Improving automatically generated code from Codex via Automated Program Repair

Zhiyu Fan  
National University of Singapore  
zhiyufan@comp.nus.edu.sg

Abhik Roychoudhury  
National University of Singapore  
abhik@comp.nus.edu.sg

Xiang Gao  
Beihang University  
xiang_gao@buaa.edu.cn

Shin Hwei Tan  
Southern University of Science and Technology  
tansh3@sustech.edu.cn

ABSTRACT

Large language models, e.g., Codex and AlphaCode, have shown capability in producing working code for many programming tasks. However, the success rate of existing models remains low, especially for complex programming tasks. One of the reasons is that language models lack awareness of program semantics (e.g., type information), resulting in incorrect programs (or even programs which do not compile). In this paper, we systematically study whether automated program repair (APR) techniques can fix the incorrect solutions produced by language models in LeetCode contests. The goal is to study whether APR techniques can enhance confidence in the code produced by language models. Our study revealed that: (1) automatically generated codes share some common programming mistakes with human-crafted solutions, indicating existing APR tools have the potential to fix auto-generated code; (2) TBar and Recoder, two well-known Java APR tools based on templates and learning respectively, increase the number of solved tasks from 37 to 42 on 60 easy-level tasks, while increase from 5 to 9 on 53 medium-level programming tasks; (3) given bug location information provided by a statistical fault localization approach, the newly released Codex edit mode, which supports changing existing code, may outperform existing APR tools in fixing incorrect solutions. By analyzing the experimental results generated by these tools, we provide several suggestions: (1) as existing APR techniques are still quite limited, including limited patch space, fix locations and patch size, enhancing APR tool to surpass these limitations (e.g., introducing a more flexible fault localization strategy) is desirable; (2) as large language models can derive more fix patterns by training on more data, future APR tools should shift focus from adding more patterns to encoding more program semantics to increase the repair rate; (3) proper combination of language model with techniques that are widely-used in traditional software engineering could be further investigated for improving the efficiency of language models;

ACM Reference Format:
Zhiyu Fan, Xiang Gao, Abhik Roychoudhury, and Shin Hwei Tan. 2022. Improving automatically generated code from Codex via Automated Program Repair. In Proceedings of ACM Conference (Conference’17). ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/nnnnnnnn.nnnnnnnn

1 INTRODUCTION

Designing AI-based systems to automatically solve programming tasks has gained considerable attention in recent years. The most notable of these comes in the form of transformer-based large-scale language models, which used to achieve impressive performance in generating text. The transformer-based models, such as Codex [9] and AlphaCode [22], have successfully generated code for many programming tasks in Python, Java, and C. Technically, those techniques treat code generation as a transformation problem, which takes as input natural language descriptions and transforms them into programming language.

Although transformer-based language models successfully solved many programming tasks, their success rate is still relatively low. When evaluating on pass@5 metric [9], the best Codex model achieves 24.52% passing rate at introductory-level tasks and 3.08% passing rate at competition-level tasks [9] from APPS dataset [14]. While the best AlphaCode model achieves 20.36% and 7.75% passing rates on introductory-level and competition-level tasks respectively [22]. Lacking deep understanding of task descriptions and program semantics are the main reasons that cause the low success rate. Transformer-based models treat code generation as a sequence-to-sequence transformation by treating description and code as token sequences which cannot capture deep semantic features of programs (e.g., program semantic can be encoded in the form of a test case). In contrast, generating entire programs requires understanding the entire task descriptions which usually comprise complex logic, and figuring out the solutions to programming tasks relies on deep algorithm reasoning.

Automated program repair (APR) is an emerging area for automated rectification of programming errors [27]. Automated repair techniques take as inputs a buggy program and a correctness specification, and produce a fixed program by slightly changing the program to make it satisfy the given specification. Typical repair tools generate patches by reasoning about the program semantics against the given specification. For instance, semantic-based repair tools (e.g., SemFix [30], Angelix [29]) generate patches by using symbolic execution and search-based repair tools (e.g., Gen-Prog [40], TBar [25]) search for correct patches among pre-defined
Table 1: Our key finding and implications on the bug patterns made by Codex and the effectiveness when applying existing repair tools and Codex-e to fix these bugs.

| Findings on Bug Pattern (Section 3) | Implications |
|-------------------------------------|--------------|
| (1) 47% of bugs made by Codex are syntax errors | (i) As Codex generated solutions shared common mistakes with human-written solutions, we can use APR tools to fix these mistakes; (ii) Designers of language models should consider incorporating program syntax into the model (address finding (1)), and extracting information from task description (address finding (2)) to enhance code generation. |
| (2) 33% of bugs made by Codex are misaligned algorithms | |
| (3) To fix the bugs made by Codex, 9% of them require small changes, e.g., change operator, modify expression, change statement. | |
| (4) To fix the bugs made by Codex, 10% of them need large patches | |

| Findings on APR’s effectiveness (Section 4) | Implications |
|--------------------------------------------|--------------|
| (5) Existing template based and learning based APR tools do not perform well in fixing the buggy solutions made by Codex, especially on the bugs that require large patches. They together fix 11 single-hunk buggy solutions, which increase the number of solved tasks from 37 to 42 on the easy-level tasks, and increase from 5 to 9 on the medium-level tasks. | The pattern-based and learning-based APR tools we tried, were found to be quite limited, including limited patch space, fix locations and patch size. |

| Findings on Codex Edit Mode (Section 5) | Implications |
|----------------------------------------|--------------|
| (6) The efficacy of Codex-e for automated repair is worth studying. Given “proper” instructions (such as where to fix), Codex-e even outperformed pattern-based and learning-based APR tools. So, what kind of guidance can be given to Codex-e, needs to be studied. | A proper combination of language model with techniques that are widely-used in traditional software engineering (e.g., fault localization, and AST information) could be further investigated for improving the efficiency of language models; |
| (7) Codex-e is able to generate patches at flexible locations beyond the given location or statement. This enables Codex-e to produce more correct and larger patches, especially when the given location is not precise. | As existing APR techniques use the program location (line numbers) provided by statistical fault localization to search for patches at specific lines, it may restrict the search space of patches to these lines. Future APR tools could explore more flexible form of fix localization to allow fixes to be generated at multiple locations. |

search space with the reference to dynamic execution results. APR has shown promising results in fixing real-world bugs and vulnerabilities. For instance, in a well-studied dataset Defects4J [19] version 1.2.0 including 395 real bugs, more than 25% (101) of bugs have been automatically fixed by at least one APR tool [25]. Nevertheless, existing APR tools are limited to generating small patches (usually one-line fixes) due to the complexity of semantic reasoning (semantic-based APR) and search space explosions problem (search-based APR) when considering multiple-line fixes. In other words, APR tools are good at generating small patches via semantic reasoning but cannot generate a large chunk of code.

The strength and weakness of language models and APR techniques inspire us to think about the following question: can automated program repair improves the code produced by language models?

In this paper, we apply existing APR techniques to the code generated by the Codex model, and answer the following research questions:

(RQ1) What mistakes do the auto-generated code usually make?

Although we know that the language models produce many wrong solutions when solving programming tasks, several open questions still remain: (i) what are the types of bugs made by language models; (ii) are the bugs made by language models similar with human-written bugs; (iii) how many of these bugs can be fixed with single-hunk change. We first study the bug pattern of code produced by Codex on how they can be fixed.

(RQ2) How effective are APR tools in fixing code from Codex?

Existing APR tools are mainly designed to fix human-written bugs. To achieve this, common APR tools generate patches by defining transformation operators (search-based APR) or specifying the program synthesis ingredients (semantic-based APR). These operators and ingredients have been proven to be efficient in fixing human-written bugs. We study whether these operators or ingredients still work when fixing the bugs in auto-generated code. Specifically, we study how effective APR tools are in fixing the code produced by Codex.

(RQ3) Can Codex edit mode fix program bugs?

In March 2022, a new version of Codex was released, which can edit existing content in a complete program rather than just completing a partial program. This new feature makes it practical to use Codex to modify or improve existing code for program transformation, code reconstruction, and bug fixing. Codex edit mode (we call this mode Codex-e throughout this paper) requires users to provide instructions to guide the revision, such as “translate the java program to javascript”. To fix a bug, users need to provide precise and clear instructions. How to automatically produce such instruction still remains an open question. We study whether the side effect of APR tools, such as fault localization results, can

1https://openai.com/blog/gpt-3-edit-insert
be used to guide Codex-e, and how efficient Codex-e is in fixing program bugs.

The key findings of our study are summarised in Table 1. According to these key findings, we discuss how to further improve the success rate of language models. In this paper, we mostly used pattern-based and learning-based APR tools in our experiments. Our result shows that existing APR tools are still quite limited, including limited patch space, fix locations and patch size, enhancing APR tool to surpass these limitations (e.g., introducing a more flexible fault localization strategy) is highly desirable. Besides those repair tools, there are many other semantic analysis techniques that are widely-used in traditional software engineering could be further investigated for improving the efficiency of language models.

Overall, our contributions can be summarized as follows.

- To the best of our knowledge, we present the first study of automated repair of buggy programs automatically generated by the Codex model.
- To the best of our knowledge, we include the first study that evaluates the efficacy of the newly released Codex-e as an automated repair tool.
- We propose LMDefects, a new dataset that contains 42 correct Java programs (37 from easy-level tasks and 5 from medium-level tasks) and 355 buggy Java programs (115 from easy-level tasks and 240 from medium-level tasks), together with a taxonomy of the defect types exist in these buggy solutions.

2 STUDY SETTING

In this section, we present the setting of our study, including the overall workflow, the Codex model, parameters, dataset, APR tools, etc. All experiments in this paper are conducted on a Ubuntu-16.04 server, with 64GB RAM and Intel(R) Xeon(R) CPU E5-2660 v4 @ 2.00GHz, and NVIDIA Titan V GPU.

2.1 Methodology

Figure 1 shows the overall workflow of our study. To answer research questions in Section 1, we build a dataset containing 113 programming tasks from the recent LeetCode contest [2]. We first use Codex to generate initial solutions for each task and use the public tests to validate the correctness of generated solutions. For the unsolved programming tasks, existing repair tools are then applied to the solutions produced by Codex. The patched solutions are then validated by the given public tests and the LeetCode online judge platform to measure how many solutions can be fixed by APR tools.

2.2 Codex Model

Codex [9] is the model that powers GitHub Copilot 2, which completes a program by given natural language prompt. Codex supports many programming languages (e.g., Python, C/C++, Java, Go). The Codex models are descendants of GPT-3 models [6]. Their training data contains both natural language and billions of lines of public code from GitHub. In our study, we use the pre-trained Codex code-davinci-002 model and Codex-e code-davinci-edit-001 model [1], which were both trained on data up to Jun 2021.

2.3 LMDefects

Several datasets consist of programming tasks exist [7, 9, 14, 22, 33, 38]. They are either based on programming contests collected from programming competition platforms such as Codeforces or handwritten tasks. We do not use existing datasets because (1) Codex has been trained on GitHub where solutions for many previous programming tasks exist (e.g., APPS, CodeContest) but we would like to check how Codex performs in a real-world programming contest setting, (2) not every programming task includes public tests which APR techniques rely on (e.g., HumanEval).

We build a new dataset LMDefects by mining programming tasks from LeetCode contest [2]. LeetCode is an online judge platform that people can use to practice their programming skills by solving programming tasks. It has over 2,300 different problems, ranging from easy level to hard level. It also has a discussion page 3 with active community where we can find the correct solutions for a programming task (important for our manual analysis of incorrect solutions). For each programming task, there are usually 1–3 public tests that provide examples with pairs of (input, output) to explain the requirement. When a solution is submitted to LeetCode, it runs a large number of private tests to validate the correctness of submissions. LeetCode has weekly and biweekly contests, where it releases new programming tasks. Each contest includes one easy-level problem, two medium-level problems, and one hard-level problem. In our study, we only consider easy-level and the first medium-level problems, since Codex still struggles with hard problems [9] (we also filter seven tasks that require customized data structure because those are unlikely to be solved by Codex, e.g., merge-nodes-in-between-zeros in weekly-contest-281 requires ListNode). To avoid the situation where our dataset has already been included in the training set of Codex, we only consider contests that are released after Jun 2021 (the end date where the Codex training data is extracted from).

Overall, we crawl through all contests in LeetCode from 4 July 2021 until 6 Apr 2022. This leads to a total of 40 weekly contests and 20 biweekly contests. In total, LMDefects contains 113 programming tasks, with 60 easy-level and 53 medium-level tasks, respectively.

2.4 Prompt and Parameters

Codex model takes as inputs a prompt, which is the combination of natural language text and code snippet, where the natural language text represent the programming task description and code snippet is the starting point for language model to complete the code. We evaluate Codex in zero-shot prompt settings (does not include example input/output in the prompt). Note that public test cases can also be embedded in the prompt (namely n-shots prompt), meaning that public tests are also available for Codex. Lines 1–8 in Figure 2 presents an example of prompt that we use for a LeetCode programming task 4. Given such a prompt for each programming task, we simulate the actual contest scenario by running Codex to generate 50 candidate solutions and choose the top five solutions with the highest probability of being correct (use the best_of parameter). The selected solutions are then submitted to the LeetCode online judge platform to validate whether they are correct. As reported

3https://leetcode.com/discuss/
4https://leetcode.com/problems/minimum-operations-to-convert-number/
```java
public static int minimumOperations(int[] nums) {
    // You are given a 0-indexed array nums consisting of n positive integers.
    // The array nums is called alternating if:
    // * nums[i - 2] == nums[i], where 2 <= i <= n - 1.
    // * In one operation, you can choose an index i and change nums[i] into any positive integer.
    // * The array nums is called alternating if:
    //   * nums[i - 2] == nums[i], where 2 <= i <= n - 1.
    //   * In one operation, you can choose an index i and change nums[i] into any positive integer.
    // Return the minimum number of operations required to make the array alternating.
    int n = nums.length;
    int[] dp = new int[n + 2];
    dp[0] = 0; dp[1] = 1;
    for (int i = 2; i < n; i++) {
        dp[i] = dp[i - 1] + 1;
        if (nums[i] == nums[i - 2])
            dp[i] = Math.min(dp[i - 2] + 1, dp[i]);
    }
    return dp[n - 1];
}
```

Figure 1: The workflow of automatically fixing programs generated by Codex

Figure 2: An example of misaligned algorithm, taken from LeetCode programming task `minimumOperations` on weekly-contest-280

in Codex [9], we use temperature as 0.8, which has the best performance when generating 50 candidate solutions, and we use the same setting in [9] to prepare the stop tokens as ["public", "class", "/", "System.out.print"]. We also follow OpenAI playground [1] default setting to set the "max token length" for each completion request to 256.

2.5 APR Tools

To evaluate whether repair tools can fix the incorrect solutions produced by Codex, we run our experiments on two Java APR tools. Although researchers have developed APR tools for fixing bugs in many programming languages (e.g., C, Java, Python), we use Codex to produce Java programs because Java APR tools have been widely studied, and many of them are open-source. Among all the open-source Java APR tools, we select TBar and Recoder because (1) they are the most recent representative of different approaches (i.e., TBar represents a search-based and a pattern-based APR tool, whereas Recoder is a learning-based approach), and (2) these tools have reported the best results by generating the highest number of correct patches on the Defect4J [19] benchmark (a benchmark in which almost all Java APR tools have been evaluated on).

Below are the two Java repair tools used in our study:

- **TBar**: TBar [25] is a search-based automated program repair tool, which first constructs a search space containing a set of candidate patches, and searches for the correct patch(es) over the space space. TBar has collected a set of widely-used fix patterns from existing APR tools, including SimFix [16], HDRepair [21], PAR [20] and so on (refer to their paper [25] for the complete list of work). In total, TBar supports 35 fix patterns.

- **Recoder**: Recoder [48] is a learning-based repair tool that learns from existing patches of open-source projects. Different from search-based APR tools that require predefined code change operators to generate candidate patches, learning-based tool learns code change operators from existing patches, hence has more flexibility in generating candidate patches.

Since both TBar and Recoder rely on test cases as correctness criteria, the public test cases from LeetCode are used to guide the repair process, while the private test cases are applied to validate the patched solutions. In this work, we run TBar and Recoder in default settings, stopping the repair process if one patch that passes all public tests is found. We set the timeout to 15 minutes, following the time limit used in prior work of fixing students’ programming assignment using APR tools [45].

Moreover, since Codex-e can change the content of existing code by generating program edits. We investigate whether Codex-e can be further used as an APR tool and compare its performance with TBar and Recoder. We discuss how to automatically generate instructions for Codex-e at in Section 5.

3 RQ1: WHAT MISTAKES DO AUTO-GENERATED CODE USUALLY MAKE?

Prior study of search-based automated program repair techniques shows that automatically generated patches are likely to exhibit certain anti-patterns (program transformations that should be prohibited as they lead to the generation of nonsensical patches) [39]. Intuitively, automatically generated code by a large language model like Codex may also contain anti-patterns. Hence, we analyze whether the code generated by Codex made the same common mistakes. Specifically, when we give a programming task in LMDeffects for Codex to solve, we first run the five automatically generated solutions on the public tests and then validate those solutions that pass public tests on the private tests by submitting them to LeetCode online judge platform. If all the five automatically generated solutions S by Codex cannot pass all tests (public and private tests), we consider S as unsolved solution. If there is no unsolved solution S for a programming task, we consider the programming task as solved. Finally, Codex produces 37 and 5 solved programming
tasks on easy and medium levels respectively, and we study the
mistakes of 355 unsolved solution S in the remaining 71 unsolved
programming tasks that lead to compilation errors or test failures.

For each unsolved solution S generated by Codex, we man-
ually fix the bugs in the solution by first referring to other solutions
for the same programming task for repair hints, and then construct-
ing a minimal patch that fixes the bugs by making the least program-
modifications to existing code. Our goal is to construct a "ground
truth" patch Sfixed for each unsolved solution to obtain the "diff" between Sbuggy and Sfixed. This "diff" represents the bugs or mis-
takes in the automatically generated program by Codex. Based on
this "diff", we manually classify each unsolved solution using the
following categories:

- **Multi-hunk fix required**
  - (M-S) Similar: Similar bugs (require similar fixes) exist at multiple
    program locations
  - (M-U) Unique: Distinct bugs exist at multiple program locations
  - (M-L) Need Large Fix: Need to edit more than five lines in total
    at multiple program locations

- **Single-hunk fix required**
  - (S-O) Operator Mutation: Change the arithmetic/logical/bitwise
    operators
  - (S-V) Variable Mutation: Replace with a different variable
  - (S-T) Type Cast: Replace the data type
  - (S-E) Expression Mutation on Operand: Modify an expression
    as part of an operand
  - (S-AS) Add Statement: Insert a new statement
  - (S-MS) Move Statement: Move a statement to a new location
  - (S-DS) Delete Statement: Delete an existing statement

- **Syntax error**
  - Incomplete Code: For the last line of the program, only parts of
    the program is printed
  - Bracket mismatch: Fails to compile due to missing/extra bracket
  - Invoke undefined functions/classes: Fails to compile due to in-
    volving a function or a class that do not exist
  - Incompatible types: Fails to compile due to type errors

- **Algorithm-related**
  - Misaligned Algorithm: The algorithm used is misaligned with the
    requirement given in the task description.

Table 2 shows the defect classification of unsolved solutions,
together with examples (the "Example" column), and the difficult
levels (the "Easy" and the "Medium" columns) of the programming
tasks to explain each defect. To allow for easier comparison of defect
types, we derive the defect classification based on categories used in
Codeflaws [38] (a benchmark that contains incorrect submissions
by participants in programming competitions). Compared to the
defect classification in Codeflaws, we realized that the types of
bugs made by automatically generated code overlap with those in
Codeflaws. Specifically, both Codeflaws and our dataset contain
defects where either multi-hunk or single-hunk fixes are required.
Moreover, for the single-hunk fixes, both datasets share similar
mutation operators (e.g., operator mutation, and variable mutation).
This indicates Codex made similar programming mistakes as human
participants for defects that require multi-hunk or single-hunk fixes.
4 RQ2: HOW EFFECTIVE ARE APR TOOLS IN FIXING THE CODE PRODUCED BY CODEX?

As shown in Section 3, the mistakes made by Codex share some similarities with human-written solutions. In this section, we study how effective existing APR tools are in fixing the code produced by Codex. Since existing APR tools are designed to produce small patches (usually one-line or few-line fixes), we exclude the buggy programs that cannot be compiled or require changing the entire algorithms, which are not supported by current APR tools.

We are aware of techniques that can automatically fix compilation errors [4, 12]. However, in this study, we only evaluate tools that fix coding errors, and we leave the evaluation of tools for fixing compilation errors as future work.

Given the unsolved solutions by the Codex model (excluding solutions that produce syntax errors and algorithm-related errors), we run TBar and Recoder to assess their ability in generating patches. During the patch validation stage, the automatically generated patches are categorized as below:

**Plausible patches.** Plausible patches are patches that make the unsolved solutions pass the given public tests.

**Correct patches.** Correct patches are patches that make the unsolved solutions pass both the public tests and private tests and accepted by LeetCode.

Table 3 shows the number of generated patches and the number of fixed tasks by TBar and Recoder (include single-hunk and multi-hunk).

| Tool       | Correct/Plausible patches | Correctly Fixed Tasks |
|------------|---------------------------|-----------------------|
|            | easy  | medium | easy  | medium |
| TBar       | 4/12  | 2/8    | 3     | 2      |
| Recoder    | 6/10  | 4/12   | 4     | 4      |

Figure 3: Examples of LeetCode programming tasks fixed by Recoder but not TBar.
the limited number of public tests is one of the reasons that prevent APR tool from generating more correct patches.

The "Correctly Fixed Tasks" columns of Table 3 show the number of programming tasks correctly fixed by TBar and Recoder, respectively. Note that each programming task corresponds to the five selected unsolved solutions. If any of solutions is correctly fixed (accepted by LeetCode), we consider that this task has been solved (after five trials). In total, Recoder fixes eight programming tasks whereas TBar only fixes five tasks. Overall, TBar increases the number of solved tasks from 37 to 40 on the easy-level tasks (Recoder further increase this number by fixing two other easy-level tasks), while Recoder increases the number of solved medium-level tasks from 5 to 9. Combining both tools, APR tools help Codex fix 5 and 4 more easy-level and medium-level tasks, respectively.

We further analyze the type of defects fixed by the two APR tools. Table 4 shows the number of solutions that can be correctly fixed for each defect category, where the "TBar" and the "Recoder" columns show the number of patches produced by the corresponding tools. For each category, the repair tools may not fix the bug by minimally changing the program (i.e., repair tools may fix a bug using different operators than the minimal fix shown in the "Defect sub-category" column). Overall, on the easy-level tasks that require single-hunk fixes, TBar correctly fixes 4 out of 23 solutions, while on medium-level tasks, TBar fixes 2 out of 9 buggy solutions. In contrast, Recoder correctly fixes 6 out of 23 easy solutions, while on medium-level tasks, it fixes 4 out of 9 buggy solutions. Figure 3 shows two examples where Recoder outperforms TBar. In the first example, despite having the “Mutate Literal Expression” pattern, TBar fails because it cannot find the correct literal to replace due to limited mutation space (e.g., only support change 1 to 1d or 1f, but we can use program synthesis technique to find the correct literal). For the second example, TBar fails to generate the correct patch because it does not have any patterns that insert statements by copying from other locations. For the tasks which require multi-hunk fixes, both TBar and Recoder fail to generate any correct patches. This shows that existing APR tools are still quite limited in generating complex patches that require edits of multiple lines.

### Table 4: The number of correctly fixed solutions using different APR tools (only single-hunk bugs considered).

| Defect Sub-category | Total | TBar | Recoder |
|---------------------|-------|------|---------|
| S-O                 | 2     | 3    | 1       |
| S-V                 | 3     | -    | 1       |
| S-T                 | 1     | -    | -       |
| S-E                 | 7     | 3    | 2       |
| S-AS                | 8     | 2    | 1       |
| S-MS                | 1     | -    | -       |
| S-DS                | 1     | 1    | 1       |
| Total (Single-Hunk) | 23    | 9    | 4       |
| M-S/M-U/M-L         | 14    | 23   | -       |

Existing repair tools increase the number of solved tasks from 37 to 42 on the easy-level tasks, while increase from 5 to 9 on the medium-level tasks.

### 5 RQ3: CAN CODEX EDIT MODE FIX PROGRAM BUGS?

Recently, OpenAI released a new edit mode of Codex which has the ability to change the content of an existing program. Codex edit mode takes a program and a natural language instruction as inputs, and outputs an edited program based on the instruction. As Codex-e can produce edited programs as patches, a natural question to ask would be “Can the Codex edit mode fix an incorrect program with proper instruction?” In this section, we explore the possibility of using Codex-e as an automated program repair tool. To reduce the influence of natural language descriptions on Codex-e, we removed the task description in each unsolved solution. We designed three strategies to construct the edit instruction for Codex-e.

- **Codex-e*_bug:* We tell Codex-e that a bug exists in the given program and ask Codex-e to fix it. The instruction is simply given as "Fix bug in the program".

- **Codex-e*_line:* We follow existing automated program repair techniques that use statistical fault localization technique (Ochiai) [3, 8] on the generated incorrect solutions to get a sequence of candidate fix line numbers. These candidate line numbers are then provided to Codex-e as fix hints. The instruction for Codex-e is formulated as "Fix line N".

- **Codex-e*_stm:* Considering the large language models like Codex are trained with plain natural language, we further investigate how Codex-e would respond if we directly use the suspicious statements instead of suspicious line numbers as instruction. To construct the edit instructions, we use the program text of the statements at the suspicious line and formulate it as "Fix s1".

For example, to fix the expression mutation bug for the `makeFancyString` task in Figure 3, we give Codex-e*_line the instruction `Fix line 6`, and provide Codex-e*_stm with the instruction `Fix "1 -= 2;"`.

For each unsolved solution (we exclude solutions that produce syntax errors and algorithm-related error as in Section 4), we select the ten most suspicious statements and ask Codex-e to generate five possible edits for each statement (i.e., Codex-e tries to fix an incorrect solution within 50 attempts). Similar to the initial solution generation in the regular Codex mode, we set the temperature at 0.8 to increase the possibility of finding a correct edit.

Table 5 shows the results for the three strategies, where columns Codex-e*_bug, Codex-e*_line and Codex-e*_stm show the number of correct patches using corresponding edit instructions. With Fix bug in the program as instruction, Codex-e*_bug only learns about the existence of bugs in the given program without any information about the fault locations. Surprisingly, with the limited information, Codex-e*_bug successfully produced 15 correct patches, including two patches that need large edits at multiple hunks (M-L, refer to supplementary material for the example). In contrast, when giving the faulty line number as instruction, Codex-e*_line only fixes six solutions that require single-hunk fix, and one solution that require multi-hunk fixes.
Table 5: The number of correctly fixed solutions using Codex-e with different strategies.

| Defect Category | Sub-Category | Total      | Codex-e^{bug} | Codex-e^{line} | Codex-e^{stm} |
|-----------------|--------------|------------|---------------|----------------|--------------|
|                 | easy medium  | easy medium| easy medium   | easy medium    | easy medium  |
| Single-Hunk     | S-O          | 2          | 3             | 1              | -            | 2            |
|                 | S-V          | 3          | -             | -              | -            | -            |
|                 | S-T          | 1          | -             | -              | -            | 1            |
|                 | S-E          | 7          | 3             | 3              | 2            | 3            |
|                 | S-AS         | 8          | 2             | -              | -            | 1            |
|                 | S-M          | 1          | -             | -              | -            | 1            |
|                 | S-DS         | 1          | 1             | 1              | -            | 1            |
| Total           | -            | 23         | 9             | 7              | 3            | 10           |
| Multi-Hunk      | M-S          | 4          | 1             | 2              | -            | 1            |
|                 | M-U          | 4          | 8             | 1              | -            | -            |
|                 | M-L          | 6          | 14            | -              | 2            | -            |
| Total           | -            | 14         | 23            | 3              | 2            | 1            |

Compared to Codex-e^{bug} and Codex-e^{line}, Codex-e^{stm} produces the best results by successfully fixing 16 buggy solutions. We attribute the effectiveness of Codex-e^{stm} to its use of program texts (e.g., “i -= 2;”) that may be more helpful in guiding a language model like Codex to match for relevant statements.

```
public static int[][] construct2DArray(int[] original, int m, int n) {
    int[][] result = new int[m][n];
    for (int i = 0; i < result.length; i++) {
        for (int j = 0; j < result[i].length; j++) {
            if (i * result[i].length + j >= original.length)
                result[i][j] = original[i * result[i].length + j];
            else
                return result;
        }
    }
    return result;
}
```

Figure 4: Flexible fault localization example of LeetCode programming task convert-1d-array-into-2d-array on biweekly-contest-62 fixed by Codex-e

Furthermore, we manually analyze patches produced by Codex-e, and find that Codex-e is able to generate patches at flexible locations. Prior APR work [17, 25, 26, 48] have shown a significant performance gap with/without perfect fault localization results. While existing APR tools strictly try to produce patches at a given faulty line number, ignoring the possibility of fixing a bug in the relevant context, Codex-e does not have such limitations. In the 16 correctly fixed solutions by Codex-e^{stm}, 6 (37.5%) of them are fixed by editing beyond the statement provided in the given instruction. Figure 4 shows one such example. The instruction provided to Codex-e^{stm} is Fix “[for(int i = 0; i < result.length; i++)],” and Codex-e^{stm} fix this by moving one if-then clause out of the loop body and changing the if-condition.

This example indicates that Codex-e^{stm} is not restricted by the given suspicious statement, and can modify the relevant surrounding code. Compared to traditional APR tools, using flexible fault localization is an important feature that enables Codex-e to produce more correct patches since fault localization techniques may fail to point to the correct fix location.

The efficacy of Codex-e as an automated repair tool could be worth studying. Without specific fix location, Codex-e^{bug} fixes 15 buggy programs and with suspicious faulty line number and suspicious statement being provided, Codex-e^{line} and Codex-e^{stm} successfully fixes 7 and 16 buggy programs, respectively.

6 IMPLICATIONS AND DISCUSSIONS

Limitations of language models. Apart from the mistakes common in human-crafted programs, our study in Section 3 reveals that syntax error and misaligned algorithm are the key limitations in automatically generated solutions by Codex. Syntax errors in automatically generated programs can be solved via (1) leveraging existing techniques on compilation error repair [4, 13, 44], (2) encoding a programming language model into the Codex model so that it will generate compilable programs (the approach taken by CURE [17]), or (3) invoking the Codex model iteratively or combining different modes of Codex to synthesize the undefined functions (the fact that we can use Codex-e for fixing the initial incorrect solutions generated by the Codex model shows the potential of this approach). The misaligned algorithm is more a severe problem that has been similarly observed by the Codex paper [9]. Based on our manual analysis of the generated solutions, Codex seems to rely heavily on the function name (e.g., minimumOperations in Figure 2) for solving the programming tasks. In fact, a recent study has also observed the tendency of Codex in generating solutions based on function name [18]. With the long prompt (function signature and the problem description) given, the function name may be more concise and easier to search for compared to the lengthy problem description in GitHub. However, this strategy fails when a customized algorithm is required to solve a programming task. Relying on function name to search for relevant code will reduce the generation power of Codex to a simple API search engine that...
returns the implementation for a given API. To solve the misalignment problem, future language models designed for code generation should focus on summarizing useful information from the problem description instead of solving the problem of generating code based on function names.

**Pattern-based APR versus learning-based APR.**

Section 4 shows that a learning-based APR approach (Recoder) outperforms a pattern-based APR tool (TBar). We think that this result shows the limitation of a pattern-based tool that relies on pre-defined fix operators. TBar, the state-of-art pattern-based APR tool, which integrates more than 30 specifically designed fixing patterns collected from previous literature, only fixes 6 out of 32 single-hunk buggy solutions. Moreover, our manual analysis revealed that although TBar has the required fix patterns for some single-hunk bugs, it still fails to fix them because the concrete implementation of a fixing pattern does not always cover all possible combinations. For example, TBar has the ability to mutate a literal, variable or expression, but it does not support generating a non-existent element which is required by the `makeFancyString` example shown in Figure 3.

Given the great variety of bug combinations that Codex can make, TBar does not perform well on many buggy solutions, especially on those that require adding statement. In contrast, Recoder, a learning-based repair tool that does not rely on pre-defined fix patterns, generates a few more correct patches than TBar. Figure 3 shows such an example where Recoder fixes this bug by adding a statement `steps += 1;` at line 19, which is not supported by TBar. In short, a pattern-based approach like TBar fails to fix bugs that require either (1) additional fix patterns, or (2) a large search space for fix ingredients (e.g., specific literal). This limitations show that a pattern-based APR tool is hard to scale. Instead of manually adding more patterns to a new APR tool, future APR research on designing fixing operators should shift to a more scalable way (e.g., synthesizing more patterns to a new APR tool, future language models designed for code generation should focus on summarizing useful information from the problem description instead of solving the problem of generating code based on function names. Here is an example of a test-based APR approach like Codex which integrates more than 30 specifically designed fixing patterns collected from previous literature, only fixes 6 out of 32 single-hunk buggy solutions. Moreover, our manual analysis revealed that although TBar has the required fix patterns for some single-hunk bugs, it still fails to fix them because the concrete implementation of a fixing pattern does not always cover all possible combinations. For example, TBar has the ability to mutate a literal, variable or expression, but it does not support generating a non-existent element which is required by the `makeFancyString` example shown in Figure 3.

**Instructions for Codex-e.** Section 5 shows the performance of Codex-e in fixing unsolved solutions with edit instructions constructed by three different strategies. Compared to Codex-e<sub>bug</sub> and Codex-e<sub>line</sub>, Codex-e<sub>stm</sub> generates the least number of correct fixes. Although the number of fixed solutions by Codex-e<sub>bug</sub> and Codex-e<sub>stm</sub> are quite close (15 versus 16 bugs), the fixed defect category varies. Codex-e<sub>bug</sub> fixes three more multi-hunk bugs, whereas Codex-e<sub>stm</sub> fixes three more single-hunk bugs. Since edit instruction like `Fix bug in the program` does not indicate a specific edit target, Codex-e may search for the statements to edit across the entire program based on its learned knowledge. This encourages generation of large patches but also may lose precision when fixing single-hunk bugs that only require simple fixes. In contrast, Codex-e<sub>stm</sub> is provided with a code context, which steers the edit to the direction that change the most relevant code context. In another perspective, we can also regard Codex-e<sub>stm</sub> and Codex-e<sub>line</sub> as test-based APR tools that fix bugs based on fault localization given by test cases, whereas Codex-e<sub>bug</sub> generates edit without guidance. As a test-based APR approach like Codex-e<sub>stm</sub> outperforms Codex-e<sub>bug</sub>, we think that test-based APR approach is more effective than an approach without any test guidance. Overall, patches produced by Codex-e rely heavily on the types of provided edit instruction. Encoding the suspicious code context into the instruction performs better at fixing small bugs while general instruction without any location guidance may find more complex and larger edits. In future, it is worthwhile to study how to construct edit instructions to guide Codex-e in generating more correct fixes.

**Comparison between TBar, Recoder and Codex-e<sub>stm</sub>.** To analyze the types of defects fixed by each tool and the reasons behind the effectiveness of each APR approach, we compare the patches produced by different tools. As our study in Section 5 shows that Codex-e<sub>stm</sub> gives the best results among all the evaluated strategies, we select Codex-e<sub>stm</sub> for comparison with other APR tools. Figure 5 shows a Venn diagram to better illustrate the set of commonly and uniquely produced patches by these three tools (TBar, Recoder, and Codex-e<sub>stm</sub>). We denote the set of patches produced by TBar as TBar, the set of patches produced by Recoder as Recoder, and the set of patches produced by Codex-e<sub>stm</sub> as Codex-e<sub>stm</sub>. We can observe from Figure 5 that the set TBar is a subset of Codex-e<sub>stm</sub> ∪ Recoder.

In fact, the set TBar is almost subsumed by the set Recoder with the only exception being the defect S-O-1 where Recoder fails to mutate a relational operator `<` into `!=` . This is due to the limitations of pattern-based approaches discussed earlier.

If we compare Codex-e<sub>stm</sub> and Recoder, both approaches correctly fix 10 solutions in common, while Codex-e<sub>stm</sub> has eight more unique patches and Recoder has three more unique patches. We think that Codex-e<sub>stm</sub> outperforms Recoder because: (1) Codex-e<sub>stm</sub> can produce complex patches at flexible locations (as shown in the example given in Figure 4); and (2) Codex-e<sub>stm</sub> is trained on a much larger dataset than Recoder (Recoder uses 82868 human patches to train the repair model), which helps Codex-e<sub>stm</sub> to learn more fix patterns as shown in the example in Figure 6 (Codex-e<sub>stm</sub> generates a patch that includes a lambda expression).

Despite being trained with less data, Recoder still produces three unique patches. Figure 7 shows one of the uniquely fixed solutions by Recoder. We think that Recoder can generate this correct fix due to its syntax-guided decoder that can guide it to copy the statement at line 6 and insert this statement at line 3 of Figure 7 (this...
invokes the copy operation of Recoder that copies the AST sub-tree rooted at the \texttt{set.remove(i)} statement. In another example (S-E-10) uniquely fixed by Recoder, it correctly replaces a branch condition of the form if \((a \&\& b)\) with if \((a)\) (which is also an AST edit operation). These examples show that \textit{encoding AST information into the deep learning model may help in generating correct patches.} In future, researchers can consider incorporating AST information into large language model like Codex-e and AlphaCode to guide its patch generation.

We also find two buggy solutions (S-O-3, S-O-4) where all three APR approaches successfully generate correct fixes, but they are not efficient enough to be accepted by LeetCode (“Time Limit Exceeded” error reported by LeetCode during submission), so we did not include them in Table 3, 4 and 5.

7 \textbf{THREATS TO VALIDITY}

\textbf{External}. During the defect categorization, one author of the paper first manually construct the “ground truth patch”, and then discuss with the other authors to resolve any unclear categorization (e.g., when multiple fixes exist for a bug). For cases where only one fix exists for the bug, the defect classification is straightforward (we classify the defect based on the fix). As the performance of the Codex model and repair tools may varies in different settings, our experiments may not generalize beyond the studied configurations and other programming languages beyond Java. We mitigate this threat by reusing configurations given in prior work, and evaluating on several APR tools that use different algorithms (e.g., search-based and learning-based). Although other large language models (e.g., AlphaCode [22]) exist, our study only evaluates on the Codex language model and the Codex edit mode. The reported findings may not generalize beyond the studied model. As the underlying algorithm used in Codex-e has not been documented, we only use it as a black-box APR tool that produces patches by editing existing programs. To ensure that the training data does not overlap with the evaluated tasks, we have confirmed with the developer of Codex-e that both Codex and Codex-e use the same dataset for training. Nevertheless, our experiments show that the Codex-e model is able to generate fixes for many unsolved solutions.

\textbf{Internal}. Our automated scripts may have bugs that can affect our reported results. To mitigate this threat, we will make our scripts available upon acceptance.

8 \textbf{RELATED WORK}

In this section, we present existing APR techniques that are relevant to our study and the language models for code generation.

\textbf{Automated Program Repair}. Automated Program Repair (APR) has gained a lot of attentions from both academia and industry in recent years [11]. The widely-studied APR techniques include search-based, semantic-based and learning-based APR.

\textit{Search-based APR}. Search-based APRs [16, 24, 25, 28, 36, 37, 41, 47], such as GenProg [40], takes a buggy program and a correct criteria as inputs, and generates patches in two steps: (1) generating a candidate patch space using predefined code transformation operators; and (2) searching for the correct patch over the patch space that satisfies the given correctness criteria. Search-based repair is able to fix many real-world bugs and can scale to large programs. However, the patch space is limited by the pre-defined fix patterns. At the same time, defining large number of fix patterns causes the search space explosion problem. In this work, we studied the effectiveness of search-based APR in fixing auto-generated code.

\textit{Learning-based APR}. The application of deep learning techniques in program repair has been explored in past few years. DeepRepair [42] and DeepFix [13] are the early attempts to fix bugs by learning fixes from similar code. SequenceR [10] adapts neural machine translation (NMT) to generate patch, where CoCoNuT [26] and CURE [17] further improve the results by either encoding program context or using a programming language model. DLFix [23] use two-layer tree-based RNN to learn code transformations, and Recoder [48] designed a syntax-guided learning approach to improve the decoder of a DL model. In this work, we select Recoder because it fixes the most number of bugs in Defects4J [19] among those DL-based APR tools with the trained model released.

\textit{Semantic-based APR}. Semantic-based APR techniques, such as SemFix [30], Nopol [43], and Angelix [29], generate patches by (1) formulating a repair constraint that needs to be satisfied by a program passing a given test-suite; and (2) solving the repair constraint to generate patches. In this work, we have only studied search-based and learning-based repair tools; semantic based APR tools have not been covered in our work.

\textbf{Large Language Model for Code Generation}. Large language models such as GPT-3 [6] have shown promising performance in the NLP domain. Hendrycks et al. [14] proposed APPS dataset and evaluated the code generation performance of several variant GPT models with APPS as the fine-tuned data. Later, Codex [9], the back-end model that powers GitHub Copilot improved the results by fine-tuning GPT-3 with much larger training data in GitHub. AlphaCode [22] is similar to Codex, but focus more on producing program solutions for difficult programming tasks. There are also emerging works combining program synthesis with large language model [5, 15, 34]. The most relevant papers to us are studies on how language model can fix bugs [31, 32]. In contrast, we evaluated whether APR tools (including Codex-e) can fix programs automatically produced by Codex.
9 CONCLUSION
Although large language models have shown promise towards auto-coding, the low success rate of existing language models remains an open problem. In this paper, we study the mistakes made by auto-generated programs by Codex, and investigate whether existing APR tools and the newly released Codex-e can fix the auto-generated buggy programs. Our experiment results reveal that: (1) program produced by Codex share common defect categories as human programmers; (2) existing APR techniques (TBar and Recoder) do not perform well at fixing bugs in auto-generated programs (3) given proper instructions (information from fault localization), Codex-e shows promising initial results in code edit generation, which outperforms TBar and Recoder by fixing 45% more buggy solutions. The implications of our study include: (1) enhancing language models using traditional software engineering techniques (e.g., AST information, fault localization), (2) highlighting the limitations of APR techniques, especially pattern-based approaches, and (3) suggesting future direction of APR research (e.g., flexible form of fault localization). The ability to generate multi-hunk patches at flexible locations can be a milestone for future (learning-based) APR research.

REFERENCES
[1] 2022. Codex Model. https://beta.openai.com/playground
[2] 2022. LeetCode Contest. https://leetcode.com/contest
[3] Rui Abreu, Peter Zoeteweij, and Arjan JC Van Gemund. 2007. On the accuracy of spectrum-based fault localization. In Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering, 602–614.
[4] Yi Li, Shaohua Wang, and Tien N Nguyen. 2020. DBEx: Context-based code transformation learning for automated program repair. In Proceedings of the 2020 IEEE/ACM 42nd International Conference on Software Engineering, 602–614.
[5] Kui Liu, Anil Koyuncu, Dongsun Kim, and Tegawendé F Bissyandé. 2019. Avatar: Fixing semantic bugs with fixes of static analysis violations. In 2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER). IEEE, 1–12.
[6] Kui Liu, Anil Koyuncu, Dongsun Kim, and Tegawendé F Bissyandé. 2019. TBar: Revisiting template-based automated program repair. In Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis. 31–42.
[7] Thibaud Latelier, Hung Viet Pham, Lawrence Pang, Yitong Li, Mosli Wei, and Lin Tan. 2020. Coconut: combining context-aware neural translation models using ensemble for program repair. In Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis. 101–114.
[8] Mateis Martinez and Martin Monperrus. 2016. ASTR: A Program Repair Library for Java (Demo). In Proceedings of the 25th International Symposium on Software Testing and Analysis (SST’16). ACM, New York, NY, USA, 441–444. https://doi.org/10.1145/2931037.2948705
[9] Sergey Mchedchava, Xiang Gao, Shin Hwei Tan, and Abhik Roychoudhury. 2018. Test-equivalence Analysis for Automatic Patch Generation. ACM Transactions on Software Engineering and Methodology (TOSEM) 27 (2018). Issue 4.
[10] Sergey Mchedchava, Jooyong Yi, and Abhik Roychoudhury. 2016. Angelix: Scalable multi-line program patch synthesis via symbolic analysis. In Software Engineering (ICSE), 2016 IEEE 38th International Conference on. IEEE, 691–701.
[11] Hoang Duong Thien Nguyen, Danwei Qi, Abhik Roychoudhury, and Satis Chandza. 2013. SemFix: Program Repair via Semantic Analysis. In Proceedings of the 2013 International Conference on Software Engineering (San Francisco, CA, USA) (ICSE ’13). IEEE Press, Piscataway, NJ, USA, 772–781. https://doi.org/10.1145/2468788.2468690
[12] Hannah Pearce, Benameche Abdah, Ramesh Karri, and Brendan Dolan-Gavitt. 2021. Can OpenAI Codex and Other Large Language Models Help Us Fix Security Bugs? arXiv preprint arXiv:2112.02125 (2021).
[13] Julian Aron Prenner and Romain Robbes. 2021. Automatic Program Repair with OpenAI Codex: Evaluating OpenAI’s Codex-gami Dataset. arXiv preprint arXiv:2111.09222 (2021).
[14] Ruchir Purpi, David J Chung, and Reem Alsharif. 2021. Project codenet: a large-scale AI for code dataset for learning a diversity of coding tasks. arXiv Available at: https://arxiv.org/abs/2105.00245 (2021).
[15] Kia Rahmani, Mohammad Raza, Sumit Gulwani, Vu Le, Daniel Morris, Arjun Radhakrishna, Gustavo Soares, and Ashish Tiwari. 2021. Multi-modal Program Inference: a Marriage of Pre-trainedLanguage Models and Component-based Synthesis. arXiv preprint arXiv:2109.02445 (2021).
[16] Edward K Smith, Earl T Barr, Claire Le Goues, and Yuryi Brun. 2015. Is the cure worse than the disease? overfitting in automated program repair. In Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, 532–543.
[17] Shin Hwei Tan, Zhen Dong, Xiang Gao, and Abhik Roychoudhury. 2018. Repairing Crashes in Android Apps. In Proceedings of the 46th International Conference on Software Engineering (ICSE) ’18. ACM, New York, NY, USA, 602–614. https://doi.org/10.1145/3180155.3180243
[18] Naman Jain, Skanda Vaidyanath, Arun Iyer, Nagarajan Natarajan, Suresh Putharasahaya, Sriram Rajamani, and Rahul Sharma. 2021. Jigsaw: Large Language Models meet Program Synthesis. arXiv preprint arXiv:2112.02969 (2021).
[39] Shin Hwei Tan, Hiroaki Yoshida, Mukul R Prasad, and Abhik Roychoudhury. 2016. Anti-patterns in search-based program repair. In Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering. 727–738.

[40] Westley Weimer, ThanhVu Nguyen, Claire Le Goues, and Stephanie Forrest. 2009. Automatically finding patches using genetic programming. In IEEE/ACM International Conference on Software Engineering (ICSE).

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

[42] Martin White, Michele Tufano, Christopher Vendome, and Derick Poshvanyk. 2016. Deep learning code fragments for code clone detection. In 2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE). IEEE, 87–98.

[43] J. Xuan, M. Martinez, F. DeMarco, M. Clement, S. Lamelas Marcote, T. Darieux, D. Le Berre, and M. Monperrus. 2016. Nopol: Automatic Repair of Conditional Statement Bugs in Java Programs. IEEE Transactions on Software Engineering PP, 99 (2016), 1–1.

[44] Michihiro Yasunaga and Percy Liang. 2020. Graph-based, self-supervised program repair from diagnostic feedback. In International Conference on Machine Learning. PMLR, 10799–10808.

[45] Jooyong Yi, Umair Z Ahmed, Amey Karkare, Shin Hwei Tan, and Abhik Roychoudhury. 2017. A feasibility study of using automated program repair for introductory programming assignments. In Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering. 740–751.

[46] Jooyong Yi, Shin Hwei Tan, Sergey Mechtaev, Marcel Böhm, and Abhik Roychoudhury. 2018. A correlation study between automated program repair and test-suite metrics. Empirical Software Engineering 23, 5 (2018), 2948–2979.

[47] Yuan Yuan and Wolfgang Banzhaf. 2018. Arja: Automated repair of java programs via multi-objective genetic programming. IEEE Transactions on software engineering 46, 10 (2018), 1040–1067.

[48] Qihao Zhu, Zeyu Sun, Yuan-an Xiao, Wenjie Zhang, Kang Yuan, Yingfei Xiong, and Lu Zhang. 2021. A syntax-guided edit decoder for neural program repair. In Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. 341–353.