Automated Program Repair, What Is It Good For? Not Absolutely Nothing!

Hadeel Eladawy
University of Massachusetts
Amherst, MA, USA
heladawy@umass.edu

Claire Le Goues
Carnegie Mellon University
Pittsburgh, PA, USA
clegoues@cs.cmu.edu

Yuriy Brun
University of Massachusetts
Amherst, MA, USA
brun@cs.umass.edu

ABSTRACT

Industrial deployments of automated program repair (APR), e.g., at Facebook and Bloomberg, signal a new milestone for this exciting and potentially impactful technology. In these deployments, developers use APR-generated patch suggestions as part of a human-driven debugging process. Unfortunately, little is known about how using patch suggestions affects developers during debugging. This paper conducts a controlled user study with 40 developers with a median of 6 years of experience. The developers engage in debugging tasks on nine naturally-occurring defects in real-world, open-source, Java projects, using Recoder, SimFix, and TBar, three state-of-the-art APR tools. For each debugging task, the developers either have access to the project's tests, or, also, to code suggestions that make all the tests pass. These suggestions are either developer-written or APR-generated, which can be correct or deceptive. Deceptive suggestions, which are a common APR occurrence, make all the available tests pass but fail to generalize to the intended specification. Through a total of 160 debugging sessions, we find that access to a code suggestion significantly increases the odds of submitting a patch. Access to correct APR suggestions increase the odds of debugging success by 14,000% as compared to having access only to tests, but access to deceptive suggestions decrease the odds of success by 65%. Correct suggestions also speed up debugging. Surprisingly, we observe no significant difference in how novice and experienced developers are affected by APR, suggesting that APR may find uses across the experience spectrum. Overall, developers come away with a strong positive impression of APR, suggesting promise for APR-mediated, human-driven debugging, despite existing challenges in APR-generated repair quality.

KEYWORDS

automated program repair, debugging, human factors, user study

ACM Reference Format:
Hadeel Eladawy, Claire Le Goues, and Yuriy Brun. 2024. Automated Program Repair, What Is It Good For? Not Absolutely Nothing!. In *2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE '24), April 14–20, 2024, Lisbon, Portugal.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3597503.3639095

1 INTRODUCTION

In 2020, operational software failures cost the U.S. economy $1.56 trillion [34]. Software debugging, testing, and verification costs account for more than $100 billion annually, between half and three quarters of the total software development budgets, taking up between a third and half of software engineers' time [60]. Automated program repair (APR) can potentially significantly improve this situation by automatically producing patches for defects identified either via bug reports or failing tests, or both [38].

APR, by many measures, has been highly impactful. Hundreds of research papers have been written on the topic [17, 49, 50, 63] and it has been deployed successfully in industry, including at Facebook and Bloomberg [3, 31, 44, 58]. However, APR is far from omnipotent. Studies have outlined numerous shortcomings that prevent wider adoption. First, test-based APR cannot even be applied to 90% of industrial defects because the necessary bug-evidencing tests do not exist prior to a human fixing the defect [58]. Second, APR can produce patches that break untested or undertested functionality [54, 65, 69]. By some measures, fewer than 5% of the defects are actually patched correctly, when published APR tools are applied to datasets other than the ones in their original evaluations [9, 51]. Third, only 7.7% of industrial defects' human-written patches are sufficiently localized to be considered the target of most state-of-the-art APR tools [58].

As a result, at Facebook, APR is a part of what is ultimately, a human-driven debugging process. Developers act as oracles of patch appropriateness for both internally deployed Getafix [3] and SapFix [44] APR tools. However, for some categories of bugs, only 12% of the patches Getafix produces are the same as the patches written by humans (although that rate goes up to 91% for some other categories) [3]. Further, developers judged that only 48% of the SapFix-produced patches correctly repaired the defects, and they still manually modified half of those correct patches [44]. Overall, while these tools can speed up the process of getting patches into production, it is not clear if that effect is simply a result of the tools bringing these particular defects to the developers' attention. On the bright side, developers are quite receptive to using APR: both Getafix and SapFix were positively received at Facebook [3, 44], and a large-scale survey of professionals suggests that developers are open-minded to working with APR tools even if they often produce low-quality suggestions [59].

These observations raise a central question at the heart of understanding the value of APR:

Does the availability of automatically produced patches help developers debug?

The answer to this question is far from obvious. As an analogy, consider the case of automated fault localization, a.k.a. automated debugging, which for more than 40 years has tackled the problem of automatically determining which source code lines are responsible for a defect [27, 82, 83]. Thirty years into that effort, controlled user studies showed that, in fact, being told the results of state-of-the-art fault localization techniques does not help non-expert developers fix defects faster [61] and can even weaken programmers' abilities in fault detection [88].[1] However, in industrial settings and for expert developers, automated debugging can accelerate fixing defects [61, 87]. The question with APR is similar. Intuition suggests that having access to potential patches should speed up debugging. But it is also possible that access to ready-made patches may reduce developer understanding of the defect, and that even just the possibility that the patch might be incorrect may derail and delay the manual repair process. Further, incorrect patches might increase the chance that developers deploy lower quality solutions than if they had no APR help at all. Overall, access to APR patches could help developers, but it could instead reduce their confidence in the solution or reduce the quality of the solution and of the codebase more generally.

We designed and executed a user study with 40 graduate and senior undergraduate college student developers, each performing four debugging tasks on real-world defects from real-world open-source systems. At the start of each task, the developer was either shown no suggestion for how to fix the defect, or given access to one of three types of suggestions: a correct developer-written suggestion, a correct APR-generated suggestion, or a deceptive APR-generated suggestion that passed all the developer-written tests in the project but failed to fully repair the underlying defect (and so would fail some other tests). The developers did not know where the suggestion came from, if it was human-written or computer-generated, and if it was correct. All suggestions passed all the developer-written tests, as that is the goal of APR. We generated the suggestions using three state-of-the-art APR tools that represent three most popular APR approaches: TBar [39], SimFix [25], and Recoder [98]. We then measured whether the patch each participant submitted, if any, was correct, incorrect, or plausible (passed the project's tests but failed to generalize to the intended behavior) and the time the debugging task took.[2]

Our study answers five research questions:

RQ1: How do code suggestions affect debugging success?
Answer: Access to a code suggestion makes participants much more likely to submit a patch. A correct suggestion helps successfully repair defects (93.3% of participants did so). Access to a deceptive suggestion hurts that ability (67.5% of participants submitted overfitting patches in these situations).

RQ2: How does APR affect debugging success?
Answer: Access to a correct APR suggestion increases participant debugging success by 14,000%, compared to having access to only tests. Meanwhile access to a deceptive APR suggestion decreases that success by 65%. Correct suggestions are far more likely to help produce correct patches than deceptive suggestions are to hinder producing them.

RQ3: How do code suggestions affect debugging time?
Answer: Access to correct suggestions speeds up debugging, as compared to access to deceptive suggestions or no suggestions at all.

RQ4: How can the effect of deceptive suggestions be mitigated?
Answer: Participants recognize that deceptive suggestions are of lower quality than correct ones, but 67.5% of the participants still submit an overfitting patch when given a deceptive suggestion. Surprisingly, we find no evidence that programming experience improved the ability to recognize deceptive suggestions, submit patches, submit correct patches, or debug faster.

RQ5: How does APR use affect behavior?
Answer: Access to a correct suggestion helps participants understand the defects. Participants overwhelmingly like having access to suggestions and, on average, rate the likelihood they would use APR in the future as 9 out of 10.

Our study focuses on whether APR helps developers debug. Unfortunately, APR research has rarely tackled this question. A 2023 study found that while many papers motivate APR research as helpful to developers, fewer than 7% include an actual evaluation with users, and some of those included in the 7% consist of a one-participant user study, while many focused on using developers to evaluate APR output, rather than how developers would be affected by it [85]. The closest study [73] to ours was conducted a decade ago, at the sunrise of APR research, and used patches produced by GenProg [37, 81] and Par [30], two of the earliest APR tools. That study applied GenProg and Par to five defects and used 44 students, 28 engineers, and 23 crowdsourced workers to evaluate whether access to patches helped improve developer debugging. The study classified patches into low- and high-quality based on human judgement, which conflated correctness, maintainability, and other factors [73]. By contrast, since then, more precise methodologies for evaluating patch quality have emerged [35, 54, 65, 69], and hundreds of more effective APR tools have been created [17, 49, 50, 63]. Our study uses a representative set of three modern tools, and evaluates both the APR code suggestions and the participant-written patches using the modern methodologies. Other studies have looked at how developers deal with multiple code suggestions presented at once and how they use suggestions [7], and have surveyed professionals to understand what issues impact their trust in APR [59]. These studies complement our work.

All questionnaires used in the study, data, and code to replicate our analyses are available at: http://doi.org/10.17605/OSF.IO/9JZHR

The remainder of this paper is structured as follows. Section 2 describes APR background. Sections 3 and 4 detail our study methodology and results, respectively. Section 5 places our work in the context of related research. Section 6 summarizes our contributions.

2 AUTOMATED PROGRAM REPAIR

APR tackles the problem of automatically producing a patch for a defect, given some evidence of that defect. The most common form of APR requires a set of program tests, some of which fail, evidencing

[1]While this conclusion is counterintuitive, its possible explanations include that fault localization tools "ignore [and fail to support] the fact that understanding the root cause of a failure typically involves complex activities, such as navigating program dependencies and rerunning the program with different inputs" [61] and "interference between the mechanism of automated fault localization and the actual assistance needed by programmers in debugging" [88].

[2]The experimental protocol was approved by the UMass Amherst IRB, protocol #4298.

the defect (although some techniques have used bug reports, or bug reports and tests together [33, 53], or contracts [79]). The problem APR solves is, fundamentally, searching through a space of possible programs for one that satisfies some behavioral specification. The two key observations that make APR possible are that (1) when humans repair defects, they tend to make a small number of changes, so the search process can be confined to programs similar to the buggy program, and (2) fault localization [27, 82, 83] can identify the lines of code most likely responsible for the defect. These two observations can significantly narrow the search space, leading APR tools to sometimes produce patches [38]. APR approaches typically use the program's tests to validate the program variants they produce. If a variant passes all tests, it is considered to patch the defect. Hundreds of APR techniques have emerged over the last decade and a half, described in several recent surveys [17, 49, 50, 63]. Section 5 discusses these techniques in more detail, including how they produce variants.

While APR approaches only report patches if they pass all the tests available during repair, because tests and specifications are typically partial, the patches APR produces are not guaranteed to be correct [54, 65, 69]. Patches that pass all tests are known as *plausible* [65], whether they are correct or not. And patches that pass all the tests but do not properly repair the defect are said to *overfit* to the tests [54, 69]. Overfitting in APR has become "the grand challenge in today's research on APR" [38].

Our study uses three representative, state-of-the-art APR tools, Recoder [98], SimFix [25], and TBar [39], as Section 3.3 will justify.

Terminology: In this paper, we use the term "suggestion" to refer to the developer-written or APR-generated code participants receive during debugging. Suggestions that properly implement the intended behavior are "correct," but those that pass all the project's tests while failing to generalize to the intended behavior are "deceptive." The participants produce "patches" as a result of their debugging, which can be "correct," "incorrect" if they fall at least one of the project's tests, or "overfitting" if they pass the tests but fail to generalize to the intended behavior.

3 STUDY DESIGN

We performed a user study with 40 participants to measure the effects code suggestions on debugging. This section describes our study design and our data analysis methodology.

3.1 Research Questions

Our study aims to answer whether APR can help developers repair defects. Of course, if APR patches were always correct, the answer would be trivial, but a key challenge with APR is that it often produces incorrect patches [54],[3] which is precisely why using APR output as suggestions for a manual repair process is a promising approach — one industry undertakes today [3, 44]. However, because APR sometimes produces deceptive suggestions, access to APR may hurt or may help debugging, in the large. Accordingly, our user study aims to answer five research questions to shed light on to how APR, and code suggestions in general, can affect debugging:

First, **RQ1** asks how code suggestions (whether developer-written or APR-generated) can affect debugging success. Next, we focus on APR, with **RQ2** asking how APR affects debugging success, as part of an analysis of the potential impact of APR as part of a tool-mediated but ultimately human-driven debugging process. Then, we study debugging speed, with **RQ3** asking how code suggestions affect debugging time. Next, **RQ4** investigates the possibility of mitigating the effect of deceptive suggestions. Finally, **RQ5** explores the qualitative side of how APR affects developer behavior, including whether it causes developer to pretend to understand code they do not understand, and whether the developers' experience in our study left a favorable impression of APR.

3.2 Participants

For our experiments, we had a total of 40 participants. All the participants were either graduate or senior undergraduate students in a Computer Science department. Of the participants, 25 self-described as male and 15 as female. Due to a survey execution error, we do not have age data for 13 of the 40 participants. For the data properly collected, the participants age varied from 21 to 34, with a median of 24.0 and standard deviation of 2.39. Their programming experience varied from 1 to 41 years, wth a median of 6.0 and standard deviation of 6.49. Experience with Java varied from 0 to 8 years, with a median of 1.75 years and standard deviation of 2.17 years. The participants were students in two software engineering courses at a university. None had prior experience with APR.

3.3 APR tools

We selected three APR tools to satisfy the following criteria. Each tool must represent state-of-the-art of modern APR, including in terms of the quality of the code suggestions they produce. The tools together must represent the three types of heuristic-based APR tools, machine-learning-based, low-level-transformation-based, and template-based (recall Section 2). Each tool must have been evaluated on the Defects4J dataset [28] (see Section 3.4) and the code suggestions it produced for those defects, both correct and deceptive, must be publicly available.

Our criteria led to the following three APR tools. Recoder[4] [98] is a machine-learning tool that learns from a dataset of historical bug fixes. Recoder uses neural machine translation to formulate the APR problem as translating buggy code sequences into a correct code sequences. SimFix [25] uses low-level code mutations and transformations from existing patches to formulate a search space, and then intersects that search space with code snippets similar to the buggy code. Finally, TBar [39] uses a set of predefined repair templates.

3.4 Defects

Our study had participants undertake debugging tasks consisting of fixing a defect in a software system. We used the Defects4J dataset of

[3]In fact, humans, when debugging, can, similarly to APR, produce patches that pass tests but fail to generalize to the intended specification, as has been observed in prior work [69, 95], and in the course of our study (Figure 2 will show that the participants submitted 11 overfitting patches even when they were not given a deceptive suggestion.)

[4]While the Recoder evaluation [98] unfortunately did not make Recoder's deceptive suggestions available, the authors later identified that several of the generated code suggestions the evaluation marked as correct were, in fact, deceptive: https://github.com/pkuzqh/Recoder/blob/master/Result/out.

defectID		description	correct			deceptive		
			Recoder	SimFix	TBar	Recoder	SimFix	TBar
Chart	12	Misuse of a setter method.	✗	✓	✗	✗	✗	✓
Lang	10	Unnecessary whitespace-handling code needs to be deleted.	✗	✗	✓	✗	✓	✗
Lang	39	Missing nullness check results in a `NullPointerException`.	✗	✓	✗	✗	✗	✓
Lang	51	Missing `return` in a `switch` statement.	✓	✗	✗	✗	✗	✓
Math	33	Use of a wrong variable as an argument.	✓	✓	✗	✗	✗	✓
Math	50	Unnecessary assignment code needs to be deleted.	✓–	✓	✗	✗	✗	✓
Math	63	Wrong comparison used for NaN values.	✗	✓	✗	✗	✗	✓
Math	96	Wrong comparison used for complex numbers.	✓	✗	✗	✗	✗	✓
Mockito	29	Missing nullness check results in a `NullPointerException`.	✓	✗	✗	✗	✗	✓

Figure 1: Summary of the nine defects from the Defects4J benchmark used in our study. Note that while both Recoder (denoted ✓–) and SimFix produce (different) correct suggestions for Math 50, our study only uses the SimFix' suggestion.

real-world defects from open-source software systems that occurred during the natural development process [28]. Defects4J v1.2 consists of 395 defects. Each defect contains the code, a set of tests at least one of which fails because of the defect, and a minimized set of changes made by the project's developers that make all the tests pass. Defects4J is a commonly used benchmark to evaluate APR.

Our criteria for selecting a subset of the defects for our study included: (1) at least one of our three APR tools produced a correct code suggestion to repair the defect, (2) at least one of our three APR tools produced a deceptive code suggestion for the defect, (3) the defect was not so trivial that it could be fixed in under a minute without help, (4) the defect was not so complex that it could not be fixed in 30 minutes, and (5) each of our study's three APR tools should contribute suggestions (whether correct or deceptive) for a nontrivial portion of the defects. We judged defect complexity manually. We excluded as too complex defects that spanned multiple files, methods, or locations within a single file. We manually judged whether the defect would take more than a minute but could be fixed in less than 30 minutes by attempting to repair the defect ourselves.

Starting with Defects4J's 395 defects, we selected the defects for which at least one of our APR tools generated at least one code suggestion, resulting in 119 defects. We then filtered that list to contain defects with one correct suggestion and at least one deceptive suggestion, resulting in 15 defects. Next, we filtered out those defects whose correct APR suggestions were different from the developer suggestions, leaving 9 defects. However, we encountered difficulty running one of those, Closure 115, on modern hardware, so we replaced it with an alternate, Math 33, whose correct APR suggestion was identical to the developer suggestion. We then verified that the selected 9 defects were neither trivial nor too complex, and that each APR tool produced suggestions for a nontrivial portion of the defects.

Figure 1 summarizes the nine defects used in our study, and the three APR tools' performance on these defects. Every defect has at least one of our three APR tools generate a correct code suggestion, and at least one generate a deceptive code suggestion. We retrieved all of the code suggestions from the original tool's released evaluation data packages [25, 39, 98]. The developer-written code suggestions come from the Defects4J's developer repairs.

For one defect, (Math 50), two tools generated two different correct code suggestions; we selected one at random (the one we did not select is denoted ✓–) in Figure 1). Our goal was to ensure that every selected defect had at least one correct and at least one deceptive code suggestion generated by an APR tool, and not to ensure a balanced distribution of the three tools across the two kinds of suggestions. (To do so would have been too restrictive for this dataset.)

3.5 Debugging Tasks

At the start of the study, the participant was asked to fill out a consent form explaining the risks and benefits of the study and giving them the option to opt out of having their data be used by the researchers. Then, the study administrator introduced the participant to the environment, and instructed them perform four debugging tasks, sequentially. Each debugging task lasted 30 minutes, though the participant could end it earlier. While we asked participants to stop at 30 minutes, some worked a little longer before stopping. We further allowed participants who had left over time from some tasks to return to earlier, unfinished tasks. The participants used Eclipse for their debugging tasks. When the participant started, an Eclipse workspace had already been prepared, containing four projects for four randomly selected defects from our nine study defects. Within the Eclipse projects, the participant had access to all of the project's unit tests and source code (though the failing unit tests quickly led the participants to the relevant method.) The administrator told the participants that "The defect is in the code, so you may change any of the source code files, but you can assume the tests are correct." Participants were allowed to use the Internet. We blocked access to certain pages that display Defects4J defects and the developer-written patches [71].

The study administrator provided a handout with a brief description of the projects, and screenshots that showed how to run and explore tests, and the difference between a failure and an error in the JUnit window in Eclipse. The study administrator walked through the handout with each participant.

Each participant was asked to fix the given defect within 30 minutes, though as described above, some took longer. At the end of each debugging task, we asked the participant to fill out an online survey. We considered the time between when the participant first expanded the project until starting the survey as the debugging time for that task. The survey asked for responses to four prompts: "Explain briefly the cause of the bug you just worked on. If you could not identify the cause, say that.", "Explain briefly how you repaired the bug. If you could not repair it, say that.", "If you were given a suggested patch when you started debugging this bug, describe whether this patch was helpful or not helpful in your debugging.", and "If you were given a suggested patch when you started debugging this bug, how would you describe the patch's quality on a scale of 0 to 10?"

Each of the four tasks was part of one of four treatments. In the "no suggestion" treatment, the participant received no additional information. In the other three treatments, the participant received a code suggestion. The participants were told that "the code suggestion may or may not help you fix the defect. You are free to use these suggestions however you wish." They were also told that "a correct fix will make all tests pass. But passing all the tests does not necessarily mean that the bug is fixed." In the "developer suggestion" treatment, the suggestion was the projects' minimized, developer-written defect repair. In the "APR correct suggestion" treatment, the suggestion was generated by one of our APR tools and correctly repaired the defect. Finally, in the "APR deceptive suggestion" treatment, the suggestion was generated by one of our APR tools but did not correctly repair the defect, though passed all of the projects' tests. (For brevity, we omit "suggestion" from the treatment names.)

The participants were not told what kind of code suggestions they received, nor whether it was correct. They could apply it and run the projects tests, though all suggestions made all the tests pass, by design.

After finishing the four debugging task sessions, each participant was asked to fill out an exit survey, which asked for demographic information, experience in programming, Java, and experience using Eclipse and JUnit. Finally, they were asked three multiple choice questions: "On a scale of 0 to 10, overall, what did you think of the quality of the suggested patches you were given, across all defects?", "How would you describe your overall experience using the suggested patches when debugging?", and "If you had access to a tool that automatically generated suggested patches like these for you, how likely are you to use such a tool in your everyday programming tasks, on a scale of 0 to 10?"

Prior to the study, we ran a small-scale pilot with two subjects to ensure the study ran smoothly and the instructions were clear. These extra subjects' data are not included in this paper.

For each participant, we randomized the choice of the four of the nine defects, the treatment used for each defect, and the order in which the participant saw the defects (and, thus, the treatments).

3.6 Data Collection

All the experiments were screen-recorded (with consent of the participants), and we used the recordings to conduct measurements for the study, such as the time to completion of each task. The final Eclipse workspaces for the debugging tasks were also saved, and we used the final patches participants produced to measure patch quality and correctness.

Overall, our 40 participants performed 4 tasks, each, for a total of 160 debugging tasks completed. However, due to a property of the code suggestions, our results contain data for 169 debugging task measurements. All the correct suggestions generated by the three tools for our nine defects were different from the developer-written suggestions except for Math 33. Accordingly, the data for the four participants in the "APR correct" treatment and the five participants in the "developer" treatment for Math 33 were treated as both "APR correct" and "developer" treatments. Thus, in the data, it appears as if these nine participants worked on five tasks, each. This results in data for 45 "developer" and 44 "APR correct" treatment instances, instead of the expected 40, and a total of 169 tasks.

Out of 169 attempts, our study generated 150 total participant-written patches; 19 attempts resulted in participants generating no patch. We evaluated the patches written by the participants following the state-of-the-art patch evaluation methodology [54] that combines manual inspection (which has been empirically shown to be subjective and biased [35]) and objective independent-test-suite-based assessment (which, on its own, can be incomplete [35]). We first manually compared and marked as "correct" the participants' patches that were identical to the developers' suggested repairs or the APR-generated correct suggestions. We next manually compared and marked as "overfitting" the participants' patches that were identical to the APR-generated deceptive suggestions. We then ran the developer tests included in the project on the remaining participant patches and marked as "incorrect" those that failed at least one test. For the remaining patches (which passed all the tests included in their projects) we used independently generated, high-coverage test suits to check if the participants' patches failed any of these tests, marking those that failed at least one test as "overfitting". For seven of the nine defects, we used the test suites generated by prior work [54]. Two defects, Math 96 and Mockito 29, did not have previously generated tests (because they were not repaired by APR tools prior to the state-of-the-art ones we use in our paper), and we followed the prior work's methodology [54] to generate the test suites. The fraction of passing, independent tests denoted the quality of the participants' patches. Finally, we manually examined the remaining six participant patches and reasoned about their correctness, attempting to generate behavior not covered by the independent test suites that might differ from the developer-written correct repair suggestion, and making an ultimate decision on patch correctness as "correct" or "overfitting". (Figure 2, which Section 4.1 will describe, reports our patch classification results.)

3.7 Analysis methodology

RQ1–RQ4 evaluate quantifiable measurements surrounding participant performance on debugging tasks under the four treatment conditions. We measure the following four response variables related to participants' debugging success:

- **Completion**: (binary variable) Whether the participant submitted a patch.
- **Correctness**: (categorical variable) As described in Section 3.6, we classified each participant's patch as "correct," "overfitting," or "incorrect." As a reminder, overfitting patches pass all of its project's developer-written tests, but nonetheless exhibit incorrect behavior.

		Participant Patch			
		Correct	Overfitting	Incorrect	None
Suggestion	no suggestion	15	6	4	15
	deceptive APR	8	27	2	3
	correct APR	44	0	1	0
	developer	39	4	0	1

Figure 2: Counts and types of the participant-written patches, by suggestion type.

- **Debugging time**: (continuous variable) The amount of time that elapsed from the moment the participant expanded the project for a debugging task and the moment they started filling out the post-task survey form for that task.

We additionally examined the quantitative and open-ended survey responses to identify interesting trends in the participants' behavior, which we examine in RQ5. For example, the participants' explanations of the cause of the defect revealed insights into whether they understood the debugging task, and their rating of the likelihood that they would use an automated tool that generates coding suggestions in the future represented an overall measure of the quality of their debugging experience using APR.

In general, we aim to compare the distribution of dependent variables across the four treatment conditions. Our study design imposes constraints on these analyses. We must control both for the participants, as their debugging and programming skills will vary, and the debugging task difficulty, as some of the defects are likely to be more difficult to repair than others. There are multiple observations per participant, and the observations are thus not independent, and there may be learning effects. We therefore use mixed-effect multivariate regression models for our analyses and treat the participants and debugging tasks as random effects. We account for learning effects by including, along with the treatment, the number of patch attempts so far (0–3), and the interaction between number of attempts so far and treatment, as independent variables. We discuss other analyses particular to specific research questions in context.

4 STUDY RESULTS

We now analyze our study results to answer our five research questions.

4.1 RQ1: How Do Code Suggestions Affect Debugging Success?

Our first question focused on whether seeing code suggestions, developer-written or APR-generated, affected participants' success in their debugging tasks. Recall that each participant performed four tasks, one per treatment, in a randomized order. Thus, we first checked if the task order affected participants' success. A mixed-effects multivariate logistic regression model including order as a predictor (recall Section 3.7) showed that neither it nor its interaction with treatment contributed in a statistically significant way to predicting whether a participant correctly patched a defect. This suggests that subjects did not necessarily get better at fixing bugs with practice. We therefore follow standard practice and exclude order from the final models we built for this research question.

	Participant Patch			
	Correct		Overfitting	
	Coeff.	p-value	Coeff.	p-value
correct APR	4.9839	< 0.0001	-19.22	0.997
developer	2.7721	0.0002	-0.3724	0.595
deceptive APR	-1.0226	0.0602	2.82	< 0.0001
intercept	-0.5659	0.243	-1.9661	< 0.0001
χ^2	27.519	< 0.0001	36.506	< 0.0001
fit	R^2_m 0.559	R^2_c 0.682	R^2_m 0.953	R^2_c 0.961

Figure 3: Mixed-effect multivariate logistic regression model parameters for predicting the participant submitting a correct patch (left) and an overfitting patch (right). For correct patches, the positive coefficients are statistically significant for correct APR and developer treatments, at $\alpha = 0.05$. For the deceptive APR treatment, the negative coefficient is only weakly statistically significant, at $\alpha = 0.1$. For overfitting patches, only the positive coefficient for the deceptive APR treatment is statistically significant, at $\alpha = 0.05$.

Figure 2 summarizes the correctness of the participants' patches (our dependent variable) across treatments. First, there is a strong relationship between the participants being provided *any* suggestion and producing *any* patch: $\chi^2 = 32.841$ with $p < 0.0001$, allowing us to reject the null hypothesis.

Finding: Participants who received a code suggestion for fixing a defect were much more likely to submit a patch than those who received no suggestion.

While producing a patch is a start, the more interesting question is whether the patch is correct, and under what circumstances. The left-hand side of Figure 3 describes the mixed-effect multivariate logistic regression model predicting the participant submitting a *correct* patch. Starting with a correct suggestion (either APR-generated or developer-written) strongly correlates with submitting a correct patch. We confirm the model fit in two ways. First, Wald χ^2 test for the significance of the treatment to the model fit rejects the null hypothesis with $p < 0.0001$. Second, we compute the marginal (R^2_m) and conditional (R^2_c) coefficients of determination for generalized mixed-effect models, which describe the proportion of variance explained by the fixed effects alone and the fixed and random effects together, respectively. Our model fits the data reasonably well overall, with $R^2_c = 68.2\%$, and the fixed effects alone (the type of patch provided) accounts for a large proportion of that variance ($R^2_m = 55.9\%$). Importantly, our data also suggest that developer-written patches and correct APR patches are equivalent in this setting: a post hoc ANOVA test (using Tukey's HSD) comparing the differences between treatment group means does not allow us to reject the null hypothesis at $\alpha = 0.05$.

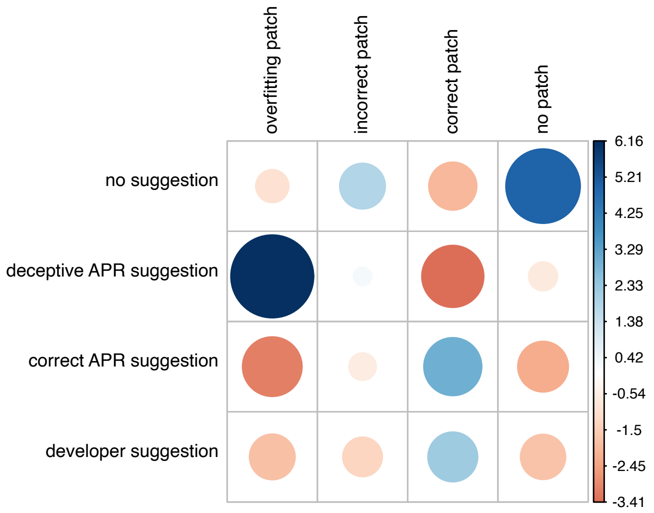

Figure 4: Correlation between patch type and generated fix type. The area of a circle shows the absolute value of the corresponding correlation coefficient. Blue-colored circles represent positive association (attraction). Red-colored circles represent negative association (repulsion).

> **Finding:** 93.3% of participants who received correct suggestions submitted correct patches. Receiving a correct suggestion *helps* successfully repair defects. We find no evidence that the source of the suggestion (APR or developer) matters; however, we do not evaluate whether informing participants of the source of the suggestion would have an effect.

Meanwhile, receiving a deceptive suggestion negatively correlates with submitting a correct patch (that relationship is only weakly significant, $p = 0.0602$) and strongly positively correlated with submitting an overfitting patch ($p < 0.0001$), as the right-hand side of Figure 3 shows. The R^2_m and R^2_c values suggest the model fits the data extremely well $R^m = 95.3$ and $R^2c = 96.1\%$, confirmed with Type II Wald χ^2 tests, and virtually all of the model's variance explained by the fixed effects (that the provided suggestion was deceptive).

> **Finding:** 67.5% of participants who received a deceptive suggestion submitted overfitting patches. Receiving a deceptive suggestion *hurts* the ability to successfully repair defects and increases the chances of submitting an overfitting patch.

A more interpretable representation of these relationships uses a χ^2 test of independence between the treatments, which allows a visualization of the resulting residuals, shown in Figure 4. The area of a circle is the absolute value of the corresponding correlation coefficient. Positive residuals — a positive association between the suggestion and the patch type — are shown in blue, while negative residuals in red. The figure clearly visualizes our key observations:

Model 1, IV: Access to any APR		
Treatment	Coeff.	p-value
any APR	0.9656	0.0146
intercept	-0.5108	0.1178
χ^2	5.9687	0.01456
fit	$R^2_m = 0.0585$	$R^2_c = 0.0585$
Model 2, IV: type of APR suggestion		
Treatment	Coeff.	p-value
correct APR	4.9404	0.00018
deceptive APR	-1.0423	0.081
intercept	-0.5551	0.2573
χ^2	16.898	0.00021
fit	$R^2_m = 0.6201$	$R^2_c = 0.7128$

Figure 5: Mixed-effect multivariate logistic regression parameters for two models, both predicting the participant submitting a correct patch. The top model compares the effect of any APR suggestion to having no suggestion, and bottom model separates out the effects of correct and deceptive APR suggestions. Access to any APR suggestion and to correct APR suggestions statistically significantly improves the chance of submitting a correct patch, while deceptive APR suggestions weakly statistically significantly decrease that chance.

deceptive suggestions are strongly positively associated with producing overfitting patches and correct patches, regardless of source, are strongly positively associated with submitting correct patches. Receiving no suggestion is strongly associated with submitting no patch at all.

4.2 RQ2: How Does APR Affect Debugging Success?

We next focus on whether having access to APR affected debugging success, envisioning APR as part of a human-driven debugging process. We explore whether an APR tool providing suggestions to humans would be practically useful on balance, in light of the fact that some of those suggestions will be deceptive. To understand the implications of this real-world scenario, we build a mixed-effect multivariate logistic regression to explicitly look at and compare the effects of an APR suggestion (whether correct or deceptive) to no suggestion at all. (Thus, we exclude the developer-written suggestions from this model.)

Figure 5 describes two mixed-effect logistic multivariate regression models for predicting the participant submitting a *correct* patch based on APR-provided suggestions. The top model compares access to any APR suggestion to having no suggestion at all. The relationship between being given any APR suggestion and producing a correct patch is positive and significant at $\alpha = 0.05$, but the model does not fit the data very well ($R^2_c = 5.85\%$), which suggests that our model accurately predicts patch correctness, on average, but suffers from high variability. The bottom model separates the

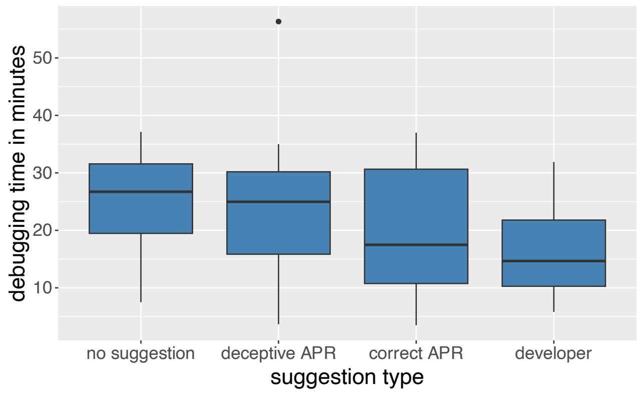

Figure 6: Debugging time (in minutes) distribution by treatment.

effect of correct and deceptive APR suggestions and fits the data much more effectively ($R_c^2 = 71.3\%$, with much of the power coming from the fixed effects). This model shows that deceptive suggestions have a modest negative impact on a participant's ability to create a correct patch (weakly statistically significant, $p = 0.081$), while correct suggestions have a strong, statistically significant positive impact. The two counteract in the top model.

The difference in the effects between correct suggestions and deceptive suggestions is quite large. An estimated coefficient β in a logistic regression model is a log-odd, and can be interpreted as an odds-ratio e^{β}. This suggests that a participant receiving a correct APR suggestion is approximately $e^{4.94} = 140$ times more likely to submit a correct patch than a participant who receives no suggestion. Receiving a deceptive suggestion reduces the probability of producing a correct patch relative to receiving no suggestion by $1 - e^{-1.0423} = 65\%$.

Finding: Compared to having no help from APR, receiving a correct APR suggestion increases participant debugging success by 14,000%, while receiving a deceptive APR suggestion decreases participant debugging success by 65% and increases the chance of submitting an overfitting patch. Correct suggestions are far more likely to help produce correct patches than deceptive suggestions are to hinder producing them, although the overall benefit of using APR depends on the impact of correct and overfitting patches and on the APR tool's correct to deceptive suggestion ratio.

4.3 RQ3: How Do Code Suggestions Affect Debugging Time?

Having understood the effect of code suggestions and APR on debugging success, we now look at their effect on debugging time. Figure 6 summarizes debugging time distributions across the treatments. The mean debugging times for the "no suggestion" treatment (1,492 seconds = 24.9 minutes) and "deceptive APR" treatment (1,409 seconds = 23.5 minutes) are higher than for the "developer" treatment (994 seconds = 16.6 minutes) and the "correct APR" treatment (1,178 seconds = 19.6 minutes). Combined, the mean debugging time for both treatments use correct suggestions ("developer" and "correct APR") was 1,087 seconds = 18.1 minutes.

	APR Correct	Developer	Deceptive
no suggestion	**0.0097**	< **0.001**	0.9838
correct APR	—	0.72617	0.1044
developer	—	—	**0.0011**

Figure 7: Results of Tukey's HSD post hoc analysis to compare means of debugging time between treatments; we omit results for order because, while accounting for order is important in the way it affects the effect of other variables, our focus in this question is how suggestions affect debugging time. The table shows p-values corresponding to whether the difference between the category in the row and the column is statistically meaningful; we bold results that are significant at $\alpha = 0.05$.

We follow a similar analysis approach to determine if these differences are statistically meaningful as we do for patch correctness, instead modeling time (a continuous variable). By contrast with correctness, task order does have a statistically significant effect in a model for task completion time, and so we include it in our analysis. Although participants did not necessarily get better at debugging defects correctly over time, they do appear to have gotten faster.

Both treatment and task order have a statistically significant effect on the model ($p < 0.0001$ for treatment, $p = 0.0089$ for presentation order, for Type II Wald χ^2 tests). Because our concern is whether the apparent differences in completion time in Figure 6 are meaningful, we omit the full model in the interest of brevity and instead focus on the results of a post hoc analysis using Tukey's procedure to compare the means between treatment types.

Figure 7 shows the p-values corresponding to whether the differences in the means of the row and column category are meaningful (with statistically significant differences in **bold**). We observe that:

- Both developer and correct APR suggestions improve debugging time, compared to having no suggestion.
- Debugging time using deceptive suggestions is not significantly different from time required with no suggestion at all.
- Debugging time when using developer suggestions is significantly slower than debugging time using deceptive suggestions.
- There is not a significant difference between correct APR and developer suggestions in terms of debugging time.

Finding: Developers with access to correct suggestions, whether developer-written or APR-generated, tend to debug faster than those with no suggestions or deceptive suggestions.

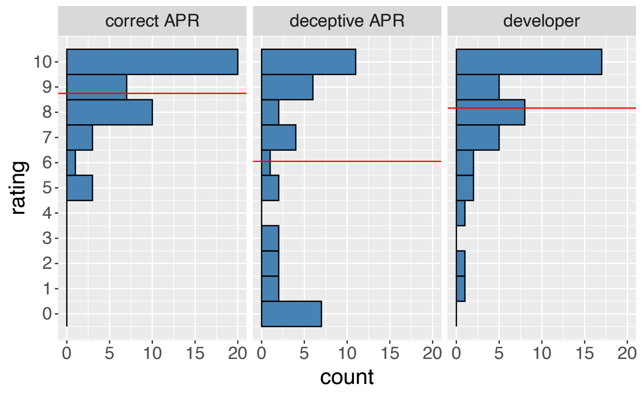

Figure 8: Distribution of the participants' ratings of the quality of the code suggestions. Red lines indicate each group's mean values.

4.4 RQ4: How Can the Effect of Deceptive Suggestions Be Mitigated?

A key to using APR effectively when debugging is being able to tell whether the code suggestion is deceptive. After completing tasks with deceptive suggestions, when responding to "Describe whether this patch was helpful or not helpful in your debugging," example participant quotes include "The patch given is wrong" from participant 15, "The patch resulted in all the test cases passing (I think), but I don't think it was a real solution (treating symptoms rather than disease)" from participant 11, "It was helpful to remove the error. But I have this suspicion that the patch was not correct" from participant 16, "The patch was not helpful, I think it was misleading" from participant 31, and "It helped me pass all test cases. However, I am unsure whether or not this fixed the bug" from participant 39. These quotes suggest some participants could tell deceptive suggestions were deceptive. Thus, we further explore whether participants can recognize deceptive suggestions.

Recall that after each debugging task, the participants were asked to rate the task's suggestion's quality on a scale of 0 to 10. Figure 8 summarizes the distribution of those ratings. The mean quality rating of the deceptive APR suggestions was 6.05 and of the correct APR suggestions, 8.75. This difference is statistically significant, $p < 0.0001$. For developer suggestions, the mean quality rating was 8.17, which is also statistically significantly different from that of deceptive suggestions ($p < 0.0001$), but indistinguishable from the correct APR suggestions.

Finding: Participants were often able to recognize that deceptive suggestions were of lower quality than correct suggestions, producing some hope that, with practice, developers can use APR more effectively to help debug. Still, 67.5% of the participants who received a deceptive suggestion submitted an overfitting patch, suggesting that recognizing low quality may not be enough to sufficiently mitigate deceptive suggestion's impact.

It is possible that more experienced developers are less susceptible to deceptive APR suggestions. We evaluated whether self-reported programming experience interacted with suggestion type to predict debugging success in a mixed-effects linear model, as well as whether self-reported experience correlated with a participant's ratings differentiating between deceptive and correct APR suggestions. We found no statistically significant relationships in these models. We, therefore, find no quantitative evidence that experience improved identification of low-quality suggestions. This suggests that novice developers may already be well equipped to distinguish between high- and low-quality suggestions while debugging. Further, we investigated whether experience predicted submitting a patch, debugging success, or debugging speed, or interacted with suggestion type for the models from RQ1, RQ2, and RQ3, and also found no statistically significant relationships. Thus, we found no evidence that experience mitigated the negative effects of deceptive patches or contributed to more success in using APR. This surprising observation is potentially encouraging as APR may find uses for both experienced and inexperienced developers.

Finding: Surprisingly, we observe no quantitative evidence that more experienced participants are better able to differentiate high- and low-quality patches, and to benefit differently from suggestions and APR, than less-experienced participants. Thus, APR may be beneficial for developers with a range of experience.

4.5 RQ5: How Does APR Use Affect Behavior?

One concern with using APR is that it could allow developers who do not understand the code to get away with simply submitting low-quality patches. Recall that in the after-task survey, we asked developers to explain the underlying defect in each task. We examined these responses to identify cases in which the participants did or did not understand the defect. For example, some responses only stated the failing test outcome or symptom, or the function the test called, with no further explanation, e.g., participant 2 explained the defect as "It was due to a null pointer exception." and participant 38 said "The bug is caused by flaws in the compareTo function." These explanations provide only symptom- or surface-level features of the bug rather than demonstrating genuine understanding. Some participants explicitly stated they could not identify the bug. The survey also asked the participants to explain their bug repairs, and their answers could also indicate understanding (or lack thereof). For example, participant 36 stated that "A patch was given and using that suggestion in the code helped resolve the test failure"; others stated outright that they could not repair the bug.

Our data indicate that suggestion quality strongly impacts participants' understanding of the defects. In 27 of the 89 (30.3%) times participants received correct suggestions (whether APR-generated or developer-provided), the participants failed to demonstrate understanding of the underlying defect, compared to 30 of the 40 (75%) cases where participants received deceptive suggestions; this difference is statistically significant at $\alpha = 0.05$ (via a z-test for differences in proportions).

Regardless of suggestion source, participants often submitted patches even when they did not understand the underlying defect: 20 of the 27 times (74%) for correct suggestions, and 23 of the 30 times (76%) for deceptive suggestions. These proportions are indistinguishable statistically. As to the remaining cases, when starting from a correct suggestion, 3 times, participants augmented a correct suggestion with extra code that, nonetheless, resulted in a correct patch; 2 wrote a different patch from scratch; 1 failed to submit a patch; and 1 thought the defect was in the test and modified the test instead. For those who received deceptive suggestions, 1 failed to submit a patch, and 6 wrote different patches from scratch.

Finding: Having access to a correct suggestion significantly improved the odds of the participants understanding the defect. However, when the participants failed to understand the defect, there was no difference in whether they used the suggestion as a patch between correct and deceptive suggestions.

Finally, we wanted to gauge the overall impression participants had of debugging with code suggestions. In the exit survey, we asked them to "describe their overall experience using the suggested patches when debugging," with the possible answer choices of "they helped me fix the bugs," "they helped me locate the bugs but not directly fix them," "they directed me in the right direction," and "they confused me." Of all the participants, 90.0% said that the suggestions helped them fix the bugs (47.5%) or helped them locate but bugs but not directly fix them (42.5%). Only 7.5% said the suggestions directed them in the right direction, and only 2.5% (one participant) said they confused him. Further, when asked how likely they are to use a tool that automatically generates suggested patches, their responses' mean rating was 9.0 on a scale of 0 to 10 (standard deviation of 1.6). Overall, participants overwhelmingly found suggestions useful when debugging and would use APR tools in the future.

4.6 Threats to Validity

To support external validity of our results, we used the well-established Defects4J benchmark of real-world projects with defects that occurred during development [28]. However, our study design required selecting defects for which at least two of three state-of-the-art APR tools produced a patch. This selection could have biased our study toward specific types of defects APR is more likely to work on, such as perhaps simpler defects. We partially mitigated this risk by ensuring the defects were not too easy to debug.

Our user study used 40 participants, which is within range of higher data confidence and is above average for similar user studies [4, 11, 26, 55, 69]. We used students, but studies have demonstrated that conclusions based on evaluations that use students can generalize to the broader developer community [64, 67].

We report results of proper statistical tests, including effect sizes, for all our findings, increasing the chance our claims generalize. To improve reproducibility, we release our study materials and anonymized data. However, to be practical, our study makes several design decisions that could affect generalizability to real world scenarios. The participants were given a target of 30 minutes to complete each debugging task, whereas such a limit is unlikely in the real world. The participants worked alone and could not seek the help of other developers. The participants were not the original developers of the projects they were debugging. Finally, our study did not investigate whether informing the participants of the source of the code suggestion they were given, e.g., telling them an automated tool produced the suggestion, would affect behavior.

5 RELATED WORK

APR techniques have historically been categorized into heuristic, learning-based or constraint-based repair [38]. Heuristic repair techniques use a set of heuristic modification strategies or edit templates and a generate-and-test methodology to search through the space of possible patched programs. Each variant, produced by instantiating one or more edit templates, is validated using the program's tests. If a variant passes all tests, it is considered to patch the defect.

Heuristics can come from one of three possible sources. The first is a set of low-level code mutations, deletions, and insertions that either delete or copy code from elsewhere (e.g., other parts of the project), or mutate it by, e.g., changing a + to a -. GenProg is one of the early examples of such an APR technique [37, 81], with many more recent advances since [18, 25, 62, 76, 84]. The second is a set of manually defined templates for changing the code, e.g., extracted from manual inspections of human-written fixes, as in the case of Par [30] and TBar [39]. The third is a set of templates for changing the code learned automatically, via machine learning [8, 22, 32, 41, 98]. Learning-based techniques [43, 86, 89, 93] for producing fixed code directly have recently received a boost from advances in neural techniques and large language models.

Constraint-based repair approaches, such as AutoFix-E [79], SOS-Repair [1], Clara [21], SearchRepair [29], Angelix [47], and SemGraft [46], translate the behavioral information available from tests or specifications into constants. Then, they synthesize patches guaranteed by construction to satisfy those constraints, though these patches may not necessarily repair the defect as the constraints are typically partial. Finally, these approaches again validating the patches using tests, sometimes generated against a reference implementation [46].

Researchers have attempted many approaches to improve APR repair rates and patch quality. For example, the effectiveness of the underlying fault localization has been shown to have significant impact [40], and so different localization strategies can improve APR [2, 24, 33, 42, 72, 90]. Advances in heuristic-based search for candidate patch generating can similarly improve APR [25, 37, 41, 62, 76], as can learning-based methods [8, 22, 66], and improved patch-validation methodologies [75, 77, 91, 94, 96]. Automatically inferring test oracles from specifications [5, 20, 52, 97], and then increasing the quality of tests used for repair can, similarly, improve patch quality. While not true in general APR applications, in some domains, the overfitting problem can be solved through other means. For example, for synthesis and repair of formal verification proofs, a theorem prover provides an absolute oracle of correctness [12–14, 68, 92]. And if a reference implementation is available for the program being repaired, comparing behavior to that implementation can reduce overfitting [46]. For repair of meta properties, such as software fairness [6, 16], statistical analyses of observed behavior can provide high-confidence guarantees of correctness [19, 23, 48, 74, 78]. Yet for repair of other properties, such as those of the data software uses, testing remains the best available approach [56, 57].

APR can be evaluated along many dimensions, such as the quality of the patches it produces [54], the impact of other technology on APR success [40], the maintainability of its patches [15], its effectiveness on real-world defects [10, 36, 45, 54, 70], etc. While many APR efforts focus on full automation, the notion of combining APR with manual debugging has been around for nearly two decades [80], and some evaluations have focused on humans, either to judge the patches, or observe the impact on humans.

Less than 7% of APR research papers include an evaluation with users, and some that do are rudimentary (e.g., a single participant) [85]. In 2014, a study [73] with 44 students, 28 engineers, and 23 crowdsourced workers conducted an evaluation of APR patches produced by very early APR tools (GenProg [37, 81] and Par [30]). The study measured the impact of access to defect locations, low-quality suggestions, and high-quality suggestions on debugging; suggestion quality was judged by humans, conflating correctness, maintainability, and other factors. The study found that 71% of patches submitted by participants with access to high-quality suggestions were correct, whereas 48% were correct for participants with access only to the defect location, and 33% with access to low-quality suggestions. With high-quality suggestions, participants took an average of 14.7 minutes to produce a patch, but 17.6 minutes with defect location, and 16.3 minutes with low-quality suggestions. By contrast, with the benefit of a decade of APR research, our study uses more advanced, state-of-the-art APR tools, slightly more defects, and more precise methodologies for evaluating patch quality [35, 54, 65, 69]. Further, our study evaluates the impact of both the APR-generated and developer-written code suggestions. Our results support the prior study's finding that (for our more rigorous definition of quality) high-quality suggestions improve and low-quality suggestions hurt both debugging speed and the quality of the resulting patches.

Other APR user studies are complementary to ours. Cambronero et al. [7] gave all participants code with a single-line defect, tests, and which line contained the defect. Half of participants had access to five APR-generated code suggestions, all modifying the defective line, and all validated to pass the tests, but only one of which was correct. Access to the five code suggestions neither increased the correctness of the submitted patches, nor reduced the time to produce patches. Meanwhile Noller et al. [59] surveyed 103 participants from software companies and crowdsourcing platforms to gauge their opinions of APR. They found that without in-depth manual review, fulltime developers did not trust APR suggestions and wanted to see *evidence* of patch correctness; that participants expected APR to produce suggestions within 30–60 minutes and not to interact with APR directly; and that additional artifacts, such as tests or analysis results can increase trust.

6 CONTRIBUTIONS

Relatively little research has investigated the impact of APR on the human-driven debugging process, despite the fact that industrial APR deployments follow this approach [3, 44]. Our 40-developer controlled user study investigated precisely that impact, showing that the benefits of correct APR suggestions may be significantly greater than the risks of deceptive ones. Developers showed a strong ability to identify deceptive APR suggestions, even if, ultimately, they were driven to produce overfitting patches from such suggestions, perhaps implying that further experience with using APR for manual debugging can improve the process. Surprisingly, however, we found no evidence that programming experience improved the effect of APR on debugging. Overall, our study provides strong evidence of promise for APR-mediated, human-driven debugging, despite existing challenges in APR-generated repair quality.

ACKNOWLEDGMENTS

This work is supported by the National Science Foundation under grants no. CCF-1750116 and CCF-2210243.

REFERENCES

[1] Afsoon Afzal, Manish Motwani, Kathryn T. Stolee, Yuriy Brun, and Claire Le Goues. 2021. SOSRepair: Expressive Semantic Search for Real-World Program Repair. *IEEE Transactions on Software Engineering (TSE)* 47, 10 (October 2021), 2162–2181. https://doi.org/10.1109/TSE.2019.2944914
[2] Fatmah Yousef Assiri and James M Bieman. 2017. Fault Localization for Automated Program Repair: Effectiveness, Performance, Repair Correctness. *Software Quality Journal* 25, 1 (2017), 171–199. https://doi.org/10.1007/s11219-016-9312-z
[3] Johannes Bader, Andrew Scott, Michael Pradel, and Satish Chandra. 2019. Getafix: Learning to fix bugs automatically. *Proceedings of the ACM on Programming Languages (PACMPL) Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA) issue* 3 (October 2019). https://doi.org/10.1145/3360585
[4] Titus Barik, Yoonki Song, Brittany Johnson, and Emerson Murphy-Hill. 2016. From quick fixes to slow fixes: Reimagining static analysis resolutions to enable design space exploration. In *International Conference on Software Maintenance and Evolution (ICSME)*. 211–221. https://doi.org/10.1109/ICSME.2016.63
[5] Arianna Blasi, Alberto Goffi, Konstantin Kuznetsov, Alessandra Gorla, Michael D. Ernst, Mauro Pezzè, and Sergio Delgado Castellanos. 2018. Translating code comments to procedure specifications. In *International Symposium on Software Testing and Analysis (ISSTA)*. 242–253. https://doi.org/10.1145/3213846.3213872
[6] Yuriy Brun and Alexandra Meliou. 2018. Software Fairness. In *ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE) NIER Track*. 754–759. https://doi.org/10.1145/3236024.3264838
[7] José Pablo Cambronero, Jiasi Shen Jürgen Cito, Elena Glassman, and Martin Rinard. 2019. Characterizing Developer Use of Automatically Generated Patches. In *IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. 181–185. https://doi.org/10.1109/VLHCC.2019.8818884
[8] Zimin Chen, Steve James Kommrusch, Michele Tufano, Louis-Noël Pouchet, Denys Poshyvanyk, and Martin Monperrus. 2019. Sequencer: Sequence-to-sequence learning for end-to-end program repair. *IEEE Transactions on Software Engineering* (2019). https://doi.org/10.1109/TSE.2019.2940179
[9] Thomas Durieux, Fernanda Madeiral, Matias Martinez, and Rui Abreu. 2019. Empirical Review of Java Program Repair Tools: A Large-scale Experiment on 2,141 Bugs and 23,551 Repair Attempts. In *ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. 302–313. https://doi.org/10.1145/3338906.3338911
[10] Thomas Durieux, Matias Martinez, Martin Monperrus, Romain Sommerard, and Jifeng Xuan. 2015. Automatic Repair of Real Bugs: An Experience Report on the Defects4J Dataset. *CoRR* abs/1505.07002 (2015). http://arxiv.org/abs/1505.07002
[11] Laura Faulkner. 2003. Beyond the five-user assumption: Benefits of increased sample sizes in usability testing. *Behavior Research Methods, Instruments, & Computers* 35, 3 (2003), 379–383. https://doi.org/10.3758/BF03195514
[12] Emily First and Yuriy Brun. 2022. Diversity-Driven Automated Formal Verification. In *International Conference on Software Engineering (ICSE)*. 749–761. https://doi.org/10.1145/3510003.3510138
[13] Emily First, Yuriy Brun, and Arjun Guha. 2020. TacTok: Semantics-Aware Proof Synthesis. *Proceedings of the ACM on Programming Languages (PACMPL) OOPSLA issue* 4 (Nov. 2020), 231:1–231:31. https://doi.org/10.1145/3428299
[14] Emily First, Markus Rabe, Talia Ringer, and Yuriy Brun. 2023. Baldur: Whole-Proof Generation and Repair with Large Language Models. In *ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. 1229–1241. https://doi.org/10.1145/3611643.3616243
[15] Zachary P. Fry, Bryan Landau, and Westley Weimer. 2012. A human study of patch maintainability. In *International Symposium on Software Testing and*

Analysis (ISSTA). 177–187. https://doi.org/10.1145/2338965.2336775

[16] Sainyam Galhotra, Yuriy Brun, and Alexandra Meliou. 2017. Fairness Testing: Testing Software for Discrimination. In *ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. 498–510. https://doi.org/10.1145/3106237.3106277

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

[18] Ali Ghanbari, Samuel Benton, and Lingming Zhang. 2019. Practical Program Repair via Bytecode Mutation. In *International Symposium on Software Testing and Analysis (ISSTA)*. 19–30. https://doi.org/10.1145/3293882.3330559

[19] Stephen Giguere, Blossom Metevier, Yuriy Brun, Bruno Castro da Silva, Philip S. Thomas, and Scott Niekum. 2022. Fairness Guarantees under Demographic Shift. In *International Conference on Learning Representations (ICLR)*. 24 pages. https://openreview.net/forum?id=wbPObLm6ueA

[20] Alberto Goffi, Alessandra Gorla, Michael D. Ernst, and Mauro Pezzè. 2016. Automatic generation of oracles for exceptional behaviors. In *International Symposium on Software Testing and Analysis (ISSTA)*. 213–224. https://doi.org/10.1145/2931037.2931061

[21] Sumit Gulwani, Ivan Radiček, and Florian Zuleger. 2018. Automated Clustering and Program Repair for Introductory Programming Assignments. In *Programming Language Design and Implementation (PLDI)*. 465–480. https://doi.org/10.1145/3192366.3192387

[22] Rahul Gupta, Soham Pal, Aditya Kanade, and Shirish K. Shevade. 2017. DeepFix: Fixing Common C Language Errors by Deep Learning. In *National Conference on Artificial Intelligence (AAAI)*. 1345–1351. https://doi.org/10.1609/aaai.v31i1.10742

[23] Austin Hoag, James E. Kostas, Bruno Castro da Silva, Philip S. Thomas, and Yuriy Brun. 2023. Seldonian Toolkit: Building Software with Safe and Fair Machine Learning. In *International Conference on Software Engineering (ICSE) Demo Track*. 107–111. https://doi.org/10.1109/ICSE-Companion58688.2023.00035

[24] 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, 10 (2019), 200102. https://doi.org/10.1007/s11432-018-1465-6

[25] Jiajun Jiang, Yingfei Xiong, Hongyu Zhang, Qing Gao, and Xiangqun Chen. 2018. Shaping Program Repair Space with Existing Patches and Similar Code. In *International Symposium on Software Testing and Analysis (ISSTA)*. 298–309. https://doi.org/10.1145/3213846.3213871

[26] Brittany Johnson, Yuriy Brun, and Alexandra Meliou. 2020. Causal Testing: Understanding Defects' Root Causes. In *International Conference on Software Engineering (ICSE)*. 87–99. https://doi.org/10.1145/3377811.3380377

[27] James A. Jones, Mary Jean Harrold, and John Stasko. 2002. Visualization of test information to assist fault localization. In *International Conference on Software Engineering (ICSE)*. 467–477. https://doi.org/10.1145/581339.581397

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

[29] Yalin Ke, Kathryn T. Stolee, Claire Le Goues, and Yuriy Brun. 2015. Repairing Programs with Semantic Code Search. In *International Conference on Automated Software Engineering (ASE)*. 295–306. https://doi.org/10.1109/ASE.2015.60

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

[31] Serkan Kirbas, Etienne Windels, Olayori McBello, Kevin Kells, Matthew Pagano, Rafal Szalanski, Vesna Nowack, Emily Rowan Winter, Steve Counsell, David Bowes, Tracy Hall, Saemundur Haraldsson, and John Woodward. 2021. On The Introduction of Automatic Program Repair in Bloomberg. *IEEE Software* 38, 4 (2021), 43–51. https://doi.org/10.1109/MS.2021.3071086

[32] Anil Koyuncu, Kui Liu, Tegawendé F. Bissyandé, Dongsun Kim, Jacques Klein, Martin Monperrus, and Yves Le Traon. 2020. FixMiner: Mining Relevant Fix Patterns for Automated Program Repair. *Empirical Software Engineering* 25, 3 (May 2020), 1980–2024. https://doi.org/10.1007/s10664-019-09780-z

[33] Anil Koyuncu, Kui Liu, Tegawendé F. Bissyandé, Dongsun Kim, Martin Monperrus, Jacques Klein, and Le Yves Traon. 2019. IFixR: Bug Report Driven Program Repair. In *ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. 314–325. https://doi.org/10.1145/3338906.3338935

[34] Herb Krasner. 2020. The Cost of Poor Software Quality in the US: A 2020 Report. https://www.it-cisq.org/pdf/CPSQ-2020-report.pdf.

[35] Xuan Bach D. Le, Lingfeng Bao, David Lo, Xin Xia, Shanping Li, and Corina S. Pasareanu. 2019. On Reliability of Patch Correctness Assessment. In *International Conference on Software Engineering (ICSE)*. 524–535. https://doi.org/10.1109/ICSE.2019.00064

[36] Claire Le Goues, Michael Dewey-Vogt, Stephanie Forrest, and Westley Weimer. 2012. A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each. In *International Conference on Software Engineering (ICSE)*. 3–13. https://doi.org/10.1109/ICSE.2012.6227211

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

[38] Claire Le Goues, Michael Pradel, and Abhik Roychoudhury. 2019. Automated Program Repair. *Communications of the ACM* 62, 12 (Nov. 2019), 56–65. https://doi.org/10.1145/3318162

[39] Kui Liu, Anil Koyuncu, Dongsun Kim, and Tegawendé F. Bissyandé. 2019. TBar: Revisiting Template-Based Automated Program Repair. In *International Symposium on Software Testing and Analysis (ISSTA)*. 31–42. https://doi.org/10.1145/3293882.3330577

[40] Kui Liu, Li Li, Anil Koyuncu, Dongsun Kim, Zhe Liu, Jacques Klein, and Tegawendé F Bissyandé. 2021. A critical review on the evaluation of automated program repair systems. *Journal of Systems and Software* 171 (2021), 110817. https://doi.org/10.1016/j.jss.2020.110817

[41] Fan Long and Martin Rinard. 2016. Automatic Patch Generation by Learning Correct Code. In *Symposium on Principles of Programming Languages (POPL)*. 298–312. https://doi.org/10.1145/2837614.2837617

[42] Yiling Lou, Ali Ghanbari, Xia Li, Lingming Zhang, Haotian Zhang, Dan Hao, and Lu Zhang. 2020. Can Automated Program Repair Refine Fault Localization? A Unified Debugging Approach. In *International Symposium on Software Testing and Analysis (ISSTA)*. 75–87. https://doi.org/10.1145/3395363.3397351

[43] Thibaud Lutellier, Hung Viet Pham, Lawrence Pang, Yitong Li, Moshi Wei, and Lin Tan. 2020. CoCoNuT: Combining Context-Aware Neural Translation Models Using Ensemble for Program Repair. In *International Symposium on Software Testing and Analysis (ISSTA)*. 101–114. https://doi.org/10.1145/3395363.3397369

[44] Alexandru Marginean, Johannes Bader, Satish Chandra, Mark Harman, Yue Jia, Ke Mao, Alexander Mols, and Andrew Scott. 2019. SapFix: Automated End-to-End Repair at Scale. In *International Conference on Software Engineering (ICSE)*. 269–278. https://doi.org/10.1109/ICSE-SEIP.2019.00039

[45] 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 (EMSE)* 22, 4 (April 2017), 1936–1964. https://doi.org/10.1007/s10664-016-9470-4

[46] Sergey Mechtaev, Manh-Dung Nguyen, Yannic Noller, Lars Grunske, and Abhik Roychoudhury. 2018. Semantic Program Repair Using a Reference Implementation. In *International Conference on Software Engineering (ICSE)*. 129–139. https://doi.org/10.1145/3180155.3180247

[47] Sergey Mechtaev, Jooyong Yi, and Abhik Roychoudhury. 2016. Angelix: Scalable Multiline Program Patch Synthesis via Symbolic Analysis. In *International Conference on Software Engineering (ICSE)*. 691–701. https://doi.org/10.1145/2884781.2884807

[48] Blossom Metevier, Stephen Giguere, Sarah Brockman, Ari Kobren, Yuriy Brun, Emma Brunskill, and Philip S. Thomas. 2019. Offline Contextual Bandits with High Probability Fairness Guarantees. In *Annual Conference on Neural Information Processing Systems (NeurIPS)*. 14893–14904. http://papers.neurips.cc/paper/9630-offline-contextual-bandits-with-high-probability-fairness-guarantees

[49] Martin Monperrus. 2018. Automatic Software Repair: A Bibliography. *ACM Computing Surveys* 51, 1 (Jan. 2018), 17:1–17:24. https://doi.org/10.1145/3105906

[50] Martin Monperrus. 2018. *The Living Review on Automated Program Repair*. Technical Report hal-01956501. HAL/archives-ouvertes.fr. https://hal.science/hal-01956501

[51] Manish Motwani. 2022. *High-Quality Automatic Program Repair*. Ph. D. Dissertation. University of Massachusetts. https://doi.org/10.7275/30288519

[52] Manish Motwani and Yuriy Brun. 2019. Automatically Generating Precise Oracles from Structured Natural Language Specifications. In *International Conference on Software Engineering (ICSE)*. 188–199. https://doi.org/10.1109/ICSE.2019.00035

[53] Manish Motwani and Yuriy Brun. 2023. Better Automatic Program Repair by Using Bug Reports and Tests Together. In *International Conference on Software Engineering (ICSE)*. 1229–1241. https://doi.org/10.1109/ICSE48619.2023.00109

[54] Manish Motwani, Mauricio Soto, Yuriy Brun, René Just, and Claire Le Goues. 2022. Quality of Automated Program Repair on Real-World Defects. *IEEE Transactions on Software Engineering (TSE)* 48, 2 (Feb. 2022), 637–661. https://doi.org/10.1109/TSE.2020.2998785

[55] Kıvanç Muşlu, Yuriy Brun, Michael D. Ernst, and David Notkin. 2015. Reducing feedback delay of software development tools via continuous analyses. *IEEE Transactions on Software Engineering (TSE)* 41, 8 (August 2015), 745–763. https://doi.org/10.1109/TSE.2015.2417161

[56] Kıvanç Muşlu, Yuriy Brun, and Alexandra Meliou. 2013. Data Debugging with Continuous Testing. In *ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE) New Ideas Track*. 631–634. https://doi.org/10.1145/2491411.2494580

[57] Kıvanç Muşlu, Yuriy Brun, and Alexandra Meliou. 2015. Preventing Data Errors with Continuous Testing. In *International Symposium on Software Testing and Analysis (ISSTA)*. 373–384. https://doi.org/10.1145/2771783.2771792

[58] Kunihiro Noda, Yusuke Nemoto, Keisuke Hotta, Hideo Tanida, and Shinji Kikuchi. 2020. Experience Report: How Effective is Automated Program Repair for Industrial Software?. In *International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. 612–616. https://doi.org/10.1109/SANER48275.2020.

9054829

[59] Yannic Noller, Ridwan Salihin Shariffdeen, Xiang Gao, and Abhik Roychoudhury. 2022. Trust Enhancement Issues in Program Repair. In *International Conference on Software Engineering (ICSE)*. 2228–2240. https://doi.org/10.1145/3510003.3510040

[60] Devon H. O'Dell. 2017. The Debugging Mindset: Understanding the Psychology of Learning Strategies Leads to Effective Problem-Solving Skills. *Queue* 15, 1 (Feb. 2017), 71–90. https://doi.org/10.1145/3055301.3068754

[61] Chris Parnin and Alessandro Orso. 2011. Are automated debugging techniques actually helping programmers?. In *International Symposium on Software Testing and Analysis (ISSTA)*. 199–209. https://doi.org/10.1145/2001420.2001445

[62] Justyna Petke and Aymeric Blot. 2018. Refining Fitness Functions in Test-Based Program Repair. In *International Workshop on Automated Program Repair (APR)*. 13–14. https://doi.org/10.1145/3387940.3392180

[63] Justyna Petke, Saemundur O. Haraldsson, Mark Harman, William B. Langdon, David R. White, and John R. Woodward. 2018. Genetic Improvement of Software: A Comprehensive Survey. *IEEE Transactions on Evolutionary Computation (TEVC)* 22, 3 (June 2018), 415–432. https://doi.org/10.1109/TEVC.2017.2693219

[64] Raphael Pham, Stephan Kiesling, Olga Liskin, Leif Singer, and Kurt Schneider. 2014. Enablers, Inhibitors, and Perceptions of Testing in Novice Software Teams. In *ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*. 30–40. https://doi.org/10.1145/2635868.2635925

[65] Zichao Qi, Fan Long, Sara Achour, and Martin Rinard. 2015. An Analysis of Patch Plausibility and Correctness for Generate-and-validate Patch Generation Systems. In *International Symposium on Software Testing and Analysis (ISSTA)*. 24–36. https://doi.org/10.1145/2771783.2771791

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

[67] Iflaah Salman, Ayse Tosun Misirli, and Natalia Juristo. 2015. Are Students Representatives of Professionals in Software Engineering Experiments?. In *International Conference on Software Engineering (ICSE)*. 666–676. https://doi.org/10.1109/ICSE.2015.82

[68] Alex Sanchez-Stern, Emily First, Timothy Zhou, Zhanna Kaufman, Yuriy Brun, and Talia Ringer. 2023. Passport: Improving Automated Formal Verification Using Identifiers. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 45, 2, Article 12 (June 2023), 30 pages. https://doi.org/10.1145/3593374

[69] Edward K. Smith, Earl Barr, Claire Le Goues, and Yuriy Brun. 2015. Is the Cure Worse than the Disease? Overfitting in Automated Program Repair. In *ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. 532–543. https://doi.org/10.1145/2786805.2786825

[70] Victor Sobreira, Thomas Durieux, Fernanda Madeiral, Martin Monperrus, and de Marcelo Almeida Maia. 2018. Dissection of a bug dataset: Anatomy of 395 patches from Defects4J. In *International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. 130–140. https://doi.org/10.1109/SANER.2018.8330203

[71] Victor Sobreira, Thomas Durieux, Fernanda Madeiral, Martin Monperrus, and Marcelo A. Maia. 2023. Defects4J@Dissection. https://program-repair.org/defects4j-dissection/.

[72] Shuyao Sun, Junxia Guo, Ruilian Zhao, and Zheng Li. 2018. Search-Based Efficient Automated Program Repair Using Mutation and Fault Localization. In *International Computer Software and Applications Conference (COMPSAC)*, Vol. 1. 174–183. https://doi.org/10.1109/COMPSAC.2018.00030

[73] Yida Tao, Jindae Kim, Sunghun Kim, and Chang Xu. 2014. Automatically Generated Patches As Debugging Aids: A Human Study. In *ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*. 64–74. https://doi.org/10.1145/2635868.2635873

[74] Philip S. Thomas, Bruno Castro da Silva, Andrew G. Barto, Stephen Giguere, Yuriy Brun, and Emma Brunskill. 2019. Preventing Undesirable Behavior of Intelligent Machines. *Science* 366, 6468 (22 November 2019), 999–1004. https://doi.org/10.1126/science.aag3311

[75] Haoye Tian, Kui Liu, Abdoul Kader Kaboré, Anil Koyuncu, Li Li, Jacques Klein, and Tegawendé F. Bissyandé. 2020. Evaluating Representation Learning of Code Changes for Predicting Patch Correctness in Program Repair. In *International Conference on Automated Software Engineering (ASE)*. 981–992. https://doi.org/10.1145/3324884.3416532

[76] Yuchi Tian and Baishakhi Ray. 2017. Automatically diagnosing and repairing error handling bugs in C. In *ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. 752–762. https://doi.org/10.1145/3106237.3106300

[77] Shangwen Wang, Ming Wen, Bo Lin, Hongjun Wu, Yihao Qin, Deqing Zou, Xiaoguang Mao, and Hai Jin. 2020. Automated Patch Correctness Assessment: How Far Are We?. In *International Conference on Automated Software Engineering (ASE)*. Association for Computing Machinery, 968–980. https://doi.org/10.1145/3324884.3416590

[78] Aline Weber, Blossom Metevier, Yuriy Brun, Philip S. Thomas, and Bruno Castro da Silva. 2022. Enforcing Delayed-Impact Fairness Guarantees. *CoRR* abs/2208.11744 (2022), 24 pages. https://arxiv.org/abs/2208.11744.

[79] Yi Wei, Yu Pei, Carlo A. Furia, Lucas S. Silva, Stefan Buchholz, Bertrand Meyer, and Andreas Zeller. 2010. Automated fixing of programs with contracts. In *International Symposium on Software Testing and Analysis (ISSTA)*. 61–72. https://doi.org/10.1145/1831708.1831716

[80] Westley Weimer. 2006. Patches As Better Bug Reports. In *International Conference on Generative Programming and Component Engineering (GPCE)*. 181–190. https://doi.org/10.1145/1173706.1173734

[81] Westley Weimer, ThanhVu Nguyen, Claire Le Goues, and Stephanie Forrest. 2009. Automatically finding patches using genetic programming. In *International Conference on Software Engineering (ICSE)*. 364–374. https://doi.org/10.1109/ICSE.2009.5070536

[82] Mark Weiser. 1981. Program Slicing. In *International Conference on Software Engineering (ICSE)*. 439–449.

[83] Mark Weiser. 1984. Program Slicing. *IEEE Transactions on Software Engineering (TSE)* 10, 4 (July 1984), 352–357. https://doi.org/10.1109/TSE.1984.5010248

[84] Aaron Weiss, Arjun Guha, and Yuriy Brun. 2017. Tortoise: Interactive System Configuration Repair. In *International Conference on Automated Software Engineering (ASE)*. 625–636. https://doi.org/10.1109/ASE.2017.8115673

[85] Emily Winter, Vesna Nowack, David Bowes, Steve Counsell, Tracy Hall, Sæmundur Haraldsson, and John Woodward. 2023. Let's Talk With Developers, Not About Developers: A Review of Automatic Program Repair Research. *IEEE Transactions on Software Engineering (TSE)* 49, 1 (2023), 419–436. https://doi.org/10.1109/TSE.2022.3152089

[86] Chunqiu Steven Xia and Lingming Zhang. 2022. Less Training, More Repairing Please: Revisiting Automated Program Repair via Zero-Shot Learning. In *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

[87] Xin Xia, Lingfeng Bao, David Lo, and Shanping Li. 2016. "Automated Debugging Considered Harmful" Considered Harmful: A User Study Revisiting the Usefulness of Spectra-Based Fault Localization Techniques with Professionals Using Real Bugs from Large Systems. In *IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 267–278. https://doi.org/10.1109/ICSME.2016.67

[88] Xiaoyuan Xie, Zicong Liu, Shuo Song, Zhenyu Chen, Jifeng Xuan, and Baowen Xu. 2016. Revisit of Automatic Debugging via Human Focus-Tracking Analysis. In *International Conference on Software Engineering (ICSE)*. 808–819. https://doi.org/10.1145/2884781.2884834

[89] Deheng Yang, Xiaoguang Mao, Liqian Chen, Xuezheng Xu, Yan Lei, David Lo, and Jiayu He. 2023. TransplantFix: Graph Differencing-Based Code Transplantation for Automated Program Repair. In *International Conference on Automated Software Engineering (ASE)*. 107:1–107:13. https://doi.org/10.1145/3551349.3556893

[90] Deheng Yang, Yuhua Qi, and Xiaoguang Mao. 2018. Evaluating the Strategies of Statement Selection in Automated Program Repair. In *International Conference on Software Analysis, Testing, and Evolution (SATE)*. 33–48. https://doi.org/10.1007/978-3-030-04272-1_3

[91] Jinqiu Yang, Alexey Zhikhartsev, Yuefei Liu, and Lin Tan. 2017. Better Test Cases for Better Automated Program Repair. In *ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. 831–841. https://doi.org/10.1145/3106237.3106274

[92] Kaiyu Yang and Jia Deng. 2019. Learning to prove theorems via interacting with proof assistants. In *International Conference on Machine Learning (ICML)*. 15 pages. http://proceedings.mlr.press/v97/yang19a/yang19a.pdf

[93] He Ye, Matias Martinez, Xiapu Luo, Tao Zhang, and Martin Monperrus. 2023. SelfAPR: Self-Supervised Program Repair with Test Execution Diagnostics. In *International Conference on Automated Software Engineering (ASE)*. 92:1–92:13. https://doi.org/10.1145/3551349.3556926

[94] He Ye, Matias Martinez, and Martin Monperrus. 2021. Automated patch assessment for program repair at scale. *Empirical Software Engineering* 26, 2 (2021), 1–38. https://doi.org/10.1007/s10664-020-09920-w

[95] Zuoning Yin, Ding Yuan, Yuanyuan Zhou, Shankar Pasupathy, and Lakshmi N. Bairavasundaram. 2011. How Do Fixes Become Bugs?. In *ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. 26–36. https://doi.org/10.1145/2025113.2025121

[96] Zhongxing Yu, Matias Martinez, Benjamin Danglot, Thomas Durieux, and Martin Monperrus. 2019. Alleviating Patch Overfitting with Automatic Test Generation: A Study of Feasibility and Effectiveness for the Nopol Repair System. *Empirical Software Engineering (EMSE)* 24, 1 (Feb. 2019), 33–67. https://doi.org/10.1007/s10664-018-9619-4

[97] Juan Zhai, Yu Shi, Minxue Pan, Guian Zhou, Yongxiang Liu, Chunrong Fang, Shiqing Ma, Lin Tan, and Xiangyu Zhang. 2020. C2S: Translating Natural Language Comments to Formal Program Specifications. In *ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. 25–37. https://doi.org/10.1145/3368089.3409716

[98] 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 *ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. 341–353. https://doi.org/10.1145/3468264.3468544