Interactive Patch Filtering as Debugging Aid

Jingjing Liang*†, Ruyi Ji*†,Jiajun Jiang‡, Shurui Zhou§, Yiling Lou*†, Yingfei Xiong*† and Gang Huang*†
*Key Laboratory of High Confidence Software Technologies (Peking University), MoE, Beijing, PR China
†Department of Computer Science and Technology, EECS, Peking University, Beijing, PR China
‡College of Intelligence and Computing, Tianjin University, Tianjin, PR China
§University of Toronto, Toronto, Canada
*{jingjingliang,jiruyi910387714,louyiling,xiongyf,hg}@pku.edu.cn, ‡jiangjiajun@tju.edu.cn, §shuruiz@ece.utoronto.ca

***Abstract*—It is widely recognized that patches generated by program repair tools have to be correct to be useful. However, it is fundamentally difficult to ensure the correctness of the patches. Many tools generate only the patches that are highly likely to be correct by taking conservative strategies which inevitably limit the recall of APR approaches. While the recall of APR can potentially be improved by relaxing the requirement on precision, more incorrect patches may also be generated.**

In this paper, we conjecture that reviewing incorrect patches also helps developers to understand the bug, and with proper tool support, reviewing incorrect patches would at least not reduce the repair performance. To evaluate this, we propose an interactive patch filtering approach to facilitate developers in the patch review process via effectively filtering out groups of incorrect patches.

We implemented the approach as an Eclipse plugin, *InPaFer*, and evaluated the effectiveness and usefulness with a mixed-method evaluation. The results show that our approach improves the repair performance of developers, with 62.5% more successfully repaired bugs and 25.3% less debugging time. In particular, even if all generated patches are incorrect, the performance of developers would not be significantly reduced, and could still be improved. Our work provides a new way of thinking for the APR research.

***Index Terms*—Interactive debugging, Patch filtering, User study, Program repair**

I. Introduction

In the past decades, automatic program repair (APR) attracted a lot of research efforts [1]–[18]. Many of the proposed APR techniques are test-based, which take a buggy program and a test suite with at least one failed test as input and automatically generate a set of patches that make all tests pass.

It is commonly recognized that the generated patches have to be correct to be useful. Multiple existing studies have revealed that the performance (i.e., debugging time and success rates) of developers increases when they are provided with only correct patches. But this performance increase disappears when incorrect patches are also provided, and becomes negative when the developers are provided with only incorrect patches [19]–[22]. However, it is fundamentally difficult to ensure the correctness of the patches because in practice we usually only have a weak specification of the program behavior, usually a test suite, such that many more incorrect patches can meet the specification than correct patches [23]. This is known as the problem of "overfitting" [24], [25] or "weak test suite" [26].

As a result, many approaches take conservative strategies [27]–[32], generating only the patches that are highly likely to be correct, and thus inevitably limiting the recall of APR approaches. For example, Hercules [18], the approach that generates correct patches for the highest number of bugs in Defects4J [33] within our knowledge, only generates correct patches for 13% of the bugs. Such a low recall limits the amount of aid that program repair approaches could give to the developers, and improving the recall of program repair approaches is desirable.

If the requirement on precision can be loosened, the recall of APR can potentially be improved. To validate this point, we consider a basic way of improving the recall at the cost of the precision: combining the patches produced by different program repair approaches. In this way, a correct patch can be generated for a bug if any of the combined approaches generates a correct patch for the bug, and the recall of the combined approach may be significantly higher than any of the individual approaches. This combination is also feasible as current repair approaches often require tens of minutes to fix a bug [29], [30], [34]–[36] and are assumed to work offline, e.g., after a nightly build and before the working time of the next day. In such a case, we have a whole night to invoke different program repair approaches.

We conducted a preliminary study by collecting the patches generated by 13 existing program repair approaches on Defects4J [33]. The details of the study can be found in Section II. The results show that the combined approach significantly improves recall with 124% more bugs fixed. However, more incorrect patches are also generated, and the incorrect patches tend to gather together: on many bugs multiple incorrect patches are generated.

Therefore, in this paper we ask a question: **would an APR approach be useful if multiple incorrect patches are generated along with the correct patches?** If the answer is yes, it enables the above approach of combining APR tools and opens doors to multiple possibilities. For example, a future APR tool could take a more aggressive strategy to generate as many patches as possible such that the chance of generating a correct patch is higher.

We have observed that the practical usage of APR tools

This work is supported in part by the National Key Research and Development Program of China No. 2019YFE0198100, National Natural Science Foundation of China under Grant No. 61922003. Yingfei Xiong is the corresponding author.

consists of two steps [37]: (1) patch generation, where the APR tool proposes candidate patches; and (2) patch review, where the developer examines the patches to ensure the correctness and other quality attributes. While patch generation has been extensively studied, we still lack understanding and tool support for the patch review step. Since incorrect patches might provide developers with information about the bug from different perspectives, **we conjecture that reviewing incorrect patches also helps developers to understand the bug, and with proper tool support, reviewing incorrect patches would at least not reduce the repair performance (i.e., debugging time and success rates) of the developers**.

To verify this conjecture, in this paper we propose an interactive patch filtering approach to help developers identify correct patches for a bug from a group of candidate patches. Our approach groups the candidate patches based on the program attributes and generates a list of filtering criteria. For example, the attributes could be about the behavior of a test (e.g., whether a statement should be executed during a test or not). Developers pick a filtering criterion and provide an answer ('Yes' or 'No') to keep or filter out the corresponding patches. The process continues until developers figure out a correct patch (by picking an automatically generated one or contriving a new one) or no more patches can be filtered out. We named our approach *InPaFer*, which stands for **In**teractive **Pa**tch **F**ilt**er**. We design a two-stage algorithm, introducing an offline preparation stage to optimize the response time of the online interaction stage. We developed *InPaFer* as an Eclipse plugin as illustrated in Figure 3.

We first quantitatively evaluate the effectiveness of the filtering criteria of *InPaFer* in an ideal setting in controlled experiments. Then we conduct non-trivial user studies with 200 debugging sessions to evaluate the effectiveness and usefulness of *InPaFer* in the real debugging scenario. The results show that *InPaFer* improves the repair performance of developers. In particular, compared with manual debugging, developers with *InPaFer* and a set of generated patches correctly repaired 62.5% more bugs and used 25.3% less time on average. Even if all generated patches are incorrect, the performance of developers would not be significantly reduced, and could still be improved. This confirms our conjecture: with proper tool support, the patch review process helps developers understand the bug, and eventually contributes to the debugging process.

This study provides a new way of thinking for the APR research – instead of sacrificing the recall for the precision of APR techniques, we could improve the patch review process by quickly narrowing down to the correct patch candidates and/or giving developers hints for correct patches. In this way, we do not have to sacrifice the recall of the APR tools and could assist developers on many more bugs.

In summary, this paper makes the following contributions:

1) A novel finding that, with proper tool support in patch review, an APR with low precision could still be useful.
2) An interactive patch filtering approach to supporting the patch review step, and a two-stage algorithm to implement the approach.
3) An Eclipse plugin with a carefully designed interface for the user to easily filter out incorrect patches and better understand the bug and patches.
4) Non-trivial user studies to investigate the effectiveness and usefulness of *InPaFer*.

TABLE I: APR tools included in *InPaFer*

Name	Description
jKali [9]	The Java version of Kali [26], which only performs functionality deletion.
jGenProg [9]	The Java version of GenProg [35], [38], which repair bugs with genetic programming algorithm.
kPAR [39]	The Java version of PAR [40], which generate patches based on predefined fix patterns.
Nopol [6]	Repairing buggy conditions with constraint solving.
jMutRepair [36]	Repairing bugs with a set of predefined mutators.
Cardumen [34]	Generating patches based on mined templates.
Avatar [4]	Repairing bugs with fix patterns of static analysis violations.
HDrepair [5]	A repair tool based on historical bug-fix information.
ACS [28]	Learning statistical information from open source programs for fixing incorrect conditions.
3sfix [41] CapGen [29] SimFix [30]	Repair approaches based on similar code match.
DeepRepair [42]	An extension of jGenProg, which leverages code similarity.

II. Preliminary Study

In order to validate the feasibility of combining program repair approaches, we conducted a preliminary study by collecting patches generated by existing APR tools. Since our goal is to maximize the recall, we considered all tools that are listed at program-repair.org in May 2019 (the time we designed the experiments in this research), and included each tool if (1) it has published patches on Defects4J [33], and (2) adding the tool increases the recall of the combined approach. As a result, we collected all patches generated by 13 APR tools (listed in Table I), covering 147 bugs on Defects4J 1.0.0. From the statistics on the numbers of bugs in Table II, we have the following three observations:

Combining existing approaches can significantly improve the recall. Table II shows that the combination can provide correct patches for 76 bugs, which is 124% more than the best performed individual approach (34 bugs) [30].

Incorrect patches are often generated along with correct patches. We observe that among all bugs where correct patches are generated, 47% (36/76) of the bugs also have incorrect patches generated.

Gathering effect: when two patches could be generated for one bug, more patches tend to be generated. Table II shows that 80% (71/89) of the bugs have at least 3 patches among all bugs where multiple patches are generated. The reason for the gathering effect is intuitive: when multiple patches could pass the tests, the tests are probably weak and more patches could be generated.

III. Interactive Patch Filtering

In this section, we first use an illustrative example to introduce the workflow of *InPaFer* (Section III-A), and then

TABLE II: Dataset in preliminary study

Project	Patches (Contain)			Patches (Not Contain)			Total
	Single	Multiple		Single	Multiple		
	(1)	(2)	(3,n)	(1)	(2)	(3,n)	
Chart	2	1	8	3	3	5	22
Closure	11	1	2	6	3	7	30
Lang	10	0	8	4	2	3	27
Math	15	0	15	3	6	21	60
Time	2	1	0	2	1	2	8
Total	40	3	33	18	15	38	147

Patches (Contain): the number of bugs whose candidate patches contain correct patches, **Patches (Not Contain)**: the number of bugs whose candidate patches do not contain correct patches.
Single (1): the number of bugs whose candidate patches contain only one patch, **Multiple (2)**: the number of bugs whose candidate patches contain two patches, **Multiple (3,n)**: the number of bugs whose candidate patches contain at least three patches.

```
313 public double eval(final double[] val, ...){
      ...
320   if(length == 1){//incorrect patches change
    here
321       var = 0.0;
322   }else if(length > 1){
323       Mean mean = new Mean();
324       double m = mean.evaluate(val,weights,...);
325       var = evaluate(val,weights,m,begin,length);
326   }
      ...
329 }

501 public double evaluate(double[] values, ...) {
      ...
520     for(int i=0;i < weights.length;i++){ //buggy
521       sumWts += weights[i];
522     }
      ...
532 }
```

Fig. 1: A code snippet from Math41

demonstrate the interactive patch filtering process in detail (Section III-B and III-C).

A. Illustrative Example

Figure 1 shows a code snippet from bug *Math41* in Defects4J [33] benchmark, which invokes the buggy method `evaluate()` when the condition `length>1` is satisfied (line 322). To repair this bug, existing APR tools generated a set of candidate patches. Table III lists three of such patches, which can make the corresponding test suite pass. In the table, the second column denotes the line number of the code in Figure 1 where the corresponding patch changes, while the last two columns present the patch details and their correctness.

From Table III we can see that the patches and the corresponding patched programs exhibit different dynamic and/or static attributes. For example, the code changes occur in method `eval()` for patches p_2 and p_3, while method `evaluate()` for p_1. In addition, after applying the candidate patches and running tests, we can find that patches p_2 and p_3 make the failing test case go into the `then` branch in line 321. In contrast, the `else` branch between lines 323 and 325 will be taken when applying p_1. Therefore, checking the correctness of such program attributes can help developers

TABLE III: Candidate patches for Math41

ID	Line	Changes	Correct
p_1	520	`- for (i = 0; i < weights.length; i++){` `+ for (i = begin; i < begin + length; i++){`	Yes
p_2	320	`- if (length == 1){` `+ if (length == 5){`	No
p_3	320	`- if (length == 1){` `+ if ((length & 1) == 1){`	No

TABLE IV: First two filtering criteria for Math41

ID	Program Attribute	Patch	Option
f_1	Modify method *evaluate()* in class *Variance*	p_1	Yes/No
f_2	Execute code in *line 321* in class *Variance*	p_2,p_3	Yes/No

Patch: the patches satisfying the corresponding program attribute.

to filter out the incorrect patches. For instance, if developers have more confidence to believe that the buggy code should reside in method `evaluate()`, the patches changing code in other methods can be filtered out (i.e., p_2 and p_3).

According to this observation, we design an interactive patch filtering tool, called *InPaFer*. Given a set of patches for a bug, *InPaFer* analyzes what program attributes could distinguish the patches, and presents Yes/No options about the correctness of the program attributes to the developer. Such an option is called a *filtering criterion*. When an answer is provided to a filtering criterion, *InPaFer* will automatically filter out the corresponding patches that contradict the answer ('Yes' for confirmation or 'No' for rejection). The process of providing an answer and filtering patches is called a *filtering step*. *InPaFer* proceeds the filtering steps interactively with developers until a termination state is reached.

Table IV presents two representative filtering criteria for the patches listed in Table III. Suppose the developer first selects f_2, and rejects it (i.e., answer *No*). The patches corresponding to the program attribute in f_2 will be filtered out (i.e., p_2 and p_3), while other patches that do not meet the attribute will remain (i.e., p_1).

Providing answers to the filtering criteria also helps understand the bug. For example, providing an answer to f_1 would draw the developer's attention to method `evaluate()`, where the root cause of the bug resides. Providing an answer to f_2 helps the developer understand that the value of `var` is related to the error (since resetting it to zero makes the test pass). Therefore, these efforts will eventually contribute to the repair process and would not be wasted even if a correct patch does not stand out in the end.

There is a challenge in implementing *InPaFer*: since collecting program attributes may take a lot of time, such as program execution traces, it is impractical to provide a timely response for online debugging. To overcome this challenge, we utilize the fact that repair approaches are assumed to work offline, e.g., after a nightly build and before the working time of the next day, and design *InPaFer* as a two-stage approach. As shown in Figure 2, the preparation stage is an offline process that collects program attributes and constructs filtering criteria, while the interaction stage is an online process that performs

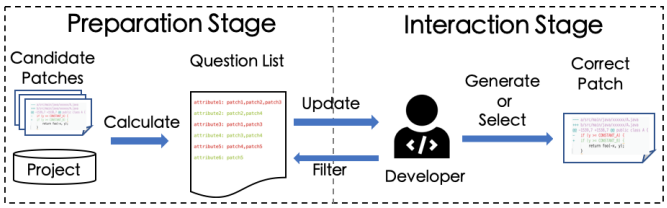


Fig. 2: The workflow of *InPaFer*

patch filtering via interacting with developers, which can be achieved within a short response time.

B. åPreparation Stage

Preparation stage is an offline process that performs data preparation for the next interaction stage. Given a set of patches related to a bug, *InPaFer* automatically collects program attributes for different patches, which will be finally leveraged to construct a set of filtering criteria for interaction.

Generally, any kind of attributes can be used in our approach as long as they can distinguish the candidate patches from some perspectives. However, the attributes that can distinguish more patches and are easy to understand for developers should be preferred, because they potentially can decrease the number of interactions and reduce the burden on developers. As a result, the current implementation of *InPaFer* employs three kinds of program attributes, including both static code property (`Modified Method`) and dynamic runtime features (`Execution Trace` and `Variable Value`). According to prior studies [19], [43]–[45], these attributes are useful for developers to understand the program during debugging and have the ability to distinguish candidate patches. The details of each attribute are described as follows.

- **`Modified Method`** can be interpreted as "*The method m in class c should be patched*", where *m* and *c* represent the names of a method and a class, respectively. *InPaFer* analyzes the patches and collects their modified methods during the preparation stage. When a patch changes multiple methods, *InPaFer* collects all of them.
- **`Execution Trace`** can be interpreted as "*The statement at line n in class c should be executed*", where *n* and *c* represent the line number and the class, respectively. However, the complete execution trace can be too lengthy for manual check. *InPaFer* currently only considers the execution trace in those methods that are modified by candidate patches, which is easier for manual check since it is close to the changed code.
- **`Variable Value`** can be interpreted as "*The value val is assigned to var during program execution*", where *val* and *var* represent a value and a variable, respectively. Particularly, *InPaFer* considers all local variables and class fields with primitive types at the entry and exit locations of the modified methods. The reason that only these variable assignments are considered is not to overwhelm the developer with too many filtering criteria.

For each kind of attribute, *InPaFer* constructs a set of filtering criteria, which will be used in the interaction stage. For `Execution Trace` and `Variable Value`, we consider the information under only failing test cases, because there are a lot of passing test cases that may overwhelm the developer. If a bug has more than one failing test case, *InPaFer* will collect the information for different patches under each failing test case. All of them will be used to construct filtering criteria. Additionally, *InPaFer* will remove all the filtering criteria that cannot distinguish any candidate patch.

C. Interaction Stage

Interaction stage is an online process with developers (shown in Figure 2) using the filtering criteria constructed in the preparation stage. The input of this stage is a list of filtering criteria and the complete project under debugging. Each time, *InPaFer* collects the feedback from developers for a certain filtering criterion, and updates the candidate filtering criteria and patches in accordance. More concretely, in the interactive debugging process, there are in total three kinds of actions that a developer can take.

- ***Filter*** Confirming or rejecting a filtering criterion to filter candidate patches.
- ***Select*** Selecting a patch from candidates as the correct patch.
- ***Generate*** Generating a correct patch manually to fix the bug.

For each filtering step, *InPaFer* (1) updates the candidate patches by keeping only the patches consistent with the answer, (2) updates the filtering criteria by keeping only those associating with only a proper subset of the remaining patches, and (3) presents the updated patches and filtering criteria to the developer. The process terminates when the developer ***Select*** or ***Generate*** a correct patch, or when there is no criterion left.

IV. Eclipse Plugin & User Interface

We implemented *InPaFer* in a prototype of the same name, which is a plugin for Eclipse with a graphical user interface (GUI) [46].Figure 3 shows a snapshot of the plugin during a debugging process, which corresponds to the example shown in Figure 1. Specifically, it consists of two embedded views, *Filter View* and *Diff View*.

Filter View is the main component of our approach, which presents the filtering criteria and corresponding candidate patches. Specifically, we designed three panels: (1) Panel 1 renders the failing test cases and the number of corresponding candidate patches; (2) Panel 2 shows a list of filtering criteria related to the currently selected test, each including the attribute detail, the number of related patches, and the state of the criterion (YES – confirm, NO – reject, UNCLEAR – no answer). The attribute detail includes the three kinds of attributes in Section III-B. For example, *Variance#321* in `Execution Trace` represents "*The code at line 321 in class Variance should be executed*". Users could click 'Yes' to filter out all patches uncovered by the criterion and 'No' to filter out the complement, and the state of the criterion is updated accordingly; and (3) Panel 3 displays the candidate patches covered by the latest filtering criterion that

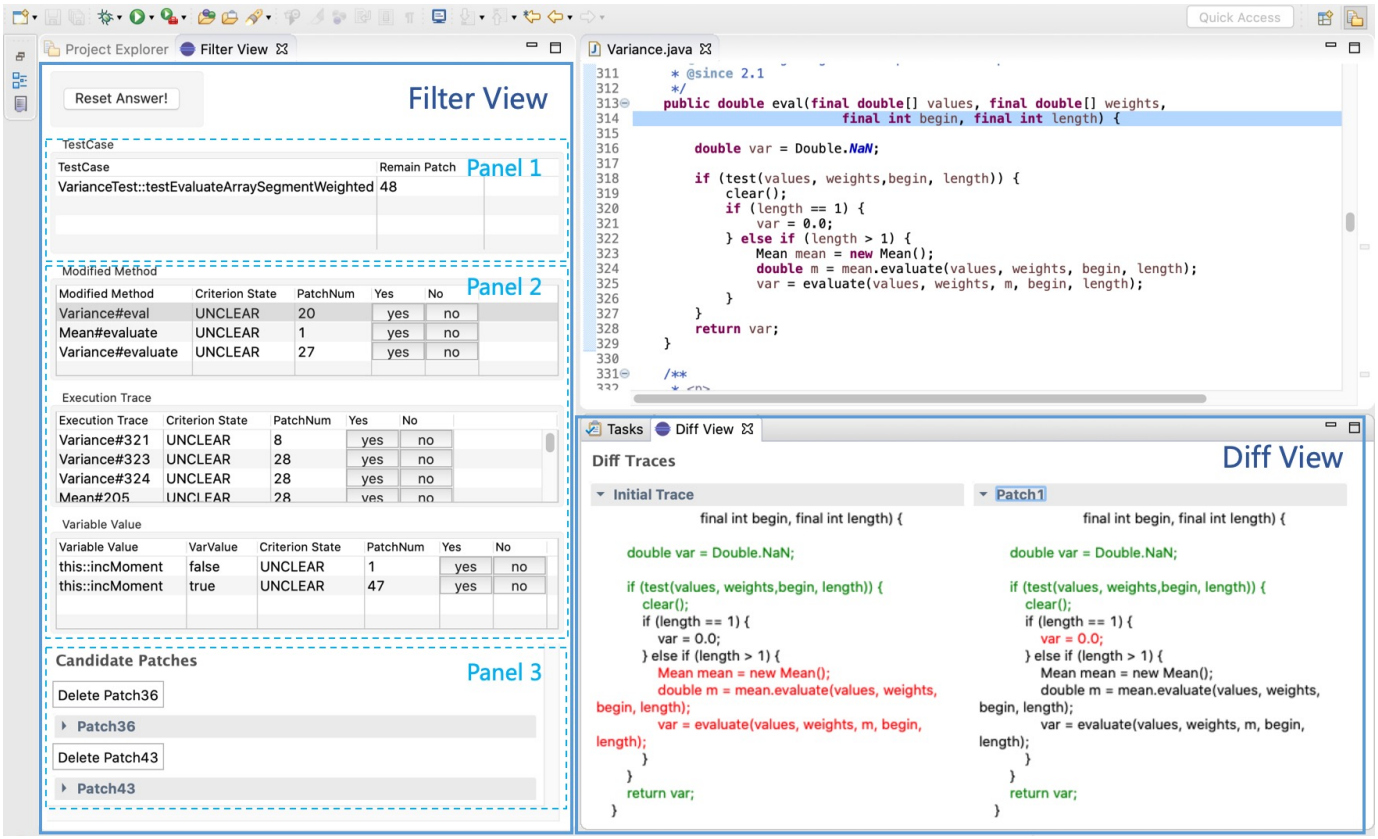


Fig. 3: The screenshot of *InPaFer* plugin. Panel 1 presents the number of current candidate patches, Panel 2 presents the filtering criteria based on three kinds of attributes, and Panel 3 presents the detailed information of candidate patches.

was clicked. Additionally, the plugin also provides a one-click rollback to reset all the answers.

Diff View is an auxiliary view to visualize the differences of execution traces before and after applying a patch to the buggy program, where the **green** lines of code are commonly covered by the failing test before and after the repair, while the **red** lines of code are particularly covered by one of them. Finally, the other lines of code are changed by the patch or not covered by any of them. In this way, the user can clearly understand the impact of the patch on the program execution.

For the convenience of debugging, all views or panels are logically interrelated, and the selection of one part may trigger the update of other places. For example, when a test is selected in Panel 1, Panel 2 will be updated to show the filtering criteria related to the test. When a patch is selected, the *Diff View* will refresh the trace difference immediately. The *Filter View* is also logically related to the text editor of Eclipse. For example, if the user selects a `Modified Method`-related filtering criterion in *Filter View*, the cursor of the text editor will jump into the corresponding modified method. Similarly, users could locate the changed code in the text editor by simply selecting a patch.

V. Evaluation

To evaluate the effectiveness and usefulness of *InPaFer*, we address the following four research questions:

- **RQ1:** How effective are the filtering criteria in filtering incorrect patches?
- **RQ2:** Can *InPaFer* improve the debugging efficiency and success rates of developers in the real world?
- **RQ3:** Do developers regard *InPaFer* useful, and if so, what aspect of *InPaFer* is most helpful?
- **RQ4:** How does *InPaFer* compare with fault localization techniques?

We answer the first research question in controlled experiments in an ideal setting, in which we quantitatively measure the ratio of incorrect patches that can be filtered out and the number of filtering steps required if all answers are correct. Subsequently, we qualitatively answer the remaining three research questions via user studies, in which we provide an Eclipse plugin to developers and measure the debugging performance of the developers. We use the Paired Wilcoxon rank-sum test to establish statistical significance.

A. Quantitative Study (RQ1)

In this study, we answer RQ1 by quantitatively measuring the ratio of incorrect patches that could be filtered out if all answers are correct, and the number of filtering steps required.

1) Experimental Setup:

a) Dataset: We use the dataset collected during our preliminary study (Section II). We filtered out 58 bugs that have only one patch, and the *Time11* bug that causes an

TABLE V: Dataset in experiments

Project	KLoC	#Bugs	#AvgPatch	#C/#NC
Chart	96	17	225	9/8
Closure	90	13	6	3/10
Lang	22	13	66	8/5
Math	85	42	92	15/27
Time	28	3	9	1/2
Total	321	88	99	36/52

KLoC: thousands of lines of code, **#Bugs**: number of bugs per project, **#AvgPatch**: average number of patches per bug, **#C/#NC**: number of bugs which contain or do not contain correct patches.

TABLE VI: (RQ1) Ratio of incorrect patches that are successfully filtered by *InPaFer* for all bugs

Ratio of Incorrect Patches Filtered	0	$(0, 100\%)$	100%	Total
Number of Bugs	21	25	42	88

instrumentation issue. In total, we collected patches for 88 bugs of 5 Java projects (See Table V).

b) Procedure: To simulate the interactive process of reviewing patches using *InPaFer*, for a given bug and corresponding candidate patches, we randomly select one filtering criterion each time and then automatically provide the correct answer via analyzing the patched program. The simulative process terminates when there is no filtering criterion left. We repeat the experiment ten times and calculate the average results. Please note that we do not simulate ***Select*** or ***Generate***, and no correct patch could be possibly filtered out.

2) Results:

a) Effectiveness: Table VI shows the ratio of the incorrect patches that are successfully filtered out by the filtering criteria in *InPaFer*. The result shows that, for 47.7% (42/88) bugs, *InPaFer* correctly filters out all incorrect patches. For 23.9% (21/88) bugs, *InPaFer* cannot generate any filtering criterion to filter out incorrect patches. For 28.4% (25/88) bugs, *InPaFer* could partially filter out incorrect patches.

To understand the reason why *InPaFer* cannot fully filter out the incorrect patches, we randomly sampled 5 bugs out of the 46 ones and manually checked the corresponding patches. The results show that the program states of these candidate patches are the same when running the failing test. For example, in Figure 4 we show a pair of candidate patches of *Chart1*, one is correct while the other is not. They both change the `if` condition (line 2) and have the same execution trace, and thus the current version of *InPaFer* is not able to distinguish them. However, *InPaFer* could be further improved by incorporating more kinds of attributes, such as invariance information [44]. We leave it to future work.

b) Number of Filtering Steps: We investigate the *number of filtering steps* needed to filter out all incorrect patches. The results show that our approach requires a median of 2.2 filtering steps (standard deviation is 2.6). There are in the median of 7 candidate patches per bug, which means without the help of *InPaFer*, developers need to go through 7 patches one by one, but with *InPaFer*, developers could just click the filtering button 2.2 times to filter out all incorrect patches.

```
1 // correct patch                    // incorrect patch
2 - if(dataset != null){              - if(dataset != null){
3 + if(dataset == null){              + if(AbsRenderer.ZERO ==
      null){
4     return result;                      return result;
5   }                                   }
```

Fig. 4: Example candidate patches left after filtering

c) Additional research questions: We also answer two additional research questions regarding the importance of different attribute types and the effect of developers' errors through controlled experiments. Due to space limitations, we put the details in the Appendix [46].

RQ1: If the answers to filtering criteria are all correct, *InPaFer* filters out all incorrect patches for 47.7% bugs with a median of 2.2 filtering steps.

B. User Study I (RQ2 & RQ3)

We answer RQ2 and RQ3 by evaluating the effectiveness and usefulness of *InPaFer* in the real debugging scenario.

1) Study Design: To investigate the impact of our approach to developers' debugging performance in practice, we design three debugging settings:

- *ManuallyFix*: repair bugs without candidate patches
- *FixWithPatches*: repair bugs with candidate patches
- *FixWithInPaFer*: repair bugs using *InPaFer*

Note that in all settings, participants are allowed to use the Eclipse debugger and run test cases as they need.

a) Tasks: Since there could be no correct patch available in the candidate patches, we design two types of tasks to reflect the two scenarios: (1) the correct patch is available, and (2) no correct patch is available. Since the design goal of *InPaFer* is to help review a group of patches, we consider only bugs with 3 or more patches. We further consider only the bugs that do not require specific domain knowledge. From the Defects4J dataset that was used in our previous experiment (see Table V), we identified 38 bugs meeting the above criteria, whose median number of patches is 20 per bug. Among them, 26 bugs contain a correct patch and 12 bugs contain no correct patch. We randomly selected 2 bugs from each category, where Task1 and Task2 contain the correct patches, while Task3 and Task4 do not (see details in Table VII). The participants would not be informed whether the candidate patch set contains a correct patch.

Please note that it is inherently difficult to increase the number of tasks in user studies and the number of tasks in our user studies is already the largest among multiple recent top-conference publications that evaluate an interactive debugging technique (1-4 tasks) [22], [47]–[49]. To mitigate the small number of tasks, we will not rely on only the statistical results of the four tasks, but also analyze the reasons of the participants' performance in different tasks. Furthermore, these bugs are representative in terms of *InPaFer*

TABLE VII: Bugs of each task in the user study

Task ID	Bug ID	#Patch (#Cor_Patch)	#Remain_Incor_Patch
Task1	Chart9	26 (4)	12
Task2	Math41	48 (1)	1
Task3	Lang14	8 (0)	0
Task4	Lang22	24 (0)	0

#Patch: number of patches, **#Cor_Patch**: number of correct patches, **#Remain_Incor_Patch**: number of remaining patches after filtering.

performance in RQ1. Among the four tasks, *InPaFer* can filter out all incorrect patches in three of them, and the filtering steps are among 2 to 4.8.

b) Pilot Study: We performed a pilot study to estimate the required power (i.e., number of participants) of our study and to tune the tasks and descriptions. We asked 10 students from computer science department to use *InPaFer* plugin on 2 tasks. We improved the UI design of *InPaFer* and the initial setup of user study with the suggestion provided by the pilot participants for the final study. The 10 pilot participants did not participate in the final study. We also measured the time and estimated that the effect size was big enough to show significant effects with few participants in the actual experiment. We had an estimation of 25 minutes when debugging without candidate patches, compared to an estimation of 20 minutes when using *InPaFer*. Based on the observation in the pilot study, we set a time limit of 30 minutes per task. We define a debugging session as successful when the bug is correctly repaired within the time limit.

c) Participants: In total, we recruited 30 participants to conduct our user study. They are all students majoring in computer science from our department. They have at least three-year programming experience and are familiar with debugging in Eclipse. Additionally, the participants have no prior knowledge of the bugs in our study.

d) Procedure: In the study, participants were evenly divided into three separate groups (i.e., A, B and C). Each participant was asked to finish four tasks under two debugging settings (as shown in Table VIII). For example, participants in Group A were asked to manually repair the bugs in Task1 and Task3 and repair the bugs in Task2 and Task4 with the help of candidate patches. To sum up, our study consisted of 120 (30×4) individual debugging sessions. This scale is significantly larger than multiple recent top-conference publications that evaluate an interactive debugging technique [22], [47]–[49], which ranges from 20 to 48 debugging sessions.

In addition, to make the participants familiar with debugging using *InPaFer*, our user study included a training task before the formal debugging session: after we described the corresponding terminology, participants in Group B and C were asked to fix an irrelevant bug using *InPaFer*. After the participants finished a debugging session, we manually checked whether the fix was correct.

At the end of each debugging session, we concluded each session with general and open-ended questions about experience and suggestions for improvement. Specifically, for each participant, we asked questions regarding the difference between the two debugging settings. For participants who have debugged with *InPaFer*, we asked questions regarding how we could improve.

TABLE VIII: Groups in user study I

Group (Participant ID)	ManuallyFix	FixWithPatches	FixWith*InPaFer*
Group A (P1-P10)	Task1+3	Task2+4	–
Group B (P11-P20)	Task2+4	–	Task1+3
Group C (P21-P30)	–	Task1+3	Task2+4

e) Analysis: We analyzed the interviews primarily qualitatively, analyzing what participants learned and how they interacted with the tool. Two of the authors transcribed and coded the interviews, following standard methods of qualitative empirical research [50].

2) Results: Participants' Performance:

a) Success Rate: We calculated the number of sessions in which the bugs were successfully repaired in three different debugging settings. Results show that our approach (*FixWithInPaFer*) significantly outperformed *ManuallyFix* and *FixWithPatches* with respectively 62.5% ($p = 10^{-4}$) and 39.3% ($p = 10^{-3}$) improvements (see Table IX). Specifically, when the correct patches existed in the candidate patches, participants could always fix the bug in the study. However, it was not the case for *FixWithPatches* though the same patches were given.

b) Debugging Time: We quantified the debugging time (see Figure 5). The results show that: comparing to *FixWithPatches*, *InPaFer* could significantly shorten the debugging time by 28.0% ($p = 4 \times 10^{-3}$) in all tasks; comparing to *ManuallyFix*, *InPaFer* could reduce the debugging time by 25.3% ($p = 10^{-5}$) for all tasks.

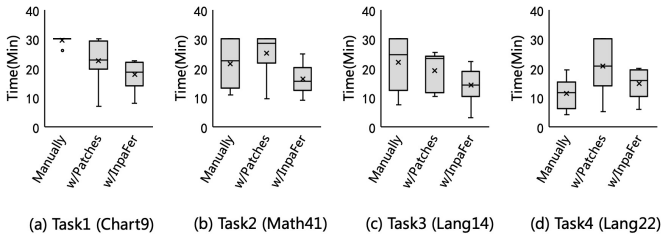
Even if all generated patches are incorrect (Task 3&4), participants in *FixWithInPaFer* still perform better than in *ManuallyFix*, repairing 27% more bugs and using 13% less time.

c) Participants' Performance and Patch Usefulness: We notice that *FixWithInPaFer* does not constantly lead to better repair performance: on Task4 participants performed slightly worse in *FixWithInPaFer* than in *ManuallyFix*. Also, the performance of *FixWithPatches* does not seem to be related to the correctness of patches: *FixWithPatches* leads to better repair performance than *ManuallyFix* on Task 1&3, where one contains the correct patch and the other one does not.

One possible explanation of this is the usefulness of patches. We observe that **even if a patch is incorrect, it may still be useful in providing hints to the participants.** Nearly all candidate patches in Task1 change the faulty code, and thus pinpoint the faulty location to the participants. For example, participant P23 said "*the patches helped me localize to the faulty code*". The patches in Task3, though all incorrect, contain part of the code in the correct patch, providing hints for the participants to figure out the correct patch. For example, participant P22 commented "*the patches provided me the API method that I should use*". According to the interview, 14 out of 20 participants (P11-P14, P17-P20, P22-P25, P29, P30) in

TABLE IX: Successful debugging sessions in user study I

Task ID	ManuallyFix	FixWithPatches	FixWith*InPaFer*
Task1	1	9	10
Task2	8	6	10
Task3	5	8	10
Task4	10	5	9
Total	**24**	**28**	**39**



(a) Task1 (Chart9) (b) Task2 (Math41) (c) Task3 (Lang14) (d) Task4 (Lang22)

Fig. 5: Debugging time in user study I

FixWithPatches and *FixWithInPaFer*, who have a chance to scan the patches, have ever mentioned that the patches in Task 1&3 helped to localize the faulty code or provide partially correct code. On the other hand, all incorrect patches in Task2 changed the code in different and wrong locations, and the incorrect patches in Task4 provided meaningless code which may mislead developers.

These observations suggest that in future we should focus on not only the correctness of the patch but also the usefulness of the patches, i.e., whether they provide useful hints to developers. Current measurement of APR techniques only considers how many correct patches are generated, here we argue that there could be a finer-grained criterion, which further considers how much help the incorrect patches provide to the developer.

Please note, even if all generated patches are incorrect and not useful, i.e., Task4, *FixWithInPaFer* still leads to close performance to *ManuallyFix*. This confirms our intuition, the filtering process in *InPaFer* helps understand the bug, and eventually contribute to the overall repair performance.

RQ2: *FixWithInPaFer* increases the success rate by 62.5% and 39.3% on average, and reduces the debugging time by 25.3% and 28.0% on average, compared to *ManuallyFix* and *FixWithPatches*, respectively. Even if all generated patches are incorrect, the performance of developers would not be significantly reduced, and could still be improved.

3) Results: Usefulness of InPaFer: We analyzed the interviews and found in multiple aspects the participants found *InPaFer* useful.

a) ***The filtering functionality helps participants to save patch review effort****:* 12 out of 20 participants (P13, P17, P20, P22-P30) mentioned that *InPaFer* could help them to quickly filter out incorrect patches. For example, P20 said "*After a few filtering steps, I only need to review a few numbers of patches, which is reliable*".

b) ***Interacting with InPaFer helps participants to understand the bug****:* Many participants generated the patch by themselves, but confirmed that the filtering process helped understand the bug. For instance, when discussing Task2, which contains a large portion of incorrect patches, P21 said " *I have no idea about the bug at the beginning, but I can confirm or reject each filtering criterion. After a few steps of filtering, I gradually understand the bug and know how to fix the bug by myself.* " Overall, there are 16 out of 40 debugging sessions for which participants mentioned that filtering patches in *InPaFer* helps understand the bug.

c) ***The UI design helps participants navigate and access key information****:* The Eclipse plugin (Figure 3) has connections between elements to help the user to overview and navigate the bugs. For example, when a concrete patch (Panel 3) is selected, the *Diff View* shows the execution trace difference before and after applying the patch. Participants gave positive feedbacks on this design. For example, P11 mentioned that when he was working on Task3, which does not contain the correct patch, "*when I chose a* `Modified Method` *criterion, the plugin located the bug to the equals function, which helps me to know that the equals function should be called. Also, the Diff View shows which branch that the execution gets into makes the test case pass. It corrects the previous misunderstanding.*"

RQ3: We found frequent and concrete evidence of the usefulness of *InPaFer*, including saving patch review effort, better understanding the bug, and accessing the key information.

C. User Study II (RQ4)

In the above user study, for all tasks, at least one participant mentioned that *InPaFer* helped them to locate the faulty code during the debugging process. Therefore, we ask:

RQ4: How does *InPaFer* compare with fault localization techniques? If participants with *InPaFer* does not show superior performance compared to participants with a fault localization technique, the latter should be preferred as it is in general more efficient without patch generation.

We conducted another user study, where participants were separated into two groups – one group debugs with the aid of *InPaFer*, while the other group debugs with a list of candidate faulty locations. Particularly, we chose the statement-level bug locations as they can provide the most comprehensive result and also are close to those provided by *InPaFer*. We used the fault localization results of the state-of-the-art technique – CombineFL [51], which achieved the best statement-level results on Defects4J benchmark as far as we know.

1) Study Design: The experimental setup was the same as user study I (Section V-B) except the controlled group and participants. We compared two debugging settings:

- *FixWithLocations*: repair bugs with a ranked list of possible faulty lines of code.
- *FixWithInPaFer*: repair bugs using *InPaFer*.

TABLE X: Groups in user study II

Group (Participant ID)	FixWithLocations	FixWith*InPaFer*
Group D (P31-P40)	Task1+3	Task2+4
Group E (P41-P50)	Task2+4	Task1+3

TABLE XI: Successful debugging sessions in user study II

Task ID	**FixWithlocations**	**FixWith*InPaFer***
Task1	2	9
Task2	8	9
Task3	10	10
Task4	7	9
Total	**27**	**37**

We recruited 20 participants, including 12 students majoring in computer science and 8 developers from companies. They have at least one-year Java programming experience and are familiar with debugging in Eclipse. Both students and developers were evenly divided into two groups (D and E). Each participant was asked to finish four tasks (same as user study I) under two debugging settings (as shown in Table X). Same as user study I, we interviewed participants after finishing the tasks to better understand their debugging process.

2) Result Analysis:

a) Succes Rate: In terms of success rates, as shown in Table XI, our approach (*FixWithInPaFer*) significantly outperformed *FixWithLocations* by 37.0% ($p = 2 \times 10^{-3}$).

b) Debugging Time: Figure 6 shows that there is no significant difference regarding the cost of time for Task 3&4 ($p > 0.1$). It is because the faulty locations suggested by CombineFL were ranked at the 1st and 2nd position correspondingly. Participants also confirmed that after checking the code in the faulty locations, it was not hard to generate the correct patches.

In Task 1&2, the faulty locations were ranked at the 13rd and 18th positions respectively, and thus could benefit little to the participants. As a result, *InPaFer* significantly shortened the debugging time by 34.4% ($p = 4 \times 10^{-4}$) comparing to *FixWithLocations*. 11 out of 20 participants said that if the top three candidate faulty locations were wrong, they would not like to go through the location list any more. While for the group of *FixWithInPaFer*, participants confirmed that *InPaFer* could provide useful information more than just the location, such as execution traces (P31, P32, P37, P38, P43), and hints from candidate patches (P32, P34, P35, P37, P38, P40, P47). The result indicates that patches provide more help than faulty locations, and the extra time for generating patches is not wasted.

RQ4: Compared to *FixWithLocations*, *FixWithInPaFer* increases the success rate by 37.0%, and reduces the debugging time by 24.2% on average.

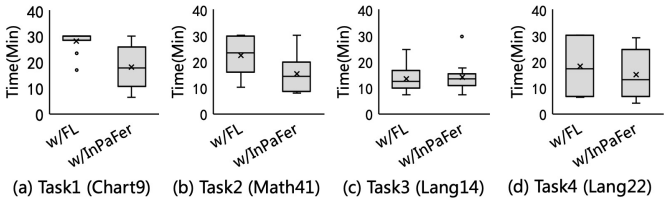


Fig. 6: Debugging time in user study II

D. Threats to Validity

Regarding *internal* validity, communication issues may have affected some answers. We mitigated this threat by refining our interview guide when questions raised confusion and involving two researchers in each interview. Despite open-ended questions and careful design, we cannot entirely exclude confirmation bias, in which participants might avoid raising critical points. We mitigated this by focusing on insights gained, not just claims.

Regarding *external* validity, first, we focused on 4 tasks in the Defects4J dataset whereas results may not be generalized to all other cases. Yet please note that it is inherently difficult to increase the number of tasks in user studies and the number of tasks in our user studies is already the largest among multiple recent top-conference publications that evaluate an interactive debugging technique (1-4 tasks) [22], [47]–[49]. To mitigate this threat, we did not rely on only the statistical results of the 4 tasks, but also analyzed the reason of the performance differences between different tasks in detail. Additionally, we argue that the 4 tasks are representative in terms of *InPaFer* performance in RQ1. Among the 4 tasks, *InPaFer* can filter out all incorrect patches in three of them, with the filtering steps among 2 to 4.8. Second, our studies may suffer from a selection bias of the participants, as common for these kinds of studies. To reduce selection bias, we recruited participants in public channels and randomly assigned them into five groups. To avoid effects due to differences between real developers who are familiar with the project and students in our user studies, we designed the tasks in a way that basic debugging experience was sufficient and did not need specific domain knowledge. Overall we selected participants with different backgrounds and did not observe any systematic differences. In total, we have conducted 200 debugging sessions for our user studies, which are signficantly more than what are used to evaluate an interactive debugging techniques (20-48 debugging sessions) in recent top-conference publications [22], [47]–[49], and we argue that a state of saturation was reached in the end. Finally, in the quantitative study, we simulated the patch filtering process under eight strategies (the other seven are shown in Appendix), but there could be cases that are not covered. To mitigate this, we run our experiment ten times. Please also note that the goal of the quantitative evaluation is to measure the effectiveness of the filtering criteria using some metrics, and the effectiveness and usefulness of *InPaFer* are evaluated by user studies.

VI. Related Work

A. Effect of Patches

Existing studies investigated the effects of providing patches to developers during debugging, including providing a single patch [19], five patches together [20], partial repair [21] or repair hints [22]. The findings of these studies are consistent: the developers' repair performance of a bug could improve when they are provided with only correct patches, but the improvement disappears and even becomes negative when the provided patches contain incorrect ones. Compared with them, our study is the first to consider the effect of tool support, and finds that good tool support could mitigate the negative effect of incorrect patches, such that the performance of developers could still improve when they are provided with incorrect patches.

B. Interactive Repair

Some recent studies tried to involve developers in the program repair process. There was a in-parallel position paper [52], which proposed an interactive patch suggestion paradigm based on *what* and *how* questions. Our work shares a vision similar to this paper, but further implements an Eclipse plugin, and conducts non-trivial user studies that reveal insights why the interactive filtering process works. Furthermore, different from our approach that asks the developers to validate attributes, the position paper proposes to ask what questions: "what should be the output of this expression?" It is yet unknown whether such what questions are easy for the developers to answer.

Cashin et al. [44] clustered a set of generated patches by program invariants. Developers ideally need only examine one patch per cluster. Compared with their approach, our approach actively asks developers to confirm or reject the filtering criteria rather than clusters the patches. These two kinds of support are orthogonal, and can be potentially combined to further support patch review in future.

Böhme et al. [45] queried developers to build a test oracle before patch generation to overcome overfitting. Compared with that, our study focuses on the patch review process after patch generation, and helping developers find a patch rather than helping the automated system.

C. Interactive Debugging

Many interactive debugging techniques [43], [47]–[49], [53]–[56] leveraged user feedback to localize faults. Algorithmic debugging [53] built a tree of method invocations for a failed test, and then repeatedly asked developers questions to prune the tree for fault localization. Li et al. [48] improved algorithmic debugging by leveraging spectrum-based fault localization (SBFL) and dynamic dependencies to optimize the order of method invocations to be questioned. Gong et al. [56] improved SBFL by asking developers to label the statements as faulty or clean, and updating the suspicious statement list. Johnson et al. [43] helped developers to understand the root cause of a bug by identifying most-similar passing tests to original failing tests. Ko and Myers proposed Whyline [49], [54], [55], which explained bug according to the dynamic slicing until the developer finds the root cause of the fault. Lin et al. [47] improved Whyline by allowing developers to select the trace execution pattern and using developer's feedback to recommend some suspicious traces.

Different from these techniques, our approach aims to help patch review by interactive patch filtering, rather than improving the fault localization. Our study indicates that patches provide more help than suspicious faulty locations.

D. Patch Correctness Identification

Multiple existing approaches automatically identified the correctness of patches. Some approaches adopted a deterministic way by generating new test cases and identifying incorrect patches which violate oracles [57] or cause crash/memory-leak errors [58], [59]. Other approaches adopted a heuristic rule, such as anti-patterns [60] or the behavior similarity of test case executions [61]. However, these techniques still cannot filter out all incorrect patches and guarantee the correctness of remaining patches. As a result, manual check will be a must. Our approach can be combined with these techniques and help developers to better understand the bug.

VII. Conclusion and Future Research Directions

In this paper, we have investigated the question: would an APR approach be useful if multiple incorrect patches are generated along with the correct patches, and proposed an interactive patch filtering approach, *InPaFer*, which contains a two-stage algorithm, to provide tool support for patch review. We have implemented our approach as an Eclipse plugin. The results show that our approach improves the repair performance of developers with 62.5% more successfully repaired bugs and 25.3% less debugging time on average. This confirms our conjecture: with proper tool support, the patch review process helps understand the bug, and eventually contributes to debugging.

Our findings open doors to multiple possibilities of future APR research directions, and we highlight two below.

(1) APR techniques could target more bugs by loosing the correctness requirement and generating more patches. Existing approaches usually try to generate a single patch per bug in order not to overwhelm the developer [28], [30], but our results suggest that with proper tool support, APR tools that generates a lot of incorrect patches can still be useful. We advocate that future APR techniques could focus on improving the recall without avoiding generating many incorrect patches. For example, they could target more bugs by expanding the search space and broadening the generation condition, and *InPaFer* ensures the mixture of incorrect and correct patches still improve the repair performance of developers.

(2) The measurement of APR techniques could consider not only the correctness, but also the usefulness of patches for developers. Our study reveals that incorrect patches can also help the developers to repair bugs, which suggests that more work is needed to fully understand and characterize patch quality. Future evaluation on APR techniques should consider not only patch correctness, but also the usefulness in providing hints to the devleopers.

REFERENCES

[1] L. Gazzola, D. Micucci, and L. Mariani, "Automatic software repair: A survey," *TSE*, vol. PP, no. 99, pp. 1–1, 2017.

[2] M. Monperrus, "Automatic Software Repair: a Bibliography," Tech. Rep., 2017. [Online]. Available: https://hal.archives-ouvertes.fr/hal-01206501/file/survey-automatic-repair.pdf

[3] https://github.com/SerVal-DTF/FL-VS-APR/tree/master/kPAR.

[4] K. Liu, A. Koyuncu, D. Kim, and T. F. Bissyandé, "AVATAR: fixing semantic bugs with fix patterns of static analysis violations," in *Proceedings of the 26th IEEE International Conference on Software Analysis, Evolution, and Reengineering*. IEEE, 2019, pp. 456–467.

[5] X.-B. D. Le, D. Lo, and C. Le Goues, "History driven program repair," in *SANER*, 2016, pp. 213–224.

[6] J. Xuan, M. Martinez, F. Demarco, M. Clément, S. Lamelas, T. Durieux, D. Le Berre, and M. Monperrus, "Nopol: Automatic repair of conditional statement bugs in java programs," *TSE*, 2017.

[7] X. D. Le, D. Chu, D. Lo, C. L. Goues, and W. Visser, "S3: syntax- and semantic-guided repair synthesis via programming by examples," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2017, Paderborn, Germany, September 4-8, 2017*. ACM, 2017, pp. 593–604.

[8] F. Long, P. Amidon, and M. Rinard, "Automatic inference of code transforms for patch generation," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2017, Paderborn, Germany, September 4-8, 2017*. ACM, 2017, pp. 727–739.

[9] M. Martinez, T. Durieux, R. Sommerard, J. Xuan, and M. Monperrus, "Automatic repair of real bugs in java: a large-scale experiment on the defects4j dataset," *Empirical Software Engineering*, vol. 22, no. 4, pp. 1936–1964, 2017.

[10] K. Wang, A. Sullivan, and S. Khurshid, "Automated model repair for alloy," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE 2018, Montpellier, France, September 3-7, 2018*. ACM, 2018, pp. 577–588.

[11] J. Hua, M. Zhang, K. Wang, and S. Khurshid, "Sketchfix: a tool for automated program repair approach using lazy candidate generation," in *Proceedings of the 2018 ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2018, Lake Buena Vista, FL, USA, November 04-09, 2018*. ACM, 2018, pp. 888–891.

[12] ——, "Towards practical program repair with on-demand candidate generation," in *Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*. ACM, 2018, pp. 12–23.

[13] M. Endres, G. Sakkas, B. Cosman, R. Jhala, and W. Weimer, "Infix: Automatically repairing novice program inputs," in *34th IEEE/ACM International Conference on Automated Software Engineering, ASE 2019, San Diego, CA, USA, November 11-15, 2019*. IEEE, 2019, pp. 399–410.

[14] T. Nguyen, W. Weimer, D. Kapur, and S. Forrest, "Connecting program synthesis and reachability: Automatic program repair using test-input generation," in *Tools and Algorithms for the Construction and Analysis of Systems - 23rd International Conference, TACAS 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings, Part I*, ser. Lecture Notes in Computer Science, vol. 10205, 2017, pp. 301–318.

[15] M. Motwani, S. Sankaranarayanan, R. Just, and Y. Brun, "Do automated program repair techniques repair hard and important bugs?" in *Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*. ACM, 2018, p. 25.

[16] A. Weiss, A. Guha, and Y. Brun, "Tortoise: interactive system configuration repair," in *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering, ASE 2017, Urbana, IL, USA, October 30 - November 03, 2017*. IEEE Computer Society, 2017, pp. 625–636.

[17] Y. Ke, K. T. Stolee, C. Le Goues, and Y. Brun, "Repairing programs with semantic code search (T)," in *30th IEEE/ACM International Conference on Automated Software Engineering, ASE 2015, Lincoln, NE, USA, November 9-13, 2015*. IEEE Computer Society, 2015, pp. 295–306.

[18] S. Saha, R. K. Saha, and M. R. Prasad, "Harnessing evolution for multi-hunk program repair," in *Proceedings of the 41st International Conference on Software Engineering, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019*. IEEE / ACM, 2019, pp. 13–24.

[19] Y. Tao, J. Kim, S. Kim, and C. Xu, "Automatically generated patches as debugging aids: a human study," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, (FSE-22), Hong Kong, China, November 16 - 22, 2014*. ACM, 2014, pp. 64–74.

[20] J. P. Cambronero, J. Shen, J. Cito, E. Glassman, and M. Rinard, "Characterizing developer use of automatically generated patches," in *2019 IEEE Symposium on Visual Languages and Human-Centric Computing, VL/HCC 2019, Memphis, Tennessee, USA, October 14-18, 2019*. IEEE Computer Society, 2019, pp. 181–185.

[21] J. Yi, U. Z. Ahmed, A. Karkare, S. H. Tan, and A. Roychoudhury, "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, ESEC/FSE 2017, Paderborn, Germany, September 4-8, 2017*. ACM, 2017, pp. 740–751.

[22] S. Kaleeswaran, V. Tulsian, A. Kanade, and A. Orso, "Minthint: automated synthesis of repair hints," in *36th International Conference on Software Engineering, ICSE '14, Hyderabad, India - May 31 - June 07, 2014*. ACM, 2014, pp. 266–276.

[23] F. Long and M. Rinard, "An analysis of the search spaces for generate and validate patch generation systems," in *ICSE*, 2016, pp. 702–713.

[24] E. K. Smith, E. T. Barr, C. Le Goues, and Y. Brun, "Is the cure worse than the disease? overfitting in automated program repair," in *FSE*, 2015, pp. 532–543.

[25] X. D. Le, F. Thung, D. Lo, and C. Le Goues, "Overfitting in semantics-based automated program repair," in *Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*.

[26] Z. Qi, F. Long, S. Achour, and M. Rinard, "An analysis of patch plausibility and correctness for generate-and-validate patch generation systems," ser. ISSTA, 2015, pp. 24–36.

[27] A. Roychoudhury and Y. Xiong, "Automated program repair: a step towards software automation," *Sci. China Inf. Sci.*, vol. 62, no. 10, pp. 200 103:1–200 103:3, 2019. [Online]. Available: https://doi.org/10.1007/s11432-019-9947-6

[28] Y. Xiong, J. Wang, R. Yan, J. Zhang, S. Han, G. Huang, and L. Zhang, "Precise condition synthesis for program repair," in *ICSE*, 2017.

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

[30] J. Jiang, Y. Xiong, H. Zhang, Q. Gao, and X. Chen, "Shaping program repair space with existing patches and similar code," in *ISSTA*, 2018.

[31] Q. Xin and S. P. Reiss, "Leveraging syntax-related code for automated program repair," ser. ASE, 2017. [Online]. Available: http://dl.acm.org/citation.cfm?id=3155562.3155644

[32] R. K. Saha, Y. Lyu, H. Yoshida, and M. R. Prasad, "Elixir: Effective object oriented program repair," in *ASE*. IEEE Press, 2017. [Online]. Available: http://dl.acm.org/citation.cfm?id=3155562.3155643

[33] R. Just, D. Jalali, and M. D. Ernst, "Defects4j: A database of existing faults to enable controlled testing studies for java programs," in *ISSTA*, 2014, pp. 437–440.

[34] M. Martinez and M. Monperrus, "Ultra-large repair search space with automatically mined templates: The cardumen mode of astor," in *Search-Based Software Engineering - 10th International Symposium, SSBSE 2018, Montpellier, France, September 8-9, 2018, Proceedings*, ser. Lecture Notes in Computer Science, vol. 11036. Springer, 2018, pp. 65–86.

[35] C. Le Goues, T. Nguyen, S. Forrest, and W. Weimer, "Genprog: A generic method for automatic software repair," *TSE*, vol. 38, no. 1, pp. 54–72, Jan 2012.

[36] M. Martinez and M. Monperrus, "Astor: A program repair library for java," in *Proceedings of ISSTA*, 2016.

[37] Z. P. Fry, B. Landau, and W. Weimer, "A human study of patch maintainability," in *International Symposium on Software Testing and Analysis, ISSTA 2012, Minneapolis, MN, USA, July 15-20, 2012*. ACM, 2012, pp. 177–187.

[38] W. Weimer, T. Nguyen, C. Le Goues, and S. Forrest, "Automatically finding patches using genetic programming," in *ICSE*, 2009, pp. 364–374.

[39] K. Liu, A. Koyuncu, T. F. Bissyandé, D. Kim, J. Klein, and Y. L. Traon, "You cannot fix what you cannot find! an investigation of fault localization bias in benchmarking automated program repair systems," in *12th IEEE Conference on Software Testing, Validation and Verification, ICST 2019, Xi'an, China, April 22-27, 2019*. IEEE, 2019, pp. 102–113.

[40] D. Kim, J. Nam, J. Song, and S. Kim, "Automatic patch generation learned from human-written patches," in *ICSE*, 2013, pp. 802–811.

[41] Z. Chen and M. Monperrus, "The remarkable role of similarity in redundancy-based program repair," *CoRR*, vol. abs/1811.05703, 2018. [Online]. Available: http://arxiv.org/abs/1811.05703

[42] M. White, M. Tufano, M. Martinez, M. Monperrus, and D. Poshyvanyk, "Sorting and transforming program repair ingredients via deep learning code similarities," *ArXiv e-prints*, Jul. 2017.

[43] B. Johnson, Y. Brun, and A. Meliou, "Causal testing: Finding defects' root causes," in *Proceedings of the 42th International Conference on Software Engineering, ICSE 2020, Seoul, South Korea, October 5-11, 2020*, 2020.

[44] P. Cashin, C. Martinez, W. Weimer, and S. Forrest, "Understanding automatically-generated patches through symbolic invariant differences," in *34th IEEE/ACM International Conference on Automated Software Engineering, ASE 2019, San Diego, CA, USA, November 11-15, 2019*. IEEE, 2019, pp. 411–414.

[45] M. Böhme, C. Geethal, and V. Pham, "Human-in-the-loop automatic program repair," in *13th IEEE International Conference on Software Testing, Validation and Verification, ICST 2020, Porto, Portugal, October 24-28, 2020*. IEEE, 2020, pp. 274–285.

[46] GitHub, "Inpafer's repository that contains the implementation and the data used in this study." https://github.com/Emilyaxe/InPaFer-ICSME, 2021.

[47] Y. Lin, J. Sun, Y. Xue, Y. Liu, and J. S. Dong, "Feedback-based debugging," in *Proceedings of the 39th International Conference on Software Engineering, ICSE 2017, Buenos Aires, Argentina, May 20-28, 2017*. IEEE / ACM, 2017, pp. 393–403.

[48] X. Li, S. Zhu, M. d'Amorim, and A. Orso, "Enlightened debugging," in *Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*. ACM, 2018, pp. 82–92.

[49] A. J. Ko and B. A. Myers, "Debugging reinvented: asking and answering why and why not questions about program behavior," in *30th International Conference on Software Engineering (ICSE 2008), Leipzig, Germany, May 10-18, 2008*. ACM, 2008, pp. 301–310.

[50] J. Saldaña, *The coding manual for qualitative researchers*. Sage, 2015.

[51] D. Zou, J. Liang, Y. Xiong, M. D. Ernst, and L. Zhang, "An empirical study of fault localization families and their combinations," *IEEE Transactions on Software Engineering*, vol. PP, no. 99, 2018.

[52] X. Gao and A. Roychoudhury, "Interactive patch generation and suggestion," in *Proceedings of the 1st International Workshop on Automated Program Repair*. IEEE / ACM, 2020.

[53] R. Caballero, A. Riesco, and J. Silva, "A survey of algorithmic debugging," *ACM Comput. Surv.*, pp. 60:1–60:35, 2017.

[54] A. J. Ko and B. A. Myers, "Finding causes of program output with the java whyline," in *Proceedings of the 27th International Conference on Human Factors in Computing Systems, CHI 2009, Boston, MA, USA, April 4-9, 2009*. ACM, 2009, pp. 1569–1578.

[55] ——, "Designing the whyline: a debugging interface for asking questions about program behavior," in *Proceedings of the 2004 Conference on Human Factors in Computing Systems, CHI 2004, Vienna, Austria, April 24 - 29, 2004*. ACM, 2004, pp. 151–158.

[56] L. Gong, D. Lo, L. Jiang, and H. Zhang, "Interactive fault localization leveraging simple user feedback," in *28th IEEE International Conference on Software Maintenance, ICSM 2012, Trento, Italy, September 23-28, 2012*. IEEE Computer Society, 2012, pp. 67–76.

[57] Q. Xin and S. Reiss, "Identifying test-suite-overfitted patches through test case generation," in *ISSTA*, 2017, pp. 226–236.

[58] J. Yang, A. Zhikhartsev, Y. Liu, and L. Tan, "Better test cases for better automated program repair," in *FSE*, 2017, pp. 831–841.

[59] X. Gao, S. Mechtaev, and A. Roychoudhury, "Crash-avoiding program repair," in *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2019, Beijing, China, July 15-19, 2019*. ACM, 2019, pp. 8–18.

[60] S. H. Tan, H. Yoshida, M. R. Prasad, and A. Roychoudhury, "Anti-patterns in search-based program repair," in *FSE*, 2016.

[61] Y. Xiong, X. Liu, M. Z. andz Lu Zhang, and G. Huang, "Identifying patch correctness in test-based program repair," in *Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*. ACM, 2018, pp. 789–799.