Automated Repair of Programs from Large Language Models

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

Xiang Gao†
Beihang University
Beijing, China
xiang_gao@buaa.edu.cn

Martin Mirchev
National University of Singapore
Singapore
mmirchev@comp.nus.edu.sg

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

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

2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE) | 978-1-6654-5701-9/23/$31.00 ©2023 IEEE | DOI: 10.1109/ICSE48619.2023.00128

***Abstract*—Large language models such as Codex, have shown the capability to produce code for many programming tasks. However, the success rate of existing models is low, especially for complex programming tasks. One of the reasons is that language models lack awareness of program semantics, 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 reliability in the code produced by large language models. Our study revealed that: (1) automatically generated code shares common programming mistakes with human-crafted solutions, indicating APR techniques may have potential to fix auto-generated code; (2) given bug location information provided by a statistical fault localization approach, the newly released Codex edit mode, which supports editing code, is similar to or better than existing Java repair tools TBar and Recoder in fixing incorrect solutions. By analyzing the experimental results generated by these tools, we provide several suggestions: (1) enhancing APR tools to surpass limitations in patch space (*e.g.*, introducing more flexible fault localization) is desirable; (2) as large language models can derive more fix patterns by training on more data, future APR tools could shift focus from adding more fix patterns to synthesis/semantics based approaches, (3) combination of language models with APR to curate patch ingredients, is worth studying.**

I. 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 can be used to achieve impressive performance in generating text. The transformer-based models, such as Codex [1] and AlphaCode [2], have successfully generated code for many programming tasks in Python, Java, and C. Technically, these techniques treat code generation as a transformation problem, which takes as input natural language descriptions and transforms them into programming language.

Although transformer-based models successfully solved many programming tasks, their success rate is still relatively low. When evaluating on *pass@5* metric [1], the best Codex model achieves 24.52% passing rate at introductory-level tasks and 3.08% passing rate at competition-level tasks [1] from APPS dataset [3]. The best AlphaCode model achieves 20.36% and 7.75% passing rates on introductory-level and competition-level tasks, respectively [2]. 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. In contrast, generating entire programs requires an understanding of the entire task's description which usually comprises complex logic, and figuring out the solutions to programming tasks relies on deep algorithm reasoning. Although it is important to systematically study the reasons behind the ineffectiveness of language models in solving programming tasks, there is little to no study that characterizes the defects made in the programs automatically generated by language models, creating a gap in understanding how to further improve these automatically generated programs.

Automated program repair (APR) is an emerging area for automated rectification of programming errors [4]. APR 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 [5], Angelix [6]) generate patches via symbolic execution, while search-based repair tools (e.g., GenProg [7], TBar [8]) search for correct patches among a pre-defined search space. APR has shown promising results in fixing real-world bugs but they are still limited to generating small patches (usually one-line fixes) due to the complexity of semantic reasoning, and search space explosion – when considering multi-line fixes.

The strength and weakness of language models and APR techniques inspire us to think about the following question:

†Corresponding author

1558-1225/23/$31.00 ©2023 IEEE
DOI 10.1109/ICSE48619.2023.00128

TABLE I
OUR KEY FINDINGS 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 III)	Implications
Auto-generated code share common mistakes with human programmers. 57% of bugs made by Codex are algorithm-related, and 11% of them are due to syntax errors. To fix the remaining bugs made by Codex. 13.4% of them require small changes (e.g., changing operator and replacing variables), 18.5% of them require larger patches.	As Codex generated code share common mistakes with human-written code, using APR techniques to enhance reliability in auto-generated code by automatically fixing the bugs in the auto-generated code is worth studying.
Auto-generated code contain negative symptoms or undesirable code patterns such as: ① names that indicate wrong algorithms; ② similar code blocks (code smells related); ③ producing irrelevant helper functions.	Instead of depending on token log-probability, language model designers can consider incorporating rigorous code quality checks from the perspective of a program itself to enhance code generation and recommendation.
Findings on APR's effectiveness (Section IV)	**Implications**
Existing pattern-based and learning-based APR approaches can fix a small number of bugs in auto-generated code. The challenges in fixing auto-generated code include: ① limited search space; ② unable to generate multi-edit patches; ③ lack of awareness of program dependencies.	Manually designing fix patterns is not scalable, and future research may either need to look more at program synthesis based approaches, or need to curate patterns automatically from huge training data.
	Statistical fault localization is widely used by APR tools to determine fix location, which might be limited. Advanced fix localization techniques based on program dependency analysis may help to improve APR's repairability.
Findings on Codex Edit Mode (Codex-e) (Section V)	**Implications**
Given "proper" instructions (such as where to fix), Codex-e even outperformed pattern-based and learning-based APR tools. With/Without controlling the fault locations affect the characteristics of generated patches by Codex-e. So, "what kind of guidance should be given to Codex-e?", needs to be further studied.	Considering the similar effectiveness of Codex-e with and without location guidance, future APR research should strike a balance between controlling the fault location and providing flexibility in the fault location (allowing it to generate multi-hunk patches). This work would be along the lines of engineering prompts for language model based code generators.
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.	Future APR tools could explore more flexible forms of fix localization to allow fixes to be generated at multiple locations.
Findings on Combining Search Space of different Tools (Section V)	**Implications**
Combining the search space of different tools (TBar and Codex-e) could produce the required patch ingredients to fix more incorrect solutions.	Combination of APR tools with language model based tools, for curating patch ingredients (possibly via symbolic analysis of code fragments) — is worth studying. One can combine multiple incorrect solutions produced by Codex to get more patch ingredients. APR tools can consider using Codex as a source of crafting rich patch ingredients.

can automated program repair improve 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 are common in auto-generated code?

Although we know that language models produce many wrong solutions when solving programming tasks, several open questions remain: (i) what are the types of bugs made by language models; (ii) are the bugs made by language models similar to the bugs made by human. We first study the bug patterns of code produced by Codex, and whether they are similar to bugs in human-written code.

(RQ2) Can APR tools effectively fix code from Codex?

Existing APR tools are mainly designed to fix human-written bugs. APR tools typically generate patches by defining transformation operators (search-based APR) or specifying the program synthesis ingredients (semantics-based APR). These operators and ingredients have been proven to be efficient in fixing human-written bugs. We study how effective APR tools (TBar and Recoder) 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 [9], which can edit existing content in a complete program rather than just completing a partial program. 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" [9]. To fix a bug, users need to provide precise and clear instructions. How to automatically produce such instructions still remains an open question. We study whether the side effect of APR tools, such as fault localization results, can be used to guide Codex-e, and how effective Codex-e is in fixing program bugs.

Table I presents the key findings of our study. Our result shows that existing APR tools (pattern-based and learning-based APR) are still quite limited, including limited patch space, fix locations and patch size — thus enhancing APR tools to surpass these limitations (e.g., introducing a more flexible fault localization strategy) is highly desirable. Specifically we see possible collaboration between APR tools and Codex-e for curating patch ingredients to construct complex patches.

Contributions: The contributions of this paper are:

- We present a systematic study of automated repair of buggy programs produced from language models.
- To the best of our knowledge, we conduct the first study that evaluates the efficacy of the newly released Codex edit mode as an automated repair tool.
- We propose LMDefects, a new dataset that contains 113 Java programming tasks. Among them, 46 tasks have been successfully solved by Codex and 67 of them remain unsolved. Our dataset and scripts, including all initial solutions produced by Codex and all patches produced by APR tools, is available at *https://github.com/zhiyufan/apr4codex*.

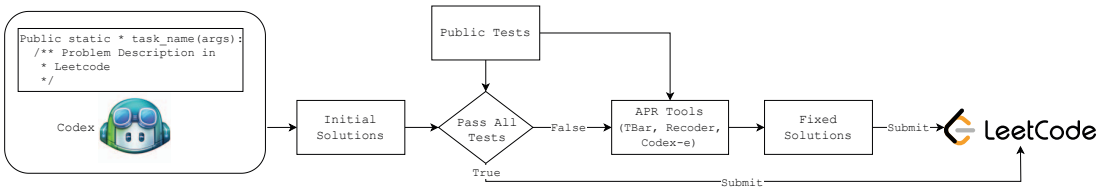

Fig. 1. The workflow of automatically fixing programs generated by Codex

II. 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 were conducted on an Ubuntu-16.04 server, with 64GB RAM and Intel(R) Xeon(R) CPU E5-2660 @ 2.00GHz, and NVIDIA Titan V GPU.

*a) **Codex Model**:* Codex [1] is the model that powers GitHub Copilot [10] which completes a program given a natural language prompt. Codex supports many programming languages (e.g., Python, C/C++, Java). 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 and Codex-e code-davinci-edit-001 model [11], which were both trained on data up to Jun 2021.

*b) **Methodology and Dataset**:* Figure 1 shows the overall workflow of our study. LeetCode is an online judge platform, which includes over 2,300 different problems, ranging from easy to hard level. It also has a forum [12] with active community where correct solutions for each programming task can be found (important for our manual analysis of incorrect solutions). We first use Codex to generate initial solutions for each task and validate the correctness of generated solutions on *public example test cases*. For each unsolved programming task, existing test-based repair tools (using the public tests) are then applied to fix the incorrect solutions produced by Codex. The patched solutions are then validated using (1) the public tests, and (2) the held-out (private) tests in the LeetCode platform. To answer our research questions, we build a dataset LMDefects with 113 programming tasks in LeetCode [13]. Each task is described using natural language text accompanied with *1–3 public tests* that provide examples with pairs of (input, output). When a solution is submitted to LeetCode, it runs a set of private tests to validate the correctness of submissions. LeetCode has weekly and biweekly contests, where it releases new programming tasks. In our study, we only consider easy-level and medium-level problems because Codex fails to solve most hard problems [1] (we also exclude seven tasks that require customized data structures that Codex is unlikely to handle). To prevent the case where the collected dataset was used 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 60 easy and 53 medium-level programming tasks. Several datasets with programming tasks exist [1], [2], [3], [14], [15], [16]. They are either based on contests from programming competition platforms (e.g., Codeforces) or hand-written programming tasks. We do not use existing datasets because (1) Codex was already trained on GitHub where solutions for many previous programming tasks exist (e.g., APPS, CodeContest) (2) some programming tasks do not have public tests which is a prerequisite for APR techniques (e.g., HumanEval [1]), (3) most APR tools only support Java programs, whereas the HumanEval dataset curates Python programs.

*c) **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 represents the programming task description and the 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*) but we feed the public tests in the natural language description to guide APR tools instead of embedding them in the prompt. Lines 1–3 in Figure 2 presents an example prompt that we use for a LeetCode programming task [1]. Given such a prompt, we run Codex to generate 50 candidate solutions and select the top five solutions with the highest probability of being correct (via the *best_of* parameter of Codex). The selected solutions are first validated by running against the public tests (e.g., lines 14–17 in Figure 2, we manually convert the public tests into JUnit tests), and then validated against private tests by submitting them to LeetCode. We reuse the same setting as prior work [1]. Specifically, we (1) set temperature to 0.8 (which has the best performance when generating 50 candidate solutions), and (2) prepare the stop sequences to be "public", "class", "//", "System.out.print". We set the "max token length" to 2048.

*d) **APR Tools**:* To evaluate whether repair tools can fix the incorrect solutions produced by Codex, we evaluate on two Java APR tools 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 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 Defects4J [17] benchmark (almost all Java APR tools

[1] https://leetcode.com/problems/minimum-operations-to-convert-number/

```
1  public static int minimumOperations(int[] nums) {
2 /** You are given a 0-indexed array nums consisting
      of n positive integers. ...
3  * Return the minimum number of operations required
      to make the array alternating.*/
4   int n = nums.length;
5   int[] dp = new int[n];
6   dp[0] = 0; dp[1] = 1;
7   for (int i = 2; i < n; i++){
8     dp[i] = dp[i - 1] + 1;
9     if(nums[i] == nums[i - 2])
10      dp[i] = Math.min(dp[i - 2] + 1, dp[i]);
11  }
12  return dp[n - 1];
13 }
14 // Public Test 1:
15 // Input: nums = [3,1,3,2,4,3], Output: 3
16 // Public Test 2:
17 // Input: nums = [1,2,2,2,2], Output: 2
```

Fig. 2. An example of misaligned algorithm, taken from LeetCode programming task *minimumOperations* on weekly-contest-280.

have been evaluated on it). Since both TBar and Recoder are test-driven APR tools, we use the public test cases given in the program description to guide the repair, while the private test cases are applied to validate the patched solutions. We run TBar and Recoder in default settings, and the repair process stops if a patch that passes all public tests is found. We set the timeout to 15 minutes, following the time limit used in prior work on automated repair of programming assignments [18]. As Codex edit mode (Codex-e) can modify existing code by generating program edits, we investigate whether Codex-e can serve as an APR tool and compare it with TBar and Recoder.

III. RQ1: What mistakes are common in auto-generated code?

Before we apply APR techniques in fixing the automatically generated solutions, we investigate its feasibility by analyzing the typical mistakes made in solutions produced by Codex.

Given a programming task in LMDefects for Codex to solve, we first run the five auto-generated solutions on the public tests and submit them to LeetCode online judge platform to validate using private tests. If an auto-generated solution s by Codex fails to pass all public and private tests, we consider s an *incorrect solution*. If all the five auto-generated solutions for a programming task are incorrect solutions, we consider this task as *unsolved*. Overall, 46 programming tasks can be *solved* by Codex. We study the mistakes of 335 *incorrect solutions* S_{buggy} in the remaining 67 *unsolved* programming tasks that lead to compilation errors or test failures.

For each incorrect solution $s_{buggy} \in S_{buggy}$, two annotators (two authors of the paper) separately and manually fix it by first referring to other solutions in LeetCode discussion forum for repair hints, and then constructing a minimal patch that fixes the bugs. The constructed patch for each incorrect solution is cross-validated by the two annotators who make sure the patched solution s_{fixed} is accepted by LeetCode platform. Our goal is to construct a "ground truth" patch s_{fixed} for each incorrect solution to obtain the "diff" between s_{buggy} and s_{fixed}. Based on this "diff", two authors manually classify each s_{buggy} using the defect categories in Table II. Each s_{buggy} is assigned to one defect category. If there is any disagreement during the ground truth construction or defects classification, annotators discuss with other authors to resolve the disagreement (there were 14 initial disagreements, all of which were successfully resolved).

We derive the defect classification based on categories used in Codeflaws [14] (a benchmark that contains incorrect submissions by participants in programming competitions). The detailed classification and their definitions are shown in Table II. It also shows the number of incorrect solutions (both "Easy" and "Medium") belongs to each category. The example code of each defect category can be found in the supplementary material. The defect classification in auto-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. We think this is expected because Codex is trained with a lot of human-written programs that can be potentially buggy. Besides the above single and multi-hunk bugs, syntax errors and algorithm-related errors are prevalent in Codex generated solutions. We manually analyzed these solutions to study the root causes behind these errors.

Syntax Errors. Our manual analysis revealed that auto-generated programs that lead to compilation errors usually have (1) incomplete code, or (2) invoking undefined variables/functions/classes. To reduce the likelihood of Codex in generating incomplete code, we select the maximum token length allowed (i.e., 2048 tokens) by Codex for generating 50 candidate solutions. Despite providing the maximum length as the bound for code generation, Codex still generates incomplete code where the average token length is 628. It is worthwhile to study the feasibility of applying code completion techniques for fixing the auto-generated incomplete code by Codex. Meanwhile, for programs with undefined functions, one needs to synthesize the function body to resolve the compilation errors. *Future research can work on using program synthesis techniques to resolve the undefined functions or invoking Codex on a function-by-function basis to synthesize the function body*. Apart from these compilation errors, we also observe that Codex is prone to generate programs which fail to compile due to a missing/extra close bracket at the end of the program (in total, there are 23 of these cases). Since bracket mismatch can be fixed easily (using a regular expression matching mechanism), we manually fix them and further classify their defects into defect categories in Table II.

Misaligned Algorithm. Among all incorrect solutions, 191 solutions use wrong algorithms to solve the given tasks, including TLE (Time Limit Exceeded). The problem of generating solutions that do not meet the user intention is known as the *misalignment* problem [1]. All defects classified as "misaligned algorithm" suffer from the misalignment problem.

Negative Symptoms in Auto-generated code. Auto-generated patches are known to exhibit certain *anti-patterns*

TABLE II
DEFECT CLASSIFICATION OF INCORRECT SOLUTIONS

Defect Category	Sub-category	Definition	Easy	Medium	Total
Multi-hunk	(M-S) **S**imilar	Similar single-hunk bugs (require similar fixes) exist at multiple discontinuous program locations	7	2	9
	(M-U) **U**nique	Distinct single-hunk bugs exist at multiple discontinuous program locations, and the total lines of patches are no more than five lines	19	20	39
	(M-L) Need **L**arge Fix	The bug is (1) neither M-S or M-U and (2) needs to edit more than five lines at multiple locations	5	9	14
Single-hunk [19]	(S-O) **O**perator Mutation	Replace arithmetic/logical/relational/bitwise operator with another operator or insert/delete operators and relevant operands or modify operator precedence	7	5	12
	(S-C) **C**onstant Mutation	Replace constant (not in array or function call) with a variable/constant/function call	3	0	3
	(S-V) **V**ariable Mutation	Replace variable (not in array or function call) with a variable/constant/function call	2	0	2
	(S-A) **A**rray Mutation	Replace the array access with other constant/variable, operands with arithmetic operators, or replace an array with another array	1	0	1
	(S-F) **F**unction Call Mutation	Replace function call with another function call or change function arguments	2	1	3
	(S-AS) **A**dd **S**tatements	Insert a continuous chunk of statements	8	1	9
	(S-DS) **D**elete **S**tatements	Delete a continuous chunk of statements	2	3	5
	(S-HO) **H**igher **O**rder	A single-hunk patch that combines multiple single-hunk bugs	5	5	10
Algorithm-related	Misaligned Algorithm	The algorithm used is misaligned with the requirement given in the task description	30	161	191
Syntax Error	Incomplete Code	For the last line of the program, only parts of the program is printed	7	17	24
	Invoke Undefined Program Elements	Fails to compile due to invoking undefined variables/functions/classes	2	11	13
Total	-	-	100	235	335

(program transformations leading to nonsensical patches) [20]. Inspired by this, we analyze whether the code generated by Codex contains negative symptoms that are unlikely to be correct programs by studying the "Misaligned Algorithm" and "Syntax Error" categories:

① ***Names indicate Wrong Algorithms:*** In the "Misaligned Algorithm" category, Codex is prone to generate solutions with certain variable names which points to the underlying algorithm/data structure used is wrong. Figure 2 shows an example of using "dp" variable, where the algorithm used (i.e., "dp" refers to dynamic programming) is incorrect. We think Codex solves the task with dynamic programming because it is misled by other programs named "minimumOperations" but for a different programming task in GitHub. Similarly, we observe other variable names such as "pq" (priority queues), "q" (queue) that indicate the wrong data structures being used.
② ***Similar Code Blocks:*** We notice that when Codex struggles to find a high quality solution for a given prompt, it tends to repeatedly generate similar code blocks (code clones with minor variations in variable names, statement usages, and control structures). Figure 3 shows a solution with similar code blocks (only variable names differ) at lines 5–8 and 9–12.
③ ***Irrelevant Helper Functions:*** Although we reuse Codex's setting in adding stop sequences (for terminating code generation once we reach the end of a function), we observe that Codex is still likely to produce redundant helper functions which are irrelevant to the given prompt.

```
1  public static int minimumSum(int num) {/** ...*/
2   String str = "" + num;
3   int first = Integer.MAX_VALUE;
4   int second = Integer.MAX_VALUE; ...
5  ++ if (firstNum.length() == 1) {
6  ++  first=firstNum.charAt(0) - '0';
7  ++ } else {
8  ++  first=Integer.parseInt(firstNum.toString());}
9  ++ if (secondNum.length() == 1) {
10 ++  second=secondNum.charAt(0) - '0';
11 ++  } else {
12 ++  second=Integer.parseInt(secondNum.toString());}
13 ...
```

Fig. 3. An example of generating similar code block (highlighted with "++"), taken from LeetCode programming task *minimumSum*.

Auto-generated programs share common mistakes with human-written programs, and contain certain negative symptoms including: (1) names indicate wrong algorithms; (2) similar code blocks; (3) irrelevant helper functions.

IV. RQ2: HOW EFFECTIVE ARE APR TOOLS IN FIXING THE CODE PRODUCED BY CODEX?

Given the 298 compilable incorrect solutions by the Codex model, 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 incorrect solutions pass the given public tests.

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

Table III shows the number of generated patches and the number of correctly fixed programming tasks by TBar and

TABLE III
THE NUMBER OF PATCHES AND FIXED TASKS PRODUCED BY TBAR AND RECODER (INCLUDE BOTH SINGLE-HUNK AND MULTI-HUNK)

Tool	Correct/Plausible patches		Correctly Fixed Tasks	
	easy	medium	easy	medium
TBar	6/16	3/22	3	3
Recoder	6/16	5/20	3	5

```
1  // task delete-characters-to-make-fancy-string
2  public static String makeFancyString(String s){...
3      if(...) {
4        sb.deleteCharAt(i);
5  -       i -= 2;
6  +       i -= 1; // constant mutation (S-C-3)
7      }...}
8  // task watering-plants
9  public static int wateringPlants(int[] plants, int
       capacity) {...
10   if (plants[i] > currWater) {
11     steps += (i - 1) * 2;
12 +   steps++; //add a statement (S-O-10)
13     ...} ...}
```

Fig. 4. Two incorrect solutions fixed by Recoder but not TBar.

Recoder, respectively. Although TBar produces 16 and 22 plausible patches on easy-level and medium-level tasks, it only produces 6 easy and 3 medium correct patches. Compared to TBar, Recoder produces less plausible patches (16 and 20 on easy and medium level, respectively), and more correct patches (6 and 5). The "Correctly Fixed Tasks" columns of Table III show the number of programming tasks correctly fixed by TBar and Recoder. Note that each programming task corresponds to the five selected incorrect solutions. If any of these solutions is correctly fixed (accepted by LeetCode), we consider that this task has been solved. Overall, Recoder fixes eight programming tasks whereas TBar only fixes six tasks. Combining both tools, APR tools help Codex solve four more easy-level and five more medium-level tasks.

We further analyze the type of defects fixed by the two APR tools. Table IV 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). The results show that existing APR tools are still limited in generating complex patches that require edits of multiple lines.

Figure 4 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 patch space. For the second example, TBar fails to generate the correct patch because it does not have the "insert statement" pattern.

For tasks that require multi-line fixes, both TBar and Recoder fail to generate any correct patches, one of the reasons is that the widely adapted statistical fault localization techniques in TBar and Recoder focus on identifying each faulty line separately, without considering program dependency among the suspicious lines. For example, to fix the bug in Figure 5,

TABLE IV
THE NUMBER OF CORRECTLY FIXED SOLUTIONS BY TBAR AND RECODER, REFER TABLE II FOR ABBREVIATION OF DEFECT CLASSIFICATION

Defect Sub-category	Total		TBar		Recoder	
	easy	medium	easy	medium	easy	medium
S-O	7	5	2	2	2	3
S-C	3	-	-	-	1	-
S-V	2	-	1	-	1	-
S-A	1	-	-	-	-	-
S-F	2	1	-	-	-	-
S-AS	8	1	-	-	-	1
S-DS	2	3	2	1	2	1
S-HO	5	5	1	-	-	-
Total (Single-Hunk)	30	15	6	3	6	5
M-S/M-U/M-L	31	31	-	-	-	-

one needs to (1) change $s.length() - 2$ to $s.length()$, and (2) simplify the if-condition. Using statistical fault localization, APR tools will generate patches for line 4 and lines 5–6 separately (without noticing that after fixing the if-condition at lines 5–6, the for-loop condition no longer need the extra "-2" at line 4 to prevent the "IndexOutOfBoundException").

```
1  public static int minimumMoves(String s) {
2      //S-HO-5
3      int count = 0;
4  -   for (int i = 0; i < s.length() - 2; i++) {
5  -   if (s.charAt(i)==s.charAt(i+1) && s.charAt(i+1)
6  -       ==s.charAt(i + 2) && s.charAt(i)=='X') {
7  +   for (int i = 0; i < s.length(); i++) {
8  +   if (s.charAt(i)=='X') {
9        count++;
10       i += 2;
11     }}
12     return count;}
```

Fig. 5. An incorrect solution that should be fixed by modifying line 4 and lines 5–6 together.

Existing pattern based and learning based APR are ineffective at fixing auto-generated code, challenges include: (1) limited search space; (2) unable to generate multi-edit patches; (3) lack of awareness of program dependencies.

TABLE V
THE NUMBER OF CORRECTLY FIXED SOLUTIONS USING CODEX-E, REFER TABLE II FOR ABBREVIATION OF DEFECT CLASSIFICATION

Defect Category	Sub -Category	Total		Codex-e^{bug}		Codex-e^{line}		Codex-e^{stm}	
		easy	medium	easy	medium	easy	medium	easy	medium
Single -Hunk	S-O	7	5	4	3	1	2	2	4
	S-C	3	-	-	-	1	-	1	-
	S-V	2	-	-	-	1	-	1	-
	S-A	1	-	1	-	-	-	-	-
	S-F	2	1	1	-	2	-	2	-
	S-AS	8	1	-	-	-	-	1	-
	S-DS	2	3	2	-	1	-	2	-
	S-HO	5	5	-	-	1	-	1	-
Total	-	30	15	8	3	7	2	10	4
Multi -Hunk	M-S	7	2	1	-	1	-	2	-
	M-U	19	20	1	2	-	1	-	-
	M-L	5	9	-	-	-	-	-	-
Total	-	31	31	2	2	1	1	2	-

V. 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 edit the content of programs, a natural question to ask would be "Can Codex-e fix an incorrect program with proper instructions?" 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) [21], [22] 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 that 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 the suspicious line numbers as instructions. We use the program text of the statements, e.g., *s1* at the suspicious line, and formulate the instruction to Codex-e as "*Fix s1*".

For example, to fix the constant mutation bug for *makeFancyString* in Figure 4, we give Codex-e^{line} the instruction *Fix line 6*, and provide Codex-e^{stm} the instruction *Fix "i -= 2;"*.

For each incorrect solution (we exclude solutions that produce syntax errors as in Section IV), 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 V 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 limited guidance, Codex-e^{bug} successfully produced 15 correct patches where four of these patches involve multi-hunk modifications (refer to supplementary material for the example). In contrast, when giving the faulty line number as instruction, Codex-e^{line} fixes nine solutions that require a single-hunk fix, and two solutions that requires a multi-hunk fixes. 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 guide a language model like Codex in matching relevant statements.

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 [8], [23], [24], [25] 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}, 8 (50%) of them are fixed by editing beyond the statement provided in the given instruction. Figure 6 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} fixes this by moving one *if*-then clause out of the loop body and changing the *if*-condition. Compared to traditional APR tools, using flexible fault localization is an important feature enabling Codex-e to produce more correct patches.

```
1  public static int[][] construct2DArray(int[]
       original, int m, int n) {
2   // Instruction: Fix "for (int i=0; i<result.length
       ; i++){"
3  +if (n*m != original.length)  // S-AS-8
4  +  return new int[0][0];
5   int[][] result = new int[m][n];
6   for (int i=0; i<result.length; i++){
7    for (int j=0; j<result[i].length; j++){
8  -  if (i*result[0].length+j >= original.length)
9  -   return new int[0][0];
10 -  else
11      result[i][j]=original[i*result[0].length+j];}}
12  return result;}
```

Fig. 6. Flexible fault localization example of LeetCode programming task *convert-1d-array-into-2d-array* on biweekly-contest-62 fixed by Codex-e^{stm}

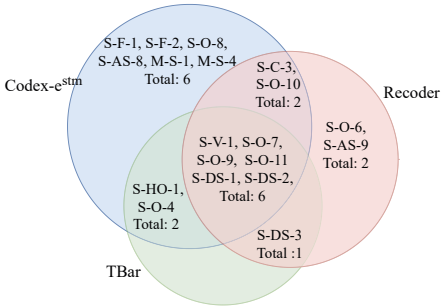

Fig. 7. The repair results of different APR tools

The effectiveness of Codex-e with a given specific fault location (Codex-e^{stm}) is nearly comparable to its effectiveness without any location guidance (Codex-e^{bug}).

A. Comparison between TBar, Recoder and Codex-e^{stm}.

To analyze the types of defects fixed by each tool and the reasons behind the effectiveness of each approach, we compare the patches produced by TBar, Recoder, and Codex-e^{stm}. As our experiments show that Codex-e^{stm} gives the best overall results among all strategies of Codex-e, we select Codex-e^{stm} for comparison with other APR tools. Figure 7 shows a Venn diagram to better illustrate the set of commonly and uniquely produced patches by these three tools. We denote the set of patches produced by TBar as $TBar$, patches produced by Recoder as $Recoder$, and patches produced by Codex-e^{stm} as $Codex$-e^{stm}. As shown in Figure 7, the patches produced by $TBar$ is a proper subset of $Codex$-e^{stm} $\cup$ $Recoder$. In fact, the patches produced by $TBar$ is almost subsumed by the set $Recoder$. This is due to the restricted search space

of pattern-based approaches (discussed in Section IV). If we compare Codex-e^{stm} and Recoder, both approaches share eight common patches, while Codex-e^{stm} has six more unique patches, and Recoder has two more unique patches. We think that Codex-e^{stm} outperforms Recoder because: (1) Codex-e^{stm} can produce complex patches at flexible locations (e.g., Figure 6); and (2) Codex-e^{stm} is trained on a much larger dataset than Recoder (Recoder uses 82868 human patches for training), which helps Codex-e^{stm} learn more fix patterns (e.g., Fig 8 where Codex-e^{stm} uses a lambda expression).

```
1  public static int minimumSum(int num){
2    // (using lambda expression) (S-F-1)
3  - Collections.sort(digits);
4  + Collections.sort(digits, (a, b) -> b - a); }
```

Fig. 8. An example that is uniquely fixed by Codex-e^{stm}

Despite being trained with less data, Recoder still produces two unique patches. Figure 9 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 it at line 3 of Figure 9 (this invokes the copy operation of Recoder that copies the AST subtree rooted at the `set.remove(i)` statement). In another example (S-O-6) 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 encoding AST information into 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*.

```
1  public static List<List<Integer>> findWinners(int
       [][] matches) {  ...
2    for (int i : map.keySet()) {
3  +   set.remove(i); // (S-AS-9)
4      if (map.get(i) == 1) {
5        ans1.add(i);
6        set.remove(i);
7        }  }  ...}
```

Fig. 9. An example that is uniquely fixed by Recoder

B. Combine Patch Space of Different Tools.

a) ***Combine patch space of Codex-e and APR****:* We further study whether the patch search space produced by APR and Codex-e complement each other by evaluating the patch ingredients produced by different tools. The *patch ingredient* is defined to be the set of operators/operands (e.g., variables, literals, operators and etc.) used to construct the corresponding patch. If APR and Codex-e produce patch ingredients that complement each other, their combination will be more likely to generate the correct patch. To do so, for each incorrect solution, we first obtain the *required patch ingredients* $I_{correct}$ by referring to the "ground truth" patch constructed in Section III (i.e., the correct patch is built using ingredients in $I_{correct}$). Then, we investigate the following; we do not consider Recoder+Codex or Recoder+Codex-e since both Recoder and Codex/Codex-e are learning based tools.

TABLE VI
THE NUMBER OF INCORRECT SOLUTIONS THAT TBAR AND CODEX-E CAN PRODUCE ALL REQUIRED PATCH INGREDIENTS, REFER TABLE II FOR ABBREVIATION OF DEFECT CLASSIFICATION

Defect Sub-category	Total	TBar	Codex-e	TBar+Codex-e	TBar+Codex
S-HO	10	1	3	4	5
M-S	9	3	2	4	4
M-U	39	-	-	1	3
Total	58	4	5	9	12

1) Can an individual tool (TBar/Codex-e) produce all required patch ingredients for each incorrect solution?
2) Can combining TBar and Codex-e (run TBar and Codex-e sequentially) produce all required patch ingredients?

Table VI shows the number of incorrect solutions whose required patch ingredients are covered by the patch space of each APR technique. "TBar+Codex-e" represents the combined patch space of TBar and Codex-e. Our results show that combining Codex-e and TBar could successfully generate the required patch ingredients of 9 incorrect solutions (with 2 of them cannot be generated by TBar and Codex-e separately).

Figure 10 shows an incorrect solution that can be fixed by changing $n{>}0$ to $n{!=}0$ **and** inserting a bound check $nums.size(){>}0$. For this incorrect solution, none of the APR tools in our experiment generates a correct fix. However, the two required patch ingredients could be separately produced by TBar and Codex-e. Specifically, TBar fixes the first bug by changing the incorrect operator from ">" to "!=" which makes the solution pass the public tests. When we submit this partially fixed solution to LeetCode, the program still fails by throwing `IndexOutOfBoundsException`. By encoding the error message into the edit instruction ("Fix IndexOutOfBoundsException"), Codex-e successfully fixes the bug by appending the check `nums.size()>0`.

```
1  public static long smallestNumber(long num) {
2    long n = num;
3    ArrayList<Integer> nums = new ArrayList<>();
4  - while(n > 0){ // Fixed by TBar
5  + while(n != 0){
6      nums.add((int)(n % 10));
7      n = n / 10;
8    }
9    Collections.sort(nums);
10 - if(nums.get(0) == 0){ // Fixed by Codex-e
11 + if(nums.size() > 0 && nums.get(0) == 0){
12     for(int i = 1; i < nums.size(); i++){ ...
```

Fig. 10. Combined patch space of TBar and Codex-e

b) ***Combine APR with Multiple Solutions of Codex****:* Codex generates a set of program candidates — each of them may slightly vary in their understanding of the problem description and hence represent slightly different code. We study the feasibility of combining the patch ingredients from these candidates ("TBar+Codex" setting). Table VI shows that "TBar+Codex" is the most effective among the evaluated combinations by producing all required patch ingredients for 12 incorrect solutions. Figure 11 shows an example incorrect solution which requires two patch ingredients. TBar generates the first patch ingredient (i.e., removing the if-branch from

lines 7–9) but the second patch ingredient (adding a for-loop to calculate the sum of absolute value of array `freq`) does not exist in the patch space of any APR technique (including Codex-e). This is mainly because generating such a large and unseen code snippet is not supported by most of the existing APR tools, and Codex-e does not have a relevant hint in instruction. However, Codex produces many candidate solutions that can be used for enriching the patch space. By borrowing the code from other candidate solutions, and modifying the variable name, we can successfully fix the below incorrect solution.

```
1  public static int minSteps(String s, String t) {
2    int[] freq = new int[26];
3    for(char c : s.toCharArray())
4      freq[c - 'a']++;
5    int steps = 0;
6    for(char c : t.toCharArray())
7  -   if(freq[c - 'a'] == 0) // Fixed by TBar
8  -     steps++;
9  -   else
10       freq[c - 'a']--;
11 // Find in another candidate solution of minSteps
12 + for(int fr : freq)
13 +   steps += Math.abs(fr);
14   return steps;
15 }
```

Fig. 11. Obtaining patch ingredients from multiple candidate solutions

Compared to fixing incorrect solutions with only APR techniques, both Codex-e and Codex's multiple solutions could provide required patch ingredients to construct correct fixes.

By using patch ingredients extracted from (1) TBar's and Codex-e's patches, and (2) TBar's patches and multiple generated solutions by Codex — we successfully identify the required patch ingredients of more incorrect solutions.

VI. Implications and Discussions

Our study identifies several important implications and suggestions for the language models and program repair research.

A. Open dataset for language model defects.

To push the limits of the code generation capability of a large language model like Codex, we believe that our systematic investigation of the mistakes made by language models is an important initial step. It would be beneficial to have a community-driven dataset and more analysis of the defects within the dataset to facilitate future improvement of the auto-generated programs. We propose the LMDefects dataset as an initiative towards this direction.

B. Negative symptoms of auto-generated Codex programs.

We have identified several negative symptoms among auto-generated Codex programs, including code that contains: (1) names that indicate wrong algorithms, (2) repeatedly producing similar code blocks, (3) irrelevant function helpers. Moreover, we observed that even after manually fixing all auto-generated Codex programs with syntax errors of bracket mismatches, these programs are still incorrect as they fail to pass the held-out tests in LeetCode.

C. Use of function names in auto-generated code.

Based on our manual analysis of the generated solutions, Codex seems to rely heavily on the function name for solving the programming tasks (e.g., `minimumOperations` in Figure 2) . In fact, a recent study has also observed the tendency of Codex in generating solutions based on function name [26]. Compared to the long prompt (function signature and the problem description), the function name is more concise and easier to search in GitHub. However, this strategy fails when a customized algorithm is required to solve a programming task. Relying on the function's 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. *Future language models designed for code generation should focus on summarizing useful information from problem description to reduce reliance on function names.*

D. Pattern-based APR versus learning-based APR.

Section IV shows that Recoder generates a few more correct fixes than TBar. The reasons are that pattern-based APR requires (1) additional fix patterns, or (2) a large search space for fix ingredients (e.g., specific literal). Figure 4 shows an example that can be uniquely fixes by Recoder by adding a statement $steps++;$ at line 12, which is not supported by TBar. However, Figure 7 shows that TBar also uniquely fixes two solutions where Recoder fails. For example, although Recoder has used the operator mutation for fixing other bugs, it fails to fix incorrect solution (S-O-4) that requires changing the relational operator in "$<$" to "!=" where TBar succeeded. This indicates learning-based APR cannot guarantee a learned pattern is always correctly applied in fixing all programs. *Future APR research on designing fixing operators could work on either (1) incorporating domain-specific knowledge into learning new patterns and (2) improving the generalizability of learned patterns.*

E. How can APR research help language models?

Although our study shows that existing APR techniques can only help to fix a small number of bugs in auto-generated programs by Codex, we believe that APR research can benefit future research in language models in the following aspects:

Test-driven repair framework. Our study adapts the test-driven repair framework [27], [28] that relies on the quality of test cases, and our results show that the public tests (input/output examples in Figure 2) in LeetCode can guide APR tools to generate correct fixes for Codex programs. Specifically, our study shows that we can apply test-driven repair for (1) fixing incorrect solutions generated by the original mode of Codex, and (2) guiding Codex-e by using fault localization information to generate more correct fixes. Language models currently produce a new program from scratch using only natural language instructions. Instead of producing the correct program from scratch, future code generation can first produce an edit of the incorrect program, and further refine it via an iterative test-driven approach.

Prioritization of correct programs. Our study shows that several negative symptoms exist in auto-generated Codex programs. As our study shows that auto-generated programs with these symptoms are unlikely to lead to correct programs, future designers of language models can integrate a filter function into the language model to automatically eliminate programs with negative symptoms. Another alternative solution is to encode these symptoms into the ranking function to guide the language model in selecting better programs. Both of these directions indicate the potential of incorporating recent advancement of APR research in patch correctness assessment [29], [30] and patch prioritization [31], [32] to guide language models like Codex in generating better programs.

Obtaining patch ingredients. Our study in Section V-B shows that we can effectively combine the patch space of TBar and Codex/Codex-e to obtain the required patch ingredients for generating complex fixes. Future research can work on automatically searching and merging the patch ingredients in generating more complex programs/patches. Specifically, we may leverage semantics-based repair approaches to extract code snippets with the same semantic meaning as patch ingredients from the candidate solutions and further stitch these ingredients with incorrect solution to finally produce a correct solution that satisfies the semantic specifications.

F. Balance between control / flexibility for guiding Codex-e.

Section V shows that patches produced by Codex-e rely heavily on the types of provided edit instruction. Compared to Codex-e^{bug} and Codex-e^{stm}, Codex-e^{line} generates the least number of correct fixes. Although the number of fixed solutions by Codex-e^{bug} and Codex-e^{stm} are quite close (15 versus 16 bugs), the fixed defect category varies. Codex-e^{bug} fixes two more multi-hunk bugs, whereas Codex-e^{stm} 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. The flexibility encourages generation of large patches but also may lose precision when fixing bugs that require single-line fixes. In contrast, Codex-e^{stm} is provided with a code context (given by fault localization), which steers the edit to the direction that change the most relevant code context. In another perspective, we can also regard Codex-e^{stm} and Codex-e^{line} as test-based APR tools that fix bugs based on fault localization given by test cases, whereas Codex-e^{bug} generates edits without guidance. Encoding the suspicious code context into the instruction provides more control and performs better at fixing simple bugs, whereas providing general instruction may find more complex and larger edits due to the increased flexibility. In the future, it is worthwhile to study how to construct edit instructions to guide Codex-e in generating more correct fixes.

G. New Usage of Language Models

Automatically generated code from LLMs is gaining traction. The quality of automatically generated code from LLM can be improved by more training data in the form of more open-source code repositories. However, code generated by LLMs can still be untrustworthy. Our work can help to automatically improve the auto-generated code.

Based on our implications, we suggest a practical use case for future software development. We suggest a test-driven development (TDD) workflow where developers specify requirements in natural language and a few test cases. LLMs are responsible for generating a program that may or may not be correct, and semantic program repair approaches [5], [6] may tweak the auto-generated code to increase its possibility of being correct by using the given test cases. For instance, if a developer wants to write a specialized library function, LLMs can generate code for the initial function given the initial method signature and natural language description, while APR can fix small mistakes in the generated function by validating via test cases. This can be a preparation for the time when more of the code is generated by automated tools like Codex.

VII. THREATS TO VALIDITY

External Threats: During the defect categorization, we eliminate the potential bias by first asking two annotators (two authors of the paper) to manually construct and cross-validate the "ground truth patch", if there is any disagreement on patch or defect classification result for a S_{buggy}, they further discuss with the other authors to resolve any unclear categorization (e.g., when multiple fixes exist for a bug) until a consensus is reached. We also release our dataset and classification result for public verification. 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 [2]) exist, our study only evaluates on the Codex language model and the Codex edit mode. Nevertheless, our implications of using APR to fix incorrect solutions from LLMs are still generally applicable because the workflow is orthogonal to any LLMs. 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 Codex and Codex-e use the same dataset for training. Nevertheless, our experiments show that Codex-e is able to generate fixes for many incorrect solutions.

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

Construct Threats: Construct threats may arise when our evaluation metric of the number of correctly fixed solutions/tasks may be too coarse-grained to reflect the effectiveness of APR. Nevertheless, we followed the same metrics used by APR research and recent code generation research.

Conclusion Threats: Conclusion threats include (1) overfitting of our benchmark and (2) subjectivity of ground truth construction. We minimize (1) by constructing a new dataset that does not overlap with the training set of the Codex model. In the experiment, the program generation, and program repair procedure are all automated, while the ground truth construction involves subjective opinions, we minimize it by cross-validating between two annotators.

VIII. Related Work

Automated Program Repair: Automated Program Repair (APR) has gained a lot of attention from both academia and industry in recent years [33]. APR techniques include search-based, semantic-based and learning-based APR. Search-based APR tools [34], [35], [36], [37], [38], [39], [8], [40] (e.g., GenProg [7]) take a buggy program and a correct criteria as inputs, and generate patches in two steps: (1) producing patches using predefined code transformation operators; and (2) searching for a patch over the patch space that satisfies a correctness criteria (e.g. passes given tests). Search-based repair can scale to large programs, but often not to large search spaces. Semantics-based APR techniques (e.g., SemFix [5], Nopol [41], and Angelix [6]) 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.The application of deep learning techniques in program repair has been explored in past few years. DeepRepair [42] and DeepFix [43] are the early attempts to fix bugs by learning fixes from similar code. SequenceR [44] adapts neural machine translation (NMT) to generate patch, whereas CoCoNuT [24] and CURE [25] further improve the results by either encoding program context or using a programming language model. DLFix [45] uses two-layer tree-based RNN to learn code transformations, and Recoder [23] 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 [17] among those DL-based APR tools whose training model is publicly available.

Large Language Model for Code Generation: Large language models such as GPT-3 [46] have shown promising performance in the NLP domain. Hendrycks et al. [3] proposed APPS dataset and evaluated the code generation performance of several variant GPT models with APPS as the fine-tuned data. Later Codex [1], the back-end model that powers GitHub Copilot, Alphacode [2], Codewhisperer [47], and [48] have emerged as language model based automatic code generation platforms. There are emerging approaches combining program synthesis with large language model on fixing API usage [49] and synthesizing regular expression [50], whereas we focus on fixing general errors in code from programming competitions. Nguyen et al. [51] evaluated the quality of code generated by Copilot on a small set of randomly selected LeetCode programming tasks (33 tasks with 132 solutions). Compared to their work, we performed a detailed analysis of 113 programming tasks via the larger dataset LMDefects which we built. The most relevant papers to us are studies on how language model can fix bugs [52], [53]; we evaluated whether APR tools (including and combining Codex-e) can fix programs automatically produced by Codex.

IX. Perspective

In this paper, we study the mistakes made by auto-generated programs from language models like Codex, and investigate whether automated program repair (APR) tools can fix the auto-generated buggy programs. Our study of code generated from language models reveal that: (1) programs produced by Codex share common defect categories as human programmers; (2) existing APR tools (TBar and Recoder) do not perform well at fixing bugs in auto-generated programs (3) given proper instructions such as information from fault localization, Codex edit mode (Codex-e) shows promising results in code edit generation, which outperforms TBar and Recoder. Our study leads us to the following view-points:

- We suggest enhancing language models with software engineering artifacts such as fault location, with the goal of generating higher quality code.
- We suggest directions for automated program repair (APR) research inspired by language models, such as (i) extracting patch ingredients from automatically generated solution set of Codex, and (ii) making fault localization (fix localization) in program repair more flexible (iii) shifting focus from adding more fix patterns to semantic program repair approaches to improve trustworthiness of auto-generated code.

X. acknowledgement

We thank the anonymous reviewers for their suggestions. This work was partially supported by a Singapore Ministry of Education (MoE) Tier 3 grant "Automated Program Repair", MOE-MOET32021-0001, and the National Natural Science Foundation of China (Grant No. 61902170, 62202026, 62141209).

References

[1] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. de Oliveira Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman, A. Ray, R. Puri, G. Krueger, M. Petrov, H. Khlaaf, G. Sastry, P. Mishkin, B. Chan, S. Gray, N. Ryder, M. Pavlov, A. Power, L. Kaiser, M. Bavarian, C. Winter, P. Tillet, F. P. Such, D. Cummings, M. Plappert, F. Chantzis, E. Barnes, A. Herbert-Voss, W. H. Guss, A. Nichol, A. Paino, N. Tezak, J. Tang, I. Babuschkin, S. Balaji, S. Jain, W. Saunders, C. Hesse, A. N. Carr, J. Leike, J. Achiam, V. Misra, E. Morikawa, A. Radford, M. Knight, M. Brundage, M. Murati, K. Mayer, P. Welinder, B. McGrew, D. Amodei, S. McCandlish, I. Sutskever, and W. Zaremba, "Evaluating large language models trained on code," *CoRR*, vol. abs/2107.03374, 2021. [Online]. Available: https://arxiv.org/abs/2107.03374

[2] Y. Li, D. Choi, J. Chung, N. Kushman, J. Schrittwieser, R. Leblond, T. Eccles, J. Keeling, F. Gimeno, A. Dal Lago *et al.*, "Competition-level code generation with alphacode," *Science*, vol. 378, no. 6624, pp. 1092–1097, 2022.

[3] D. Hendrycks, S. Basart, S. Kadavath, M. Mazeika, A. Arora, E. Guo, C. Burns, S. Puranik, H. He, D. Song, and J. Steinhardt, "Measuring coding challenge competence with apps," *NeurIPS*, 2021.

[4] M. Martinez and M. Monperrus, "Astor: A program repair library for java (demo)," in *Proceedings of the 25th International Symposium on Software Testing and Analysis*, ser. ISSTA 2016. New York, NY, USA: ACM, 2016, pp. 441–444. [Online]. Available: http://doi.acm.org/10.1145/2931037.2948705

[5] H. D. T. Nguyen, D. Qi, A. Roychoudhury, and S. Chandra, "Semfix: Program repair via semantic analysis," in *Proceedings of the 2013 International Conference on Software Engineering*, ser. ICSE '13. Piscataway, NJ, USA: IEEE Press, 2013, pp. 772–781. [Online]. Available: http://dl.acm.org/citation.cfm?id=2486788.2486890

[6] S. Mechtaev, J. Yi, and A. Roychoudhury, "Angelix: Scalable multiline program patch synthesis via symbolic analysis," in *Software Engineering (ICSE), 2016 IEEE/ACM 38th International Conference on*. IEEE, 2016, pp. 691–701.

[7] W. Weimer, T. Nguyen, C. L. Goues, and S. Forrest, "Automatically finding patches using genetic programming," in *IEEE/ACM International Conference on Software Engineering (ICSE)*, 2009.

[8] K. Liu, A. Koyuncu, D. Kim, and T. F. Bissyandé, "Tbar: Revisiting template-based automated program repair," in *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2019, pp. 31–42.

[9] "Codex edit mode," 2022. [Online]. Available: https://openai.com/blog/gpt-3-edit-insert

[10] "Github copilot," 2021. [Online]. Available: https://copilot.github.com

[11] "Codex model," 2022. [Online]. Available: https://https://beta.openai.com/playground

[12] "Leetcode discussion forum." [Online]. Available: https://leetcode.com/discuss/

[13] "Leetcode contest," 2022. [Online]. Available: https://leetcode.com/contest

[14] S. H. Tan, J. Yi, S. Mechtaev, A. Roychoudhury *et al.*, "Codeflaws: a programming competition benchmark for evaluating automated program repair tools," in *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*. IEEE, 2017, pp. 180–182.

[15] E. Caballero, . OpenAI, and I. Sutskever, "Description2Code Dataset," 8 2016. [Online]. Available: https://github.com/ethancaballero/description2code

[16] R. Puri, D. S. Kung, G. Janssen, W. Zhang, G. Domeniconi, V. Zolotov, J. Dolby, J. Chen, M. Choudhury, L. Decker *et al.*, "Project codenet: a large-scale ai for code dataset for learning a diversity of coding tasks," *ArXiv. Available at https://arxiv. org/abs*, vol. 2105, 2021.

[17] R. Just, D. Jalali, and M. D. Ernst, "Defects4j: A database of existing faults to enable controlled testing studies for java programs," in *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, 2014, pp. 437–440.

[18] J. Yi, U. Z. Ahmed, A. Karkare, S. H. Tan, and A. Roychoudhury, "A feasibility study of using automated program repair for introductory programming assignments," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, 2017, pp. 740–751.

[19] S. Saha *et al.*, "Harnessing evolution for multi-hunk program repair," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 2019, pp. 13–24.

[20] S. H. Tan, H. Yoshida, M. R. Prasad, and A. Roychoudhury, "Anti-patterns in search-based program repair," in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2016, pp. 727–738.

[21] J. Campos, A. Riboira, A. Perez, and R. Abreu, "Gzoltar: an eclipse plug-in for testing and debugging," in *Proceedings of the 27th IEEE/ACM international conference on automated software engineering*, 2012, pp. 378–381.

[22] R. Abreu, P. Zoeteweij, and A. J. Van Gemund, "On the accuracy of spectrum-based fault localization," in *Testing: Academic and industrial conference practice and research techniques-MUTATION (TAICPART-MUTATION 2007)*. IEEE, 2007, pp. 89–98.

[23] Q. Zhu, Z. Sun, Y.-a. Xiao, W. Zhang, K. Yuan, Y. Xiong, and L. Zhang, "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*, 2021, pp. 341–353.

[24] T. Lutellier, H. V. Pham, L. Pang, Y. Li, M. Wei, and L. Tan, "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*, 2020, pp. 101–114.

[25] N. Jiang, T. Lutellier, and L. Tan, "Cure: Code-aware neural machine translation for automatic program repair," in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 2021, pp. 1161–1173.

[26] E. Jones and J. Steinhardt, "Capturing failures of large language models via human cognitive biases," *arXiv preprint arXiv:2202.12299*, 2022.

[27] Z. Qi, F. Long, S. Achour, and M. Rinard, "An analysis of patch plausibility and correctness for generate-and-validate patch generation systems," in *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, 2015, pp. 24–36.

[28] C. Le Goues, T. Nguyen, S. Forrest, and W. Weimer, "Genprog: A generic method for automatic software repair," *IEEE Trans. Softw. Eng.*, vol. 38, no. 1, pp. 54–72, Jan. 2012. [Online]. Available: http://dx.doi.org/10.1109/TSE.2011.104

[29] S. H. Tan, H. Yoshida, M. R. Prasad, and A. Roychoudhury, "Anti-patterns in search-based program repair," in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 2016, pp. 727–738.

[30] S. Wang, M. Wen, B. Lin, H. Wu, Y. Qin, D. Zou, X. Mao, and H. Jin, "Automated patch correctness assessment: How far are we?" in *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, 2020, pp. 968–980.

[31] H. Ye, J. Gu, M. Martinez, T. Durieux, and M. Monperrus, "Automated classification of overfitting patches with statically extracted code features," *IEEE Transactions on Software Engineering*, 2021.

[32] A. Ghanbari, "Objsim: lightweight automatic patch prioritization via object similarity," in *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2020, pp. 541–544.

[33] C. L. Goues, M. Pradel, and A. Roychoudhury, "Automated program repair," *Communications of the ACM*, vol. 62, pp. 56–65, 2019.

[34] S. H. Tan and A. Roychoudhury, "Relifix: Automated repair of software regressions," in *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ser. ICSE '15. Piscataway, NJ, USA: IEEE Press, 2015, pp. 471–482. [Online]. Available: http://dl.acm.org/citation.cfm?id=2818754.2818813

[35] S. H. Tan, Z. Dong, X. Gao, and A. Roychoudhury, "Repairing crashes in android apps," in *Proceedings of the 40th International Conference on Software Engineering*, ser. ICSE '18. New York, NY, USA: ACM, 2018, pp. 187–198. [Online]. Available: http://doi.acm.org/10.1145/3180155.3180243

[36] M. Wen, J. Chen, R. Wu, D. Hao, and S.-C. Cheung, "Context-aware patch generation for better automated program repair," in *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*. IEEE, 2018, pp. 1–11.

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

[38] S. Mechtaev, X. Gao, S. H. Tan, and A. Roychoudhury, "Test-equivalence analysis for automatic patch generation," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 27, 2018.

[39] Y. Yuan and W. Banzhaf, "Arja: Automated repair of java programs via multi-objective genetic programming," *IEEE Transactions on software engineering*, vol. 46, no. 10, pp. 1040–1067, 2018.

[40] K. Liu, A. Koyuncu, D. Kim, and T. F. Bissyandé, "Avatar: Fixing semantic bugs with fix patterns of static analysis violations," in *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2019, pp. 1–12.

[41] J. Xuan, M. Martinez, F. DeMarco, M. Clement, S. L. Marcote, T. Durieux, D. L. Berre, and M. Monperrus, "Nopol: Automatic repair of conditional statement bugs in java programs," *IEEE Transactions on Software Engineering*, vol. PP, no. 99, pp. 1–1, 2016.

[42] M. White, M. Tufano, C. Vendome, and D. Poshyvanyk, "Deep learning code fragments for code clone detection," in *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2016, pp. 87–98.

[43] R. Gupta, S. Pal, A. Kanade, and S. Shevade, "Deepfix: Fixing common c language errors by deep learning," in *Thirty-First AAAI Conference on Artificial Intelligence*, 2017.

[44] Z. Chen, S. Kommrusch, M. Tufano, L.-N. Pouchet, D. Poshyvanyk, and M. Monperrus, "Sequencer: Sequence-to-sequence learning for end-to-end program repair," *IEEE Transactions on Software Engineering*, vol. 47, no. 9, pp. 1943–1959, 2019.

[45] Y. Li, S. Wang, and T. N. Nguyen, "Dlfix: Context-based code transformation learning for automated program repair," in *Proceedings of*

the ACM/IEEE 42nd International Conference on Software Engineering, 2020, pp. 602–614.

[46] T. Brown, B. Mann, N. Ryder, M. Subbiah, J. D. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell *et al.*, "Language models are few-shot learners," *Advances in neural information processing systems*, vol. 33, pp. 1877–1901, 2020.

[47] "Amazon codewhisperer," 2022. [Online]. Available: https://aws.amazon.com/codewhisperer/

[48] J. Austin, A. Odena, M. Nye, M. Bosma, H. Michalewski, D. Dohan, E. Jiang, C. Cai, M. Terry, Q. Le *et al.*, "Program synthesis with large language models," *arXiv preprint arXiv:2108.07732*, 2021.

[49] N. Jain, S. Vaidyanath, A. Iyer, N. Natarajan, S. Parthasarathy, S. Rajamani, and R. Sharma, "Jigsaw: Large language models meet program synthesis," in *Proceedings of the 44th International Conference on Software Engineering*, 2022, pp. 1219–1231.

[50] K. Rahmani, M. Raza, S. Gulwani, V. Le, D. Morris, A. Radhakrishna, G. Soares, and A. Tiwari, "Multi-modal program inference: A marriage of pre-trained language models and component-based synthesis," *Proc. ACM Program. Lang.*, vol. 5, no. OOPSLA, oct 2021. [Online]. Available: https://doi.org/10.1145/3485535

[51] N. Nguyen and S. Nadi, "An empirical evaluation of github copilot's code suggestions," in *2022 IEEE/ACM 19th International Conference on Mining Software Repositories (MSR)*. IEEE, 2022, pp. 1–5.

[52] H. Pearce, B. Tan, B. Ahmad, R. Karri, and B. Dolan-Gavitt, "Can openai codex and other large language models help us fix security bugs?" *arXiv preprint arXiv:2112.02125*, 2021.

[53] J. A. Prenner, H. Babii, and R. Robbes, "Can openai's codex fix bugs?: An evaluation on quixbugs," in *2022 IEEE/ACM International Workshop on Automated Program Repair (APR)*. IEEE, 2022, pp. 69–75.