–5 and lines 7–8) are close and only separated by a single return statement (line 6). In this case, applying either of the partial patches will result in a repaired program that does not compile. EOH is different from OA as the patch is not done with one action.

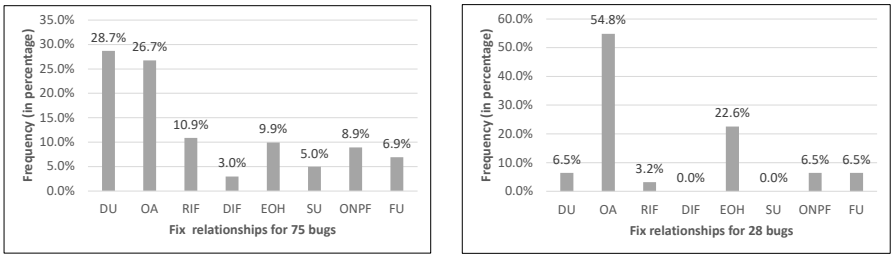

Fig. 3. Frequencies of the fix relationships identified based on the developer patches for the 75 sampled bugs (left) and the 28 bugs repaired by various tools (right).

Setup-and-use (SU). Some of the partial patches are created to do some setup work by updating a variable, field, or method. The others use what has been updated for fixing. For example, the partial patch shown in line 2 of Figure 2f updates the map `primitiveType` by adding the classes for each of the 8 primitive data types of Java along with `void`. This map is further used by the second and third partial patches (lines 5, 7, and 8) to resolve the class for the primitives.

Original-and-new-problem-fix (ONPF): Some of the partial patches can fix the original problem and resolve the original failure. Unfortunately, they also raise new problems triggering new failures, which can be tackled by other partial patches. Figure 2g presents an example of partial patches with this relationship to address Time_8b1. The original failure is due to the inability of the program in handling negative minute offsets to compute time, and the first partial patch (lines 1 and 2) can fix this problem. Unfortunately, the partially repaired program has a new problem, as it implicitly allows any invalid inputs with positive hour offsets and negative minute offsets to be processed. This problem is newly raised because inputs with negative minute offsets are not previously allowed. To fix this problem, the second partial patch adds an if-statement (lines 4 and 5) to check for the invalid cases. Note that ONPF is different from DIF and RIF in that the original buggy program does not have the new problem, which is only triggered by the partial patches used to fix the original problem.

Fix-and-undo (FU): In this relationship, some of the partial patches serve as the primary changes to correct the misbehavior of the program. These partial patches alone are however not enough. Other partial patches are needed to undo the negative influence brought by the previous changes. Lines 1 and 2 of Figure 2h present a partial patch used to correct a misbehavior of the program that triggers an index-out-of-bounds exception when adding two-dimensional coordinate points with duplicated x-values. To fix the problem, the partial patch inserts a new if-statement that directly adds the coordinate if duplicated x-values are allowed. Because the original program also accounts for duplicated x-values (line 5), an additional patch (lines 5 and 6) is needed to perform the undo job, by removing the unnecessary check for cleanup. Note that for FU, all partial patches are needed to address the (original) problem, which makes it different from ONPF.

The left part of Figure 3 presents in percentage the frequencies of the 8 relationships we identified for the 75 bugs. It is worth noting that for a complex bug having many partial patches, there can be more than one relationship. One can see that some of the relationships occur more frequently than others. In particular, DU and OA are the most frequent relationships with the sum of their frequencies being 55.4%. This means that fixing a complex bug often involves defining new variables, fields, or methods or using them from other packages. It also shows that over 26% of the partial patches can be done with single repair actions. In addition to DU and OA, RIF and EOH also have high frequencies (10.9% and 9.9%), followed by ONPF, FU, and SU. DIF has the lowest frequency, which suggests that one can often fix one or a set of related issues to address a single failure.

In addition to analyzing the behavioral relationships, we also investigated the syntactic relationship of partial patches. The way we did this is by using the Hercules tool we implemented to compare the context similarity of the repair locations addressed by the partial patches to decide whether they can be considered as evolutionary siblings for simultaneous correction. Our result shows that for only 9 (12%) of the bugs, the repair locations can all be considered as evolutionary siblings. The developer patches have the OA relationship for 6 bugs, EOH relationship for two bugs, and the RIF relationship for one bug. This result implies that, for indivisible bug repair, one should not generally assume that the repair locations are syntactically similar or related.

4.3 Analysis of the Patches for Tool-Repaired Bugs

We additionally analyzed the behavioral relationships of the developer patches and tool-made patches for all the 28 bugs correctly repaired by the 14 tools. We identified six relationships that exist in the developer patches and presented their frequencies in the right part of Figure 3. The result shows that current techniques do not go far beyond generating single-hunk-alike patches: About 77% of the patch relationships are OA and EOH, and an advanced single-hunk technique can already generate patches with these relationships.

For the 28 bugs, the tools generated 116 correct patches in total. 102 (88%) of the patches are strictly single-hunk, suggesting that many multi-hunk bugs can be addressed by single-hunk repairs. 11 of the remaining patches have the OA relationship and 1 patch has the EOH relationship. These are all generated by single-hunk techniques. Finally, ARJA-e generated one ONPF patch for Lang_34, and ITER generated one for Math_46. For these bugs, the partial patches tackling the original problem and causing new problems can be easily detected by the test-passing result.

4.4 Implications on Multi-Hunk Repair

Our study has led to important implications for multi-hunk bug repair.

Tailored repair strategies. A key implication is that one can develop specialized repair strategies based on the relationships and then apply these strategies to obtain guided exploration for effective multi-hunk repair. As we will show later in this section, specialized strategies can significantly reduce the search space, much like how patterns enable a restricted search space for single-hunk repair. We next sketch the possible ways to realize the specialized strategies.

OA and EOH suggest performing one-action and one-hunk operations to deal with multiple edits. Current advanced single-hunk approaches, especially those based on deep learning and LLMs, are already capable of generating patches supporting one-action repairs by for example adding an if-condition wrapping a statement and fairly sophisticated repairs handling the EOH cases.

DU and SU patches are specialized in that they require the import of a class or the definition/setup of a variable, field, or method and later use what has been introduced or set up for bug correction. We noticed that for a multi-hunk patch involving the definition/setup and use of local variables, the repair locations are often physically close. For 85% of the local-variable-related DU and SU patches we analyzed in the study, the repair locations have a distance of no more than five lines of code. In other cases, the repair locations are far away, but there is often a chance to refactor them to be close or even contiguous. For example, a variable definition can be refactored to be done at a location closer to the use of the variable, and a method can be inlined. This suggests that one may start with a single-hunk or single-area repair to do the right functionality correction and later refactor the repair into multi-hunk partial patches that are possibly far away from one another to improve readability and performance. This strategy can be realized with the help of LLM-based repair that handles a local code area [71] together with an automated refactoring approach [65].

It is also possible to generate DU and SU patches iteratively by first leveraging single-hunk approaches to generate a key fix template, which contains for example undefined or uninitialized

variables and then performing initialization or setup for concretization and completion. Existing intelligent code synthesis [8, 27] and generation [21, 38] approaches can automate the initialization or setup part. With the prefix and suffix contexts included for a target location expecting a variable definition where the prefix shows what code does up to the target location and the suffix contains the use of the undefined variable, a code synthesis/generation approach can generate a definition statement that fits the context. For example, with a prompt including the prefix of a `containsAny` method that checks whether a `CharSequence` object contains any characters of a given array, the suffix containing the code that handles supplementary characters, and an explicit request of generating variable definitions, ChatGPT (with the GPT-3.5 Turbo model) [11] found the correct definitions of `csLastIndex` and `searchLastIndex` to repair the Lang_31 bug.

RIF suggests that related issues can be identified and dealt with at multiple locations. A large fraction of the issues (over 50% of the cases we analyzed) are related in that they are caused by a similar problem and can be addressed with similar patches. The Hercules' approach [57] for detecting and evolving code siblings is promising to tackle this type of similar issues. Other RIF issues can be addressed with patches constructed with the assistance of refactoring (e.g., to implement fixes done in different methods with a calling relationship) and the incorporation of domain knowledge (e.g., to update both the `equals` and `hashCode` methods).

Unlike RIF, DIF needs to tackle distinct issues, and a DIF-patch can be constructed via iterative repair. The strategy iteratively looks for a target location to repair via fault localization, generates partial patches for that location with for example single-hunk approaches, and verifies that there has been some issue validly resolved and goes on addressing the remaining issues. A key challenge for DIF-based repair lies in the identification of promising patches implying positive progress towards issue-resolving. In general, repair progress detection is challenging. A promising solution is to use learning-based methods by for example gathering a set of labeled bug-fix pairs, extracting a variety of program syntax and semantic features, and training a classifier for progress inference.

ONPF implies that a partial patch fixes the original problem but unfortunately also causes new problems. To deal with ONPF, one can use a similar iterative approach that employs single-hunk approaches to construct patches fixing the original problem, detects any new problem exposed by the test execution result or indicated by a trained classifier, and then addresses the new problem.

Finally, FU consists of a primary fix that almost does the right correction and a secondary fix that cancels out any semantic redundancies. An FU patch can be realized via iterative repair as well by first creating the primary fix with the help of single-hunk-oriented repair and then searching via dependency analysis for problematic locations causing semantic redundancies and fixing those locations with for example pre-defined patterns designed for code cleanup.

Complexity analysis. The advantage of using specialized repair strategies is that they can greatly narrow down the search space for addressing a specific type of errors. To see this, first consider a full iterative approach $tech_f$ that iterates k times to find a k-hunk correct patch. In each iteration, it focuses on generating a one-hunk partial patch. Because it does not know where the bug is, it performs fault localization to identify n candidate locations for independent, single-hunk repair in each iteration. Suppose that $tech_f$ generates m single-hunk partial patches for one location. This results in a total of mn partial patches generated at the end of the iteration. With all partial patches chained together, in combination, the search space of $tech_f$ is $O((mn)^k)$. A specialized repair technique adopts a tailored strategy to avoid generating partial patches in combination and thus significantly reduces the search space. Consider a specialized approach $tech_s$ for multi-hunk repair. $tech_s$ first performs fault localization to identify n candidate locations for independent single-hunk repair. For each location, it generates m partial patches, and based on each partial patch, it can proactively produce the remaining $(k-1)$-hunk partial patches with an effective

specialized repair strategy. Without generating patches in combination, the specialized strategy can result in $O(\alpha(k-1)m)$ patches, where α approximates the number of partial patches generated for each hunk with the specialized strategy. With all n locations considered, the search space of $tech_s$ is $O(\alpha(k-1)mn)$, which is much smaller than the exponential $O((mn)^k)$.

The above analysis shows that a single specialized patch generator can result in reduced search space. To have all kinds of specialized generators used for multi-hunk repair, a simple strategy would be running all the generators in parallel and gathering all the patches after they are generated. In this case the search space is still polynomial. More intelligent ways of assembling the specialized generators include training a multi-classifier to select the most suitable specialized generators [47] and using an ensemble approach based on heuristics [90] for generator prioritization. The generators assembled this way can result in a smaller space. Of course, a complex repair may involve multiple patch relationships, which means the patches generated by the specialized generators may still need to be combined. We nevertheless believe that evolutionary algorithms can provide promising solutions to patch combination to avoid search space explosion. Finally, we note that even realizing a few strategies targeting the most frequent relationships such as DU, OA, and RIF can be beneficial.

Improved single-hunk patch generation. Our statistics of behavioral relationships showed that a significant fraction (over 36%) of the partial patches contain the OA and EOH relationships. We also showed that the frequent DU patches can emerge as a single-hunk patch first and then be refactored into multi-hunk and that an iterative approach that produces single-hunk partial patches in multiple steps is promising to construct effective multi-hunk patches. These insights imply that a successful multi-hunk repair hinges on effective single-hunk patch construction. To benefit indivisible bug repair, single-hunk patch generation should focus on improving the ability of producing wrapping-based OA patches and generating multi-line edits.

Simultaneous vs. iterative repair. To deal with multi-hunk bugs, simultaneous and iterative methods have been both proposed by previous work [41, 57, 80]. The former suggests identifying all the locations that are failure-responsible and then addressing them all at once for patch generation, whereas the latter uses an iterative method focusing on localizing and repairing only one location at a time. Our analysis suggested opportunities for both methods to work. To handle a complex bug, however, we believe that it is extremely challenging to have all the failure-responsible locations repaired at once and that an iterative method, which follows the divide-and-conquer discipline to perform location-wise repair, is more promising.

5 THREATS TO VALIDITY

We discuss the potential external, internal, and contruct threats to validity.

External threats. Our study and evaluation are based on the multi-hunk bugs from six projects in the Defects4J dataset. The results and conclusions may not generalize to other datasets. We nevertheless think this is a reasonable first exploration, as the bugs from the six projects have been broadly used for evaluating existing approaches. We used the Defects4J-v2.0 framework, as the IBugFinder tool we developed and some of the APR tools (e.g., Recoder) we tested require Java-8 or higher versions, which is not compatible with the Defects4J-v1.2 framework. For this reason, the repair result for the 118 bugs collected from previous evaluation, which is based on the Defects4J-v1.2 framework, poses a potential threat. We nevertheless note that the threat is minimal, as by checking the commit ids, we found that only three (<1%) bugs are inconsistent across the two versions, and only one was included in our evaluation and was not repaired by any tools. Finally, due to the high cost of program understanding and semantic analysis (Section 4), we used a sample of 75 bugs to identify the 8 behavioral relationships, and the results and conclusions of the study may not generalize. We note that the sample, albeit incomplete, is representative. It has already covered about 1/3 of the 223 (118+105) indivisible bugs, and we drew these bugs from all the

projects to increase diversity. As we analyzed more and more bugs, we had increasing confidence that there would be diminishing gain of finding new relationships and that the primary conclusion about the relationship frequency would not change. To test this, we additionally sampled 10 bugs for analysis and found that the patches are all covered by the 8 relationships and that our primary conclusion, which implies the dominance of DU and OA patches, still holds.

Internal threats. A first threat is that the bug divisibility result may be affected by any implementation errors of IBugFinder. We mitigated this threat by carefully testing IBugFinder and checking the result. We also made the tool and the result available for public review and extensions. To assess the effectiveness of existing techniques, we ran 7 APR tools to repair 105 isolated bugs. Because the Hercules tool is not available, we faithfully implemented the tool based on the technical description in [57]. For patch assessment, we manually determined the correctness. To reduce bias, two of the authors independently checked the patches and held discussions when necessary to reach a consensus.

Construct threats. First, we used a line-based diff utility to identify partial patches. A line-based method has its inherent limitation in capturing syntactic code changes, and one could alternatively use an AST-based diff utility and create a different notion of code hunks based on the changes of nodes. We note that certain patch relationships such as OA and EOH defined in this paper may not exist in a different context. For example, a common line-based OA patch that adds an if-wrap may become a single-hunk AST-based patch, as it adds only one node to the tree. Second, IBugFinder does not consider any partial patches affecting a code area that is smaller than a hunk for divisible bug detection. It can however occur that a single-hunk patch may contain fixes for multiple bugs. Examining smaller-than-a-hunk partial patches enables deeper exploration for divisible bug detection. Third, the IBugFinder algorithm has efficiency limitation and uses two thresholds to avoid excessive running time and memory use. The efficiency can be improved by using for example static analysis to rule out semantically invalid sub-programs and identify patch-affecting test cases for testing. Finally, the test minimization algorithm, because of producing test cases with inactive assertions, can incur additional testing overhead and affect code readability and understandability. Future research may explore cleanup methods to mitigate this effect.

6 RELATED WORK

Automated program repair (APR) aims to make debugging easier by automatically correcting bugs. A group of APR techniques focus on handling complex bugs by generating patches addressing different program locations via evolutionary search [39, 83–85], symbolic and variational executions [46, 70], evolutionary siblings identification and update [57], deep learning [41], iterative repair [80], and patch combination [36]. The majority of existing APR techniques are single-hunk-oriented. They produce patches repairing a single program location based on patterns for good patch construction (e.g., [42]) and bad patch identification [63], syntactic and semantic similarities (e.g., [1, 31, 69, 74, 78]), code synthesis (e.g., [23, 46, 76, 77]), program state analysis [13], bytecode mutation [24], and machine learning [88]. Recently, various LLM-based techniques have been proposed [22, 32, 71, 73]. They have demonstrated superior abilities in repairing a buggy line, hunk, or even a method [71].

The Defects4J benchmark [34] is widely used for evaluating APR techniques but contains a significant fraction of divisible multi-hunk bugs triggering multiple failures. To provide a better basis for multi-hunk repair research, we developed IBugFinder and applied it to the benchmark to detect existing divisible bugs and further isolate them to create new indivisible bugs. IBugFinder is related to but is different from the various methods proposed to facilitate parallel debugging [61, 86], improve fault localization [87] and prediction [14, 67], and identify single-fault fixes [51] in that it is designed to be applied to an existing benchmark for detecting multi-fault bugs and further isolating them into single-fault bugs. In our context, where the buggy program, the developer patch,

and the test cases are known, IBugFinder can determine, instead of inferring, the number of bugs to be isolated. IBugFinder performs test case minimization to create new tests with single assertions. It is different from existing test suite minimization methods [82] in that it is not aimed at reducing the number of test cases.

Our study of multi-hunk fixes is related to many existing studies of bugs and patches [3, 9, 12, 30, 60]. The key difference is that we focused on analyzing the behavioral relationships for patches of the indivisible, single-fault multi-hunk bugs rather than for example understanding the general syntactic properties of patches [60], creating a multi-fault benchmark [3], and investigating the timeline of changes for failure exposure [12].

The study conducted by Yi et al. [81] is related to ours in that it involves evaluating the effectiveness of APR in generating partial patches serving as repair hints. The key difference between their study and ours is that their study was not designed for indivisible bugs. Their definition of partial patches is based on test case execution result, and this type of partial patches do not exist for indivisible bug repair.

7 CONCLUSION AND FUTURE WORK

Previous evaluation of multi-hunk repair techniques is highly misleading, as the widely used Defects4J benchmark contains a significant number of divisible bugs. A divisible bug triggers different failures. It is not something a developer typically deals with for realistic debugging. To provide a better basis for the research of multi-hunk bug repair, we developed an approach IBugFinder for automatically detecting and creating indivisible multi-hunk bugs. We have applied the approach to 281 Defects4J multi-hunk bugs and created a new dataset containing indivisible bugs only. We evaluated existing techniques with the indivisible multi-hunk bugs. Our result shows that current APR techniques repaired only a few of the bugs and calls for more advanced approaches. Finally, we studied a variety of the patches of the indivisible bugs to understand how to repair these bugs effectively and analyze how existing techniques perform. As the main result, we identified 8 behavioral relationships characterizing the internal roles of the partial patches used for failure resolving and suggested different strategies for indivisible bug repair. We also showed that existing APR techniques do not go far beyond generating single-hunk-alike patches.

In future work, in addition to improving IBugFinder to detect divisible bugs with better efficiency and produce minimized tests with less testing overhead and applying IBugFinder to more bugs from Defects4J and other datasets for indivisible bug detection, we will also investigate using an AST-based diff utility and defining small-scale partial patches for bug isolation. Moreover, we will explore leveraging LLM to produce effective multi-hunk patches by incorporating the behavioral relationships we identified.

8 DATA-AVAILABILITY STATEMENT

Data and code for reproducing our results are available on Zenodo [75]. Updated information about the project can be found at https://github.com/qixin5/indivisible_multihunk_bug_repair.

ACKNOWLEDGMENTS

We thank He Ye for kindly sharing the ITER tool and helping us use it for the experiment. We also thank the authors of AlphaRepair, ARJA-e, Recoder, SimFix, and TBar for making their tools available. We are grateful for the valuable comments and suggestions given by the anonymous reviewers. This work was partially supported by the National Natural Science Foundation of China under the grant numbers 62202344 and 62141221 and the OPPO Research Fund.

REFERENCES

[1] Afsoon Afzal, Manish Motwani, Kathryn T Stolee, Yuriy Brun, and Claire Le Goues. 2019. SOSRepair: Expressive semantic search for real-world program repair. *IEEE Transactions on Software Engineering (TSE)* 47, 10 (2019), 2162–2181. https://doi.org/10.1109/TSE.2019.2944914

[2] AlphaRepair 2023. *The AlphaRepair tool.* https://zenodo.org/records/6819444

[3] Gabin An, Juyeon Yoon, and Shin Yoo. 2021. Searching for multi-fault programs in defects4j. In *International Symposium on Search Based Software Engineering (SSBSE)*. Springer, 153–158. https://doi.org/10.1007/978-3-030-88106-1_11

[4] Angelix 2023. *The Angelix tool.* http://angelix.io/

[5] ARJA-e 2023. *The ARJA-e tool.* https://github.com/yyxhdy/arja/tree/arja-e

[6] Artifact 2023. *The artifact.* https://github.com/qixin5/indivisible_multihunk_bug_repair

[7] Artifact APR Repair Tools and Results 2023. *The tool evaluation part of artifact.* https://github.com/BaiGeiQiShi/RepairResults

[8] Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, et al. 2021. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732* (2021). https://doi.org/10.48550/arXiv.2108.07732

[9] Lili Bo, Yue Li, Xiaobing Sun, Xiaoxue Wu, and Bin Li. 2023. Vulloc: vulnerability localization based on inducing commits and fixing commits. *Frontiers of Computer Science* 17, 3 (2023), 173207. https://doi.org/10.1007/s11704-022-1729-x

[10] CatenaD4J 2023. *The augmented bug dataset.* https://github.com/universetraveller/CatenaD4j

[11] ChatGPT 2023. *ChatGPT.* https://chat.openai.com/

[12] An Ran Chen, Md Nakhla Rafi, Shaohua Wang, et al. 2023. Back to the Future! Studying Data Cleanness in Defects4J and its Impact on Fault Localization. *arXiv preprint arXiv:2310.19139* (2023). https://doi.org/10.48550/arXiv.2310.19139

[13] Liushan Chen, Yu Pei, and Carlo A Furia. 2017. Contract-based program repair without the contracts. In *Proceedings of 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 637–647. https://doi.org/10.1109/ASE.2017.8115674

[14] Tian Cheng, Kunsong Zhao, Song Sun, Muhammad Mateen, and Junhao Wen. 2022. Effort-aware cross-project just-in-time defect prediction framework for mobile apps. *Frontiers of Computer Science* 16, 6 (2022), 166207. https://doi.org/10.1007/s11704-021-1013-5

[15] Coming 2023. *Coming.* https://github.com/SpoonLabs/coming

[16] DEAR 2023. *The DEAR tool.* https://github.com/AutomatedProgramRepair-2021/dear-auto-fix

[17] DLFix 2023. *The DLFix tool.* https://github.com/ICSE-2019-AUTOFIX/ICSE-2019-AUTOFIX

[18] Thomas Durieux, Benjamin Danglot, Zhongxing Yu, Matias Martinez, Simon Urli, and Martin Monperrus. 2017. *The Patches of the Nopol Automatic Repair System on the Bugs of Defects4J version 1.1.0.* Research Report hal-01480084. Université Lille 1 - Sciences et Technologies. https://hal.science/hal-01480084

[19] Thomas Durieux and Martin Monperrus. 2016. *IntroClassJava: A Benchmark of 297 Small and Buggy Java Programs.* Technical Report hal-01272126. Universite Lille 1. https://hal.archives-ouvertes.fr/hal-01272126/document

[20] Eclipse 2023. *The Eclipse IDE 2023-06 R Packages.* https://www.eclipse.org/downloads/packages/release/2023-06/r

[21] Aryaz Eghbali and Michael Pradel. 2024. De-Hallucinator: Iterative Grounding for LLM-Based Code Completion. *arXiv preprint arXiv:2401.01701* (2024). https://doi.org/10.48550/arXiv.2401.01701

[22] Zhiyu Fan, Xiang Gao, Martin Mirchev, Abhik Roychoudhury, and Shin Hwei Tan. 2023. Automated repair of programs from large language models. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. 1469–1481. https://doi.org/10.1109/ICSE48619.2023.00128

[23] Xiang Gao, Bo Wang, Gregory J Duck, Ruyi Ji, Yingfei Xiong, and Abhik Roychoudhury. 2021. Beyond tests: Program vulnerability repair via crash constraint extraction. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 30, 2 (2021), 1–27. https://doi.org/10.1145/3418461

[24] Ali Ghanbari, Samuel Benton, and Lingming Zhang. 2019. Practical program repair via bytecode mutation. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*. 19–30. https://doi.org/10.1145/3293882.3330559

[25] GitDiff 2023. *The Git diff utility.* https://git-scm.com/docs/git-diff

[26] Claire Le Goues, Michael Pradel, and Abhik Roychoudhury. 2019. Automated program repair. *Commun. ACM* 62, 12 (2019), 56–65. https://doi.org/10.1145/3318162

[27] Sumit Gulwani, Oleksandr Polozov, Rishabh Singh, et al. 2017. Program synthesis. *Foundations and Trends® in Programming Languages* 4, 1-2 (2017), 1–119. https://doi.org/10.1561/2500000010

[28] GZoltar 2023. *The GZoltar tool.* https://gzoltar.com/

[29] ITER 2024. *The ITER tool.* https://github.com/ASSERT-KTH/ITER

[30] Jiajun Jiang, Yingfei Xiong, and Xin Xia. 2019. A manual inspection of defects4j bugs and its implications for automatic program repair. *Science china information sciences* 62 (2019), 1–16. https://doi.org/10.1007/s11432-018-1465-6

[31] Jiajun Jiang, Yingfei Xiong, Hongyu Zhang, Qing Gao, and Xiangqun Chen. 2018. Shaping program repair space with existing patches and similar code. In *Proceedings of ACM 27th SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*. ACM, 298–309. https://doi.org/10.1145/3213846.3213871

[32] Nan Jiang, Kevin Liu, Thibaud Lutellier, and Lin Tan. 2023. Impact of Code Language Models on Automated Program Repair. In *Proceedings of the 45th International Conference on Software Engineering (ICSE)*. IEEE, 1430–1442. https://doi.org/10.1109/ICSE48619.2023.00125

[33] Nan Jiang, Thibaud Lutellier, Yiling Lou, Lin Tan, Dan Goldwasser, and Xiangyu Zhang. 2023. Knod: Domain knowledge distilled tree decoder for automated program repair. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 1251–1263. https://doi.org/10.1109/ICSE48619.2023.00111

[34] René Just, Darioush Jalali, and Michael D Ernst. 2014. Defects4J: A database of existing faults to enable controlled testing studies for Java programs. In *Proceedings of ACM 23rd SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*. ACM, 437–440. https://doi.org/10.1145/2610384.2628055

[35] Dongsun Kim, Jaechang Nam, Jaewoo Song, and Sunghun Kim. 2013. Automatic patch generation learned from human-written patches. In *Proceedings of the 35th International Conference on Software Engineering (ICSE)*. IEEE, 802–811. https://doi.org/10.1109/ICSE.2013.6606626

[36] Jisung Kim and Byungjeong Lee. 2023. MCRepair: Multi-Chunk Program Repair via Patch Optimization with Buggy Block. In *Proceedings of the 38th ACM/SIGAPP Symposium on Applied Computing (SAC)*. 1508–1515. https://doi.org/10.1145/3555776.3577762

[37] Amy J Ko and Brad A Myers. 2008. Debugging reinvented: asking and answering why and why not questions about program behavior. In *Proceedings of the 30th international conference on Software engineering (ICSE)*. 301–310. https://doi.org/10.1145/1368088.1368130

[38] Triet HM Le, Hao Chen, and Muhammad Ali Babar. 2020. Deep learning for source code modeling and generation: Models, applications, and challenges. *ACM Computing Surveys (CSUR)* 53, 3 (2020), 1–38. https://doi.org/10.1145/3383458

[39] Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, and Westley Weimer. 2011. GenProg: A generic method for automatic software repair. *IEEE Transactions on Software Engineering (TSE)* 38, 1 (2011), 54–72. https://doi.org/10.1109/TSE.2011.104

[40] Yi Li, Shaohua Wang, and Tien N Nguyen. 2020. Dlfix: Context-based code transformation learning for automated program repair. In *Proceedings of IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*. IEEE/ACM, 602–614. https://doi.org/10.1145/3377811.3380345

[41] Yi Li, Shaohua Wang, and Tien N Nguyen. 2022. DEAR: A novel deep learning-based approach for automated program repair. In *Proceedings of IEEE/ACM 44th International Conference on Software Engineering (ICSE)*. IEEE/ACM, 25–27. https://doi.org/10.1145/3510003.3510177

[42] Kui Liu, Anil Koyuncu, Dongsun Kim, and Tegawendé F Bissyandé. 2019. TBar: Revisiting template-based automated program repair. In *Proceedings of ACM 28th SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*. ACM, 31–42. https://doi.org/10.1145/3293882.3330577

[43] Fernanda Madeiral, Simon Urli, Marcelo Maia, and Martin Monperrus. 2019. Bears: An extensible java bug benchmark for automatic program repair studies. In *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 468–478. https://doi.org/10.1109/SANER.2019.8667991

[44] Matias Martinez, Thomas Durieux, Romain Sommerard, Jifeng Xuan, and Martin Monperrus. 2017. Automatic repair of real bugs in java: A large-scale experiment on the defects4j dataset. *Empirical Software Engineering* 22 (2017), 1936–1964. https://doi.org/10.1007/s10664-016-9470-4

[45] MCRepair 2023. *The artifact of MCRepair*. https://github.com/kimjisung78/MCRepair

[46] Sergey Mechtaev, Jooyong Yi, and Abhik Roychoudhury. 2016. Angelix: Scalable multiline program patch synthesis via symbolic analysis. In *Proceedings of IEEE/ACM 38th International Conference on Software Engineering (ICSE)*. IEEE/ACM, 691–701. https://doi.org/10.1145/2884781.2884807

[47] Xiangxin Meng, Xu Wang, Hongyu Zhang, Hailong Sun, and Xudong Liu. 2022. Improving fault localization and program repair with deep semantic features and transferred knowledge. In *Proceedings of the 44th International Conference on Software Engineering (ICSE)*. IEEE/ACM, 1169–1180. https://doi.org/10.1145/3510003.3510147

[48] Xiangxin Meng, Xu Wang, Hongyu Zhang, Hailong Sun, Xudong Liu, and Chunming Hu. 2023. Template-based Neural Program Repair. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. 1456–1468. https://doi.org/10.1109/ICSE48619.2023.00127

[49] Martin Monperrus. 2018. *The Living review on automated program repair*. Technical Report hal-01956501. HAL/archives-ouvertes.fr.

[50] ODS 2023. *ODS*. https://github.com/SophieHYe/ODSExperiment

[51] Alexandre Perez, Rui Abreu, and Marcelo d'Amorim. 2017. Prevalence of single-fault fixes and its impact on fault localization. In *2017 IEEE International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 12–22. https://doi.org/10.1109/ICST.2017.9

[52] Julian Aron Prenner, Hlib Babii, and Romain Robbes. 2022. Can OpenAI's codex fix bugs? an evaluation on QuixBugs. In *Proceedings of the Third International Workshop on Automated Program Repair (APR)*. 69–75. https://doi.org/10.1145/3524459.3527351

[53] Recoder 2023. *The Recoder tool.* https://github.com/pkuzqh/Recoder

[54] RewardRepair 2023. *The RewardRepair tool.* https://github.com/ASSERT-KTH/RewardRepair

[55] Ripon K Saha, Yingjun Lyu, Wing Lam, Hiroaki Yoshida, and Mukul R Prasad. 2018. Bugs.jar: A large-scale, diverse dataset of real-world java bugs. In *Proceedings of the 15th international conference on mining software repositories (MSR)*. 10–13. https://doi.org/10.1145/3196398.3196473

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

[57] Seemanta Saha, Ripon k. Saha, and Mukul r. Prasad. 2019. Harnessing evolution for multi-hunk program repair. In *Proceedings of IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE/ACM, 13–24. https://doi.org/10.1109/ICSE.2019.00020

[58] SimFix 2023. *The SimFix tool.* https://github.com/xgdsmileboy/SimFix

[59] Dominik Sobania, Martin Briesch, Carol Hanna, and Justyna Petke. 2023. An analysis of the automatic bug fixing performance of chatgpt. *arXiv preprint arXiv:2301.08653* (2023). https://doi.org/10.48550/arXiv.2301.08653

[60] Victor Sobreira, Thomas Durieux, Fernanda Madeiral, Martin Monperrus, and Marcelo de Almeida Maia. 2018. Dissection of a bug dataset: Anatomy of 395 patches from defects4j. In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 130–140. https://doi.org/10.1109/SANER.2018.8330203

[61] Yi Song, Xiaoyuan Xie, Quanming Liu, Xihao Zhang, and Xi Wu. 2022. A comprehensive empirical investigation on failure clustering in parallel debugging. *Journal of Systems and Software* 193 (2022), 111452. https://doi.org/10.1016/j.jss.2022.111452

[62] Soot 2023. *The Soot framework.* http://soot-oss.github.io/soot/

[63] Shin Hwei Tan, Hiroaki Yoshida, Mukul R Prasad, and Abhik Roychoudhury. 2016. Anti-patterns in search-based program repair. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*. 727–738. https://doi.org/10.1145/2950290.2950295

[64] TBar 2023. *The TBar tool.* https://github.com/TruX-DTF/TBar

[65] Nikolaos Tsantalis, Ameya Ketkar, and Danny Dig. 2020. RefactoringMiner 2.0. *IEEE Transactions on Software Engineering (TSE)* 48, 3 (2020), 930–950. https://doi.org/10.1109/TSE.2020.3007722

[66] VarFix 2023. *The VarFix tool.* https://github.com/chupanw/vbc

[67] Song Wang, Taiyue Liu, and Lin Tan. 2016. Automatically learning semantic features for defect prediction. In *Proceedings of the 38th International Conference on Software Engineering (ICSE)*. 297–308. https://doi.org/10.1145/2884781.2884804

[68] Yuxiang Wei, Chunqiu Steven Xia, and Lingming Zhang. 2023. Copiloting the copilots: Fusing large language models with completion engines for automated program repair. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. 172–184. https://doi.org/10.1145/3611643.3616271

[69] Ming Wen, Junjie Chen, Rongxin Wu, Dan Hao, and Shing-Chi Cheung. 2018. Context-aware patch generation for better automated program repair. In *Proceedings of IEEE/ACM 40th International Conference on Software Engineering (ICSE)*. 1–11. https://doi.org/10.1145/3180155.3180233

[70] Chu-Pan Wong, Priscila Santiesteban, Christian Kästner, and Claire Le Goues. 2021. VarFix: Balancing edit expressiveness and search effectiveness in automated program repair. In *Proceedings of ACM 29th Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. ACM, 354–366. https://doi.org/10.1145/3468264.3468600

[71] Chunqiu Steven Xia, Yuxiang Wei, and Lingming Zhang. 2023. Automated program repair in the era of large pre-trained language models. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 1482–1494. https://doi.org/10.1109/ICSE48619.2023.00129

[72] Chunqiu Steven Xia and Lingming Zhang. 2022. Less training, more repairing please: revisiting automated program repair via zero-shot learning. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. 959–971. https://doi.org/10.1145/3540250.3549101

[73] Chunqiu Steven Xia and Lingming Zhang. 2023. Keep the Conversation Going: Fixing 162 out of 337 bugs for $0.42 each using ChatGPT. *arXiv preprint arXiv:2304.00385* (2023). https://doi.org/10.48550/arXiv.2304.00385

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

[75] Qi Xin, Haojun Wu, Jinran Tang, Xinyu Liu, Steven P. Reiss, and Jifeng Xuan. 2024. Detecting, Creating, Repairing, and Understanding Indivisible Multi-Hunk Bugs: Replication Package. https://doi.org/10.5281/zenodo.11097846

[76] Yingfei Xiong, Jie Wang, Runfa Yan, Jiachen Zhang, Shi Han, Gang Huang, and Lu Zhang. 2017. Precise condition synthesis for program repair. In *Proceedings of IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. 416–426. https://doi.org/10.1109/ICSE.2017.45

[77] Jifeng Xuan, Matias Martinez, Favio Demarco, Maxime Clement, Sebastian Lamelas Marcote, Thomas Durieux, Daniel Le Berre, and Martin Monperrus. 2016. Nopol: Automatic repair of conditional statement bugs in java programs. *IEEE Transactions on Software Engineering (TSE)* 43, 1 (2016), 34–55. https://doi.org/10.1109/TSE.2016.2560811

[78] Deheng Yang, Xiaoguang Mao, Liqian Chen, Xuezheng Xu, Yan Lei, David Lo, and Jiayu He. 2022. TransplantFix: Graph Differencing-based Code Transplantation for Automated Program Repair. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 1–13. https://doi.org/10.1145/3551349.3556893

[79] He Ye, Matias Martinez, and Martin Monperrus. 2022. Neural program repair with execution-based backpropagation. In *Proceedings of the 44th International Conference on Software Engineering (ICSE)*. 1506–1518. https://doi.org/10.1145/3510003.3510222

[80] He Ye and Martin Monperrus. 2024. ITER: Iterative Neural Repair for Multi-Location Patches. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering (ICSE)*. 1–13. https://doi.org/10.1145/3597503.3623337

[81] Jooyong Yi, Umair Z Ahmed, Amey Karkare, Shin Hwei Tan, and Abhik Roychoudhury. 2017. A feasibility study of using automated program repair for introductory programming assignments. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering (FSE)*. 740–751. https://doi.org/10.1145/3106237.3106262

[82] Shin Yoo and Mark Harman. 2012. Regression testing minimization, selection and prioritization: a survey. *Software testing, verification and reliability* 22, 2 (2012), 67–120. https://doi.org/10.1002/stv.430

[83] Yuan Yuan and Wolfgang Banzhaf. 2018. ARJA: Automated repair of java programs via multi-objective genetic programming. *IEEE Transactions on software engineering (TSE)* 46, 10 (2018), 1040–1067. https://doi.org/10.1109/TSE.2018.2874648

[84] Yuan Yuan and Wolfgang Banzhaf. 2019. A hybrid evolutionary system for automatic software repair. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO)*. 1417–1425. https://doi.org/10.1145/3321707.3321830

[85] Yuan Yuan and Wolfgang Banzhaf. 2020. Toward better evolutionary program repair: An integrated approach. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 29, 1 (2020), 1–53. https://doi.org/10.1145/3360004

[86] Abubakar Zakari and Sai Peck Lee. 2019. Parallel debugging: An investigative study. *Journal of Software: Evolution and Process* 31, 11 (2019), e2178. https://doi.org/10.1002/smr.2178

[87] Abubakar Zakari, Sai Peck Lee, Rui Abreu, Babiker Hussien Ahmed, and Rasheed Abubakar Rasheed. 2020. Multiple fault localization of software programs: A systematic literature review. *Information and Software Technology* 124 (2020), 106312. https://doi.org/10.1016/j.infsof.2020.106312

[88] Quanjun Zhang, Chunrong Fang, Yuxiang Ma, Weisong Sun, and Zhenyu Chen. 2023. A survey of learning-based automated program repair. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 33, 2 (2023), 1–69. https://doi.org/10.1145/3631974

[89] Hao Zhong and Zhendong Su. 2015. An empirical study on real bug fixes. In *Proceedings of IEEE/ACM 37th International Conference on Software Engineering (ICSE)*, Vol. 1. IEEE/ACM, 913–923. https://doi.org/10.1109/ICSE.2015.101

[90] Wenkang Zhong, Chuanyi Li, Kui Liu, Tongtong Xu, Jidong Ge, Tegawendé F Bissyandé, Bin Luo, and Vincent Ng. 2024. Practical Program Repair via Preference-based Ensemble Strategy. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering (ICSE)*. 1–13. https://doi.org/10.1145/3597503.3623310

[91] Qihao Zhu, Zeyu Sun, Yuan-an Xiao, Wenjie Zhang, Kang Yuan, Yingfei Xiong, and Lu Zhang. 2021. A syntax-guided edit decoder for neural program repair. In *Proceedings of ACM 29th Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. ACM, 341–353. https://doi.org/10.1145/3468264.3468544

Received 2023-09-28; accepted 2024-04-16