CCTest: Testing and Repairing Code Completion Systems

Zongjie Li[a], Chaozheng Wang[b], Zhibo Liu[a], Haoxuan Wang[c], Dong Chen[a], Shuai Wang*[a], Cuiyun Gao[b]
[a] The Hong Kong University of Science and Technology, Hong Kong SAR
[b] Harbin Institute of Technology, Shenzhen, China
[c] Swiss Federal Institute of Technology Lausanne, Switzerland
{zligo,zliudc,dchenbl,shuaiw}@cse.ust.hk, {wangchaozheng}@stu.hit.edu.cn
{gaocuiyun}@hit.edu.cn, {haoxuan.wang}@epfl.ch

***Abstract*—Code completion, a highly valuable topic in the software development domain, has been increasingly promoted for use by recent advances in large language models (LLMs). To date, visible LLM-based code completion frameworks such as GitHub Copilot and GPT are trained using deep learning over vast quantities of unstructured text and open source code. As the paramount component and the cornerstone in daily programming tasks, code completion has largely boosted professionals' efficiency in building real-world software systems.**

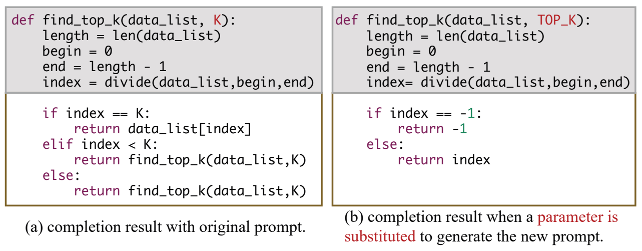
In contrast to this flourishing market, we find that code completion systems often output suspicious results, and to date, an automated testing and enhancement framework for code completion systems is not available. This research proposes CCTest, a framework to test and repair code completion systems in black-box settings. CCTest features a set of novel mutation strategies, namely program structure-consistent (PSC) mutations, to generate mutated code completion inputs. Then, it detects inconsistent outputs, representing possibly erroneous cases, from all the completed code cases. Moreover, CCTest repairs the code completion outputs by selecting the output that mostly reflects the "average" appearance of all output cases, as the final output of the code completion systems. With around 18K test inputs, we detected 33,540 inputs that can trigger erroneous cases (with a true positive rate of 86%) from eight popular LLM-based code completion systems. With repairing, we show that the accuracy of code completion systems is notably increased by 40% and 67% with respect to BLEU score and Levenshtein edit similarity.

I. Introduction

Large language models (LLMs) such as GitHub Copilot [2], OpenAI's Codex [1], Tabnine [9], and Jurassic-1 [5] are increasingly promoted for use within the software development domain. Machine learning (ML) over vast quantities of unstructured text, including websites, books, and open source code, is used to build such models, enabling them to produce "completions" given inputs made up of code and comments (documentation). To date, de facto LLM-based code completion frameworks are advocated with the aim to provide an "AI pair programmer" capable of automatically generating programs from natural language specifications or code snippets.

Despite being a compelling and promising component in augmenting modern software development, we observe that code completion systems are not perfect: they frequently generate confusing and possibly erroneous results. The "sub-optimal" and even buggy behavior of code completion systems

*Corresponding author.



(a)
```
def find_top_k(data_list, K):
    length = len(data_list)
    begin = 0
    end = length - 1
    index = divide(data_list,begin,end)

    if index == K:
        return data_list[index]
    elif index < K:
        return find_top_k(data_list,K)
    else:
        return find_top_k(data_list,K)
```

(b)
```
def find_top_k(data_list, TOP_K):
    length = len(data_list)
    begin = 0
    end = length - 1
    index= divide(data_list,begin,end)

    if index == -1:
        return -1
    else:
        return index
```

(a) completion result with original prompt.
(b) completion result when a parameter is substituted to generate the new prompt.

Fig. 1. Motivating example over Codegen (input and autocompletion is marked with gray and white background, respectively).

are practically undesirable since it undermines the reliability and usability of code completion. Nevertheless, it is yet neglected by today's research community, whereas existing studies on LLM-based code completion frameworks mainly focus on their security implication, cost reduction, or potential extension in different domains [40, 57, 59, 66, 91].

Considering Fig. 1(a), where a popular code completion system, Codegen [53], generates a code snippet as the completion of the input. However, by slightly tweaking the input, we observe that Codegen outputs a dramatically different code snippet, as illustrated in Fig. 1(b). While it is generally obscure to somehow directly decide the "correctness" of these two completed code snippets, given that the two inputs are identical to a human programmer, the high distinction between two completed code snippets indicates that Codegen's outputs are of low consistency, which is a sign of unwanted outputs. Overall, this apparent *inconsistency* in the generated code completions motivates us to test and enhance code completion systems. To do so, we form a testing oracle with the intuition that the completed code snippets should be structure-consistent.

We present CCTest, an automated testing and enhancing framework for code completion systems. Our research thrusts are two-fold. First, we design a set of program mutation strategies, namely program structure-consistent (PSC) mutations, to generate mutated code snippets with similar or identical program structures.[1] Given the corresponding set O of code completion outputs derived from a seed input and its mutated inputs, we identify erroneous cases (i.e., outliers)

[1]To clarify, "structure-consistent" in this paper refers to the process of transforming code snippets (i.e., inputs of code completion systems) into mutated code snippets with identical or little altered program structures.

in O by defining and comparing distances of generated code completions. A code completion output exhibits unusually large distances from other outputs will be deemed spurious. Second, We design a code enhancement strategy over completion outputs by selecting output $\hat{o}$ that is mostly close to the "average" appearance of O. We show that $\hat{o}$ generally manifests higher consistency with the ground truth, extensively improving the accuracy of code completion systems. Furthermore, CCTEST treats code completion systems as a "black box", so we do not assume any specific implementation details of the code completion systems or their underlying LLMs.

CCTEST offers an up-to-date assessment of de facto LLM-based code completion systems and the quality of their outputs, whose defects impede the full potential of modern "AI pair programmer" in software development. From a total of 182,464 test inputs used for this study, we found 33,540 programs exposing code completion errors from eight widely-used LLM-based code completion systems, one of which (Copilot) is a popular commercial tool, and the other seven are either actively developed and maintained by the non-profit organization (CodeParrot [25]) or hosted by the industrial companies (EleutherAI's GPT-Neo [14] and Salesforce's Codegen [53]). With enhancement, we show that the average performance of code completion systems is notably increased by 40.25% and 67.43% with respect to BLEU score and Levenshtein edit similarity. In summary, we make the following contributions:

- We introduce and advocate a new focus on testing and enhancing code completion systems. This is a very important, timely topic, yet no existing testing work has been launched. We thus envision that our efforts can guide future research that aims to use or improve code completion tools.
- CCTEST is an automated testing and enhancement framework for code completion systems even in a black-box, remote API setting. PSC transformations, though relatively simple, are practical and low-cost, and specifically designed to deliver structure-consistent mutations. CCTEST also features a black-box scheme to enhance the quality of completion outputs without the need for retraining.
- We evaluate one commercial (Copilot) and seven popular free code completion systems, in which we successfully found 33,540 programs causing inconsistent code completion outputs. Our enhancement further improves the accuracy of the code completion systems by a large margin. We made various observations and obtained inspiring findings regarding modern code completion systems.

Artifact availability. We have released CCTEST to facilitate further research [12].

II. PRELIMINARY

This section introduces the background of code completion systems to deliver a self-contained paper. Note that the automated testing and enhancement pipeline shipped by CCTEST treats each code completion system as a *black box*, which means we do *not* require prior knowledge of the models or the underlying implementation. Nevertheless, given the dominating usage of LLMs in today's code completion systems [1, 2, 26], we primarily introduce LLM-based code completion.

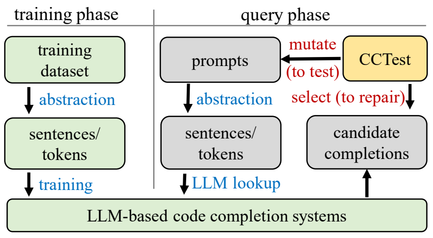


Fig. 2. A holistic view of LLM-based code completion system and how CCTEST facilitates testing and enhancement.

Fig. 2 presents a holistic view of LLM-based code completion systems, and explains how CCTEST fits its workflow for testing and enhancement. Benefiting from the prosperous development and success of Transformer-based natural language models such as OpenAI's GPT2 and GPT3 [15, 19, 42], the code completion task, as a typical conditioned open-ended language generation task, is extensively improved with much higher accuracy and applicability.[2]

Though training data details are often obscure, modern LLMs-based code completion systems are advertised as being trained with millions or even billions of lines of code [2]. Typically, the input training data are dissected into sentences and further into tokens, where each token comprises a sequence of characters before being fed to LLMs. For instance, the tokenizer of Codex and GPT-3 is configured to produce tokens with an average of four characters. Tokens are encoded with unique numeric identifiers, up to user-defined vocabulary size. This process is often referred to as byte pair encoding (BPE) [28], allowing the models to encode any rare words with appropriate subword tokens. Various techniques are proposed to improve the performance of LLMs [85], such as learning from rare tokens and deciding on a proper stop word.

During the query phase, the code completion system's input is often referred to as a *prompt*, denoting an incomplete code snippet. Similarly, the prompt code snippet is first abstracted into sentences and further into tokens, for which the code completion system can predict a list of suggestions (ranked by the confidence scores) that are more likely to continue/complete the input prompt. For instance, the code completion system is frequently assessed by completing a function body, given the function prototypes and some statements in the function prologue. Note that modern code completion systems can process prompts of different types, including the expected program's code snippets and natural language descriptions (comments). Such comments are usually divided into background, input, and output to describe a competitive programming problem [44].

CCTEST is designed to test the code completion system by mutating the seed *prompts* and identifying bug-triggering prompt variants by cross comparing the outputs. Moreover, CCTEST can enhance the code completion output by analyzing outputs of prompt variants to select cases that are closest to the "average" appearance of O.

[2]A full introduction of transformer and its usage in open-ended language generation is beyond the scope of this paper. Audiences can refer to [71].

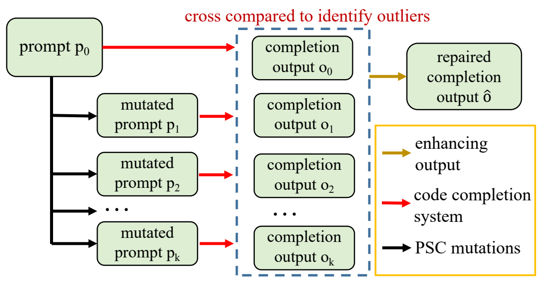


Fig. 3. Workflow of CCTEST. CCTEST launches PSC testing by mutating a prompt into a collection of prompt variants, cross compares code completion results, and enables automated code completion enhancement.

III. Approach Overview

Fig. 3 presents an overview of CCTEST in terms of testing and enhancing code completion systems. In particular,
① **Prompt Variants Generation.** CCTEST launches PSC mutations and generates a set of structure-consistent variants P of an input prompt p_0. Here, we propose a novel approach, PSC, that mutates a prompt with code structure-consistent transformations. The mutated prompts manifest closely consistent structures from human perspective.
② **Testing Oracle Generation.** The target code completion system outputs a set of completion outputs in accordance with the prompt variants. Here, we form a testing oracle over completion outputs. The key observation is that *the code completion outputs should invariably manifest high (visual) consistency for prompt variants produced by using PSC*. Thus, "outlier" completion outputs are deemed erroneous, whose corresponding prompt variants are defect-triggering inputs.
③ **Completion Output Enhancement.** The testing pipeline can be extended to enhance code completion outputs. To this end, CCTEST identifies output $\hat{o}$ that is closest to the "average" appearance of O (with respect to program distance metrics; after excluding outliers in ②). This step does not assume any prior knowledge of the code completion system implementation and, therefore, applies to black-box scenarios.
Study Scope. We primarily target the erroneous code completion outputs, denoting "stealthy logic bugs" of code completion systems. As detailed in Sec. III-B, our testing approach can automatically expose inconsistency defects in code completion systems, meaning that when given a set of structurally consistent prompts, the completion outputs are deemed to share closely similar appearances as well.

During the preliminary study, we also find that certain (mutated) prompts may simply impede code completion systems from generating any outputs. To some extent, this is comparable to identifying a "crash" of the code completion system. Although such obviously anomalous states are not the primary focus of CCTEST, we still record and report all such defects we encountered during the evaluation.

Note that we are *not* using extreme (broken) prompts to stress code completion systems. Modern code completion models are on the basis of LLMs, and our preliminary study shows that providing some trivial code snippets as prompts may make the code completion system generate meaningless outputs. The current implementation of CCTEST focuses on generating *syntactically valid* Python code snippets as testing prompts, given the Python language's popularity and representativeness. Nevertheless, our method is *not* limited to Python; we leave supporting other programming languages as one future work.

A. Program Structure-Consistent (PSC) Mutations

To test code completion systems, our key observation is that programs with identical/mildly-changed control structures will retain such consistency in the completion outputs. Hence, by observing certain completion outputs manifesting high inconsistency, potential code completion errors can be flagged.

TABLE I
TRANSFORMATION SCHEMES IMPLEMENTED IN CCTEST.

Class	Methods	Abbreviations
Identifier Level	rename parameter regulate	REP_R
	rename parameter context	REP_C
	rename local variable regulate	REL_R
	rename local variable context	REL_C
Instruction Level	instruction replacement	IRR
	replace bool expression	RTF
Block Level	garbage code insertion regulate	GRA_R
	garbage code insertion context	GRA_C
	print statement insertion	INI

Design Goal. To generate structure-consistent mutants, we implement a set of PSC transformations, which mutate a seed prompt from different levels of the program hierarchy. Table I lists all the proposed PSC transformations. Each of them is designed to be "lightweight," in the sense that it only slightly changes the prompts and retains the *structure consistency* of mutated prompts. However, we make an encouraging observation that such straightforward, incremental mutations impose notable challenges for code completion systems to process inputs and accordingly generate consistent completion outputs. We now introduce each PSC scheme below.
REP_R & REP_C. Our first two schemes rename function parameters. Overall, they will replace one function parameter with a new identifier, and every usage of this parameter will be identified (by matching identifier names) and replaced accordingly. The naming can either be "regulate" or "context." Considering the following case,

```
// seed prompt
def add(a,b):
    res = a + b
```

```
def add(a,Param1):        // REP_R
    res = a + Param1
def add(a, Add_Param_b): // REP_C
    res = a + Add_Param_b
```

where for the "regular" scheme (REP_R), we replace the function parameter b with Param1. As REP_C, which takes the specific context into account, we extend the function parameter b with a new identifier that subsumes both function name add and b. It is evident that these two methods choose a relatively fixed "pattern" to mutate prompts. However, we clarify that they are designed by taking account developers' programming habits to replace the parameters, which shall help (LLM-based) code completion systems avoid generating irrelevant outputs when they behave correctly. This design intuition applies to following PSC transformations as well, and they are very effective to provoke code completion errors.

REL_R & REL_C. These schemes rename local variables. Similar to renaming parameters, these schemes randomly select a local variable whose scope is in the function. Then, we replace all of its references with a new identifier. Considering the following case,

```
// seed prompt
def compare(a,b):
    res = a > b
```

```
def compare(a,b):    // REL_R
    LocalVar1 = a > b
def compare(a,b):    // REL_C
    compare_res = a > b
```

where for REL_R, we replace the local variable `res` with `LocalVar1`. As for REL_C, which takes the context information into account, we rewrite the local variable `res` with a new identifier that subsumes both function name `compare` and `res`.
IRR. This scheme implements a set of mapping rules to search and replace certain common arithmetic operators with semantic equivalent, yet syntactically different forms. Considering the following case,

```
def addassign(a,b): // seed
    a += b
    return a
```

```
def addassign(a,b): // IRR
    a = a + b
    return a
```

where the += operator is replaced with an addition. We implement four mapping rules over different common arithmetic operators; audiences can refer to our codebase for details [12].
RTF. Besides mutating arithmetic expressions, we also implement RTF to mutate boolean expressions, particularly expressions used in forming branch conditions, with their semantics equivalent variants. Considering the following case,

```
def add (a,b,ignore): // seed
    if ignore:
        return a
```

```
def add (a,b,ignore): // RTF
    if ignore == (b==b):
        return a
```

where the boolean expression is extended with an always-true condition. This would not alter the functionality of the input prompt, nor does it largely change the program structures. However, we find that such mutated boolean expressions can effectively impede code completion systems from generating structure-consistent outputs, as shown in Sec. V.
GRA_R & GRA_C. These schemes insert a small chunk of "garbage code" into the program, which does not alter program semantics, but slightly increases program control flow complexity. We use always-false conditions to form an `if` condition, then insert a small set of statements, which will never be executed, into the "dead" branch. Similar to mutations mentioned above, creating garbage code may take context information into account. Considering the following case,

```
// seed prompt
def add(a, b):
    res = a + b
```

```
def add(a, b):      // GRA_R
    if (False): TempVar = a
    res = a+b
def add(a, b):      // GRA_C
    if (b!=b): Add_TempVar = a
    res = a + b
```

where GRA_R inserts a branch whose `if` condition expression is "False" with a new variable named `TempVar`. As for GRA_C, which considers context (local variables and parameters), we create an always-false condition on the basis of the syntactic form of the function parameters, and use it to form the `if` condition expression. Similarly, variable names in the enclosed branch also subsume both function name and `TempVar`.
INI. The last mutation scheme inserts a `print` statement into the prompt. This scheme is designed based on the observation that programmers often insert such `print` statements to ease debugging. See the following example:

```
// seed prompt
def add(a, b):
    res = a + b
```

```
def add(a, b): // INI
    print(b)
    res = a + b
```

where we insert `print` into the prompt. Though INI only slightly changes a prompt, we find that its induced code completion output effectively improves the output quality of code completion systems; see details in Sec.V-C.
Clarification. While applying most PSC transformations preserve the control-flow structures of a seed prompt, garbage code insertion (GRA_R and GRA_C) schemes slightly alter program structures. Intuitively, code completion outputs for GRA_R- and GRA_C-mutated prompts should appear more distinct than completion outputs induced by other PSC schemes. However, our preliminary study and observation show that more extensive PSC transformations may not inevitably "dominate" outliers in the code completion outputs. In fact, as reported in Sec. V-B, *all* PSC schemes contribute reasonably to uncovering code completion outliers. Thus, we believe designing PSC schemes like garbage code insertion is proper.
Alternative: Mutating Natural Language Comments. Besides processing prompts in the source code form, modern code completion systems can also generate code completions given prompts in natural language sentences. In such cases, the input sentences are usually code comments or descriptions of the intended code functionality. Careful readers may wonder about the feasibility of mutating natural language comments to test code completion models, whose expected workflow may be similar to recent advances in testing machine translation systems [68, 69]. In fact, we tentatively explored this direction. We clarify that unlike machine translation system testing, whose inputs are *arbitrary* natural language sentences, code comments often have *limited mutation space*; for instance, a vast majority of code comments have no adjectives [58, 60]. This distinction makes it hard to mimic testing methods for machine translation.

B. PSC Testing: Forming Testing Oracles

Given a list of code completion outputs $\mathcal{O}$ in accordance with the mutated prompts $\mathcal{P}$, we compare each output $o_i \in \mathcal{O}$ with the remaining cases in $\mathcal{O}$ and decide if it is an "outlier." The complete algorithm is shown in Algorithm 1. We first iterate each case in $\mathcal{O}$, decide its similarity with the remaining cases (lines 2–3), and normalize the scores (line 4). Here, ***Sim*** is implemented using the Levenshtein string-level edit similarity provided by fuzzywuzzy [3], a standard algorithm to decide the edit similarity among two programs. Then, we employ a threshold T to decide whether a case exhibits anomalously low similarity with other cases for more than T times (lines 5–12) and deem the abnormal case as an outlier. The corresponding p_i is deemed as an error-inducing input. Overall, given that we have implemented N ($N = 9$ in the current implementation

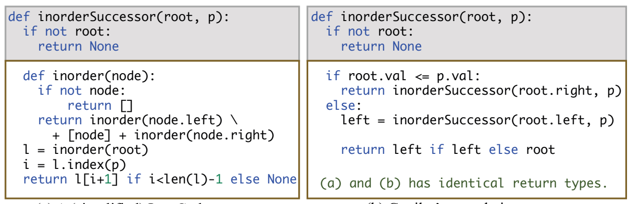
Algorithm 1 Outlier selection algorithm.

```
    Input: $\mathcal{O}$: Code completion output set of size $k$
    Input: $T$: threshold
    Output: $\mathcal{L}$: Outliers
 1: $ScoreMatrix$ = []
 2: for $i$ in 1 to k do
 3:     for $j$ in i to k do $ScoreMatrix[i][j] = \textbf{Sim}(o_i, o_j)$
 4: Normalize($ScoreMatrix$)
 5: for $i$ in 1 to k do
 6:     count = 0
 7:     for $j$ in 1 to k do
 8:         if $ScoreMatrix[i][j] < \textbf{Median}(ScoreMatrix)$ then
 9:             count = count + 1
10:             if count $\geq T$ then
11:                 $\mathcal{L}.append(o_i)$
12:                 break
13: return $\mathcal{L}$
```

of CCTEST) schemes to mutate a prompt, T is a configurable hyperparameter ($T \leq N$), such that a code completion output deems an outlier if its edit distance scores with T code completion outputs are less than the average distance score (computed by ***Median***) of code completion output pairs.

T will be decided with empirical evidence, as will be discussed in Sec. V-B. After performing Algorithm 1, we keep the remaining code completion outputs $\mathcal{O}^* = \mathcal{O} \setminus \mathcal{L}$ for usage in the enhancement phase (Sec. III-C).

Alternative: *Sim* Metrics. One may question the current implementation of ***Sim***, which is based on string-level edit distance rather than structure-level distance. During preliminary study, we explored using structure-level distance metrics to implement ***Sim***. Nevertheless, speed is a main bottleneck, as tree-edit distance is much slower. Moreover, code completion outputs are usually code snippets consisting of approximately ten (or fewer) lines of code whose "structures" may be less significant; we find that standard tree-edit distance metrics often give vague results. Thus, we deem that using Levenshtein string-level edit distance proper and practicable for our testing.



```
def inorderSuccessor(root, p):
  if not root:
    return None

  def inorder(node):
    if not node:
      return []
    return inorder(node.left) \
      + [node] + inorder(node.right)
  l = inorder(root)
  i = l.index(p)
  return l[i+1] if i<len(l)-1 else None
```

(a) A (simplified) LeetCode program.

```
def inorderSuccessor(root, p):
  if not root:
    return None

  if root.val <= p.val:
    return inorderSuccessor(root.right, p)
  else:
    left = inorderSuccessor(root.left, p)

    return left if left else root

  (a) and (b) has identical return types.
```

(b) Copilot's completion output.

Fig. 4. Comparing the syntactic form with ground truth appears overly strict and lead to false positives. The code snippets are simplified for readability.

Alternative Testing Oracles. We deem our formed structure-consistency testing oracle as an instance of metamorphic testing oracle [21]. One may wonder about the feasibility of forming alternative testing oracles. Below, we discuss two alternative oracles and deliberate their (in-)capability in our research.

Comparing Syntactic Form with Ground Truth. One way of forming an oracle is to cut a program p into two chunks p_1 and p_2, whereas p_1 serves the prompt, and p_2 serves the ground truth. We test code completion by comparing the syntactic-level equality between o_1 (completion output of p_1) with p_2.

We clarify that this oracle is often *too strict*. Fig. 4 compares the code completion output of Copilot and the ground truth code snippet (this code snippet is from LeetCode [7]). While they have identical semantics, their syntactic forms are distinct. The ground truth uses a helper function `inorder` to deliver a classic inorder traversal of the input binary tree, whereas Copilot output appears to be leveraging recursive calls. Overall, though the code completion output may frequently appear syntactically distinct from ground truths, the completion outputs should be deemed as functionally correct. Therefore, comparing the syntactic equality of completion outputs with ground truth may not faithfully reflect true defects of code completion systems. Such a strict oracle may induce many false positives.

Checking Functionality According to Ground Truth. Recent works explored strictly checking the functionality correctness of code completion outputs in accordance with ground truth. [52] tests Copilot using prompts created from LeetCode programs, then counts the number of passed LeetCode test cases over the code completion outputs (each completion output forms a LeetCode solution, when put together with the prompts). Similarly, HumanEval [18] assesses the functionality correctness of code completion outputs with hand-written programming problems. Overall, for those approaches, more passed test cases indicate that the completion outputs are of higher quality. Those approaches, however, may be less desirable than our formed oracle. Modern code completion systems are *not* championed to replace human programmers; rather, it aids human programmers by suggesting useful code snippets. Therefore, strictly correct functionality may not be the first design target for a code completion system, e.g., some sloppy arithmetic errors in completion outputs may be easily fixed by users. Thus, we believe it is more desirable to cross compare the structural-level consistency (as in CCTEST), rather than using test inputs to check the functionality correctness.

C. *Completion Output Enhancement*

This section presents the technical pipeline of enhancing the code completion outputs. By launching the testing in Sec. III-B, we collected a set of code completion outputs $\mathcal{O}^*$, with outliers being excluded. After that, we aim to identify a code completion output that appears mostly close to the "average appearance" of O^*. To this end, we measure the average pair-wise edit similarity $\hat{s}$ of every $o_i, o_j \in O^*$. Note that at this step, we re-use the ***Sim*** function implemented in Alg. 1, meaning that we compute the Levenshtein string-level edit distance as the similarity score. Then, we search for an $\hat{o} \in O^*$, whose average pair-wise similarity scores with all other elements in O^* is the closest to $\hat{s}$. This $\hat{o}$ will be the repaired code completion output returned to the users.

To clarify, CCTEST aims to enhance the quality of code completion outputs, in the sense that the repaired output is closer to the "average-looking" output. As in our evaluation (**RQ3**; see Sec. V-C), we use edit similarity and BLEU scores as metrics to unveil that CCTEST can find better completions, by showing that the repaired outputs have closer distance with the ground truth under both metrics. Nevertheless, CCTEST cannot guarantee that the repaired outputs become "semantically correct", if the original outputs are not. In other words, CCTEST does *not* directly repair the semantics-level defects/inconsistency. Semantics-level code repairing is inherently hard. More importantly, it is unaligned with the design goal of CCTEST. As already noted in **Alternative Oracle** above, we believe it is less proper to check the functionality correctness in this research context.

IV. Implementation and Evaluation Setup

CCTEST is implemented in Python, with about 5k LOC. CCTEST currently focuses on mutating Python code, given the popularity of this language in software development and the matureness of corresponding code completion systems. We now discuss the implementation and evaluation setup.

Parsing and Mutating Programs. We parse Python code with tree-sitter [16], a mature and dependency-free parser generator tool with an open-source license. It is widely used in code-related projects such as Codebert [26], and it does not require the input code to be executable, meaning that incomplete code fragments without a building script can also be parsed. We first parse the prompt code into a concrete syntax tree, and conduct several sanity checks (see in Sec. V-A) on the target code to see which PSC transformation could be performed. Then, after applying feasible PSC transformations, CCTEST will output the corresponding transformed prompts with IDs of the applied PSC transformations.

TABLE II
Statistics of the Python prompts used in the study.

Split	LeetCode	CodeSearchNet
Total # seed programs	613	2,297
Average # token per seed	149.4	115.4
Max # token among seeds	513	623
Total # generated variants	4,296	15,602
Average # token per mutated seed	151.6	117.3
Max # token among mutated seeds	525	629

Seed Programs. Consistent with most research in this field, we form our evaluation dataset from a popular repo of LeetCode solution programs [6] and CodesearchNet [39]. LeetCode is an online platform to practice algorithmic coding challenges for technical interviews. As will be noted in Sec. III, since LLM-based code completion systems limit the max token size, we only select programs whose token length is between 32 and 2048. Overall, each seed program contains a medium size function with generally complex control structures. As in Table II, we pick 613 code snippets from the LeetCode solution repo as the seed programs. The total number of generated variants for LeetCode programs is 4,296 (recall not every PSC transformation is applicable to arbitrary Python code).

CodeSearchNet is particularly designed in the research of code representation learning. The Python component of this dataset contains train, validation, and test splits; we use the test split for the evaluation. Similar to LeetCode, we only select programs with token length between 32 and 2048, and for the code snippets which share the same attribute named "path", we only maintain one to keep the result balanced.

We clarify that both LeetCode and CodeSearchNet are deemed proper for neural code learning tasks like code completion [39] and they are extensively used in benchmarking relevant models [26, 33, 34, 81]. Thus, errors and enhancement demonstrated in our evaluation should mostly reflect common obstacles and improvement users can expect in real-life scenarios. In contrast, *overly complex prompts* (e.g., real-world complex software) may unavoidably impede the understanding and assessment of LLM-based code completion systems.

TABLE III
Code completion systems evaluated in the study.

System Name	# Params	# Vocab (tokens)	# Max. tokens
Copilot	?	?	?
CodeParrot-small	110M	32,768	1024
CodeParrot	1.5B	32,768	1024
GPT-Neo-125M	125M	50,257	2048
GPT-Neo-13B	1.3B	50,257	2048
GPT-J	6B	50,400	2048
Codegen-2B	2B	50,400	2048
Codegen-6B	6B	50,400	2048

Statistics of Test Cases. Table II reports statistics of the test cases. We collect a total of 2,910 programs from LeetCode/-CodeSearchNet as test seeds for each code completion system. The total number of generated variants is 19,898 (see Table VII for the breakdown). For each seed program with its variants, we equally split the function into two parts: the first part is used as a prompt, and the remaining is used as "ground truth" to assess our enhancement, see details in Sec. V-C.

Code Completion Systems. Table III reports the code completion systems used in the evaluation. First, we use Github Copilot, one highly visible commercial code completion system that generates quite a buzz in the community [82]. We purchase the standard commercial license (for single user) to unleash its full potential. As a "black-box" commercial product, it is unclear about their implementation (marked as ? in Table III).

We also evaluate several well-known LLM models for code completion, including CodeParrot [25], GPT-Neo [14], GPT-J [24], and CodeGen [53]. All of these models are deemed to employ large-scale language models, given that up to billions of parameters are involved in their underlying models, and tens of thousands of vocabularies are considered. CodeParrot is a GPT-2 model trained specifically for Python code generation. We use two variants, CodeParrot-small and CodeParrot. The smaller variant contains less amount of parameters and is trained over fewer data. Nevertheless, both variants manifest a high level of code generation capability. As for GPT-Neo, we use two variants, GPT-Neo-125M and GPT-Neo-13B, which are both GPT3-like models. Our observation shows that GPT-Neo-125M is prone to generate less diverse but more straightforward code snippets compared to its larger variant. Given that said, we believe both models are well-suited for code generation and manifest reasonably high robustness under our testing campaign. GPT-J, also referred to as GPT-J-6B, is a transformer model with 6B parameters in the pre-trained model. Two versions of CodeGen specially designed for program synthesis are evaluated. Both versions, CodeGen-2B-mono and CodeGen-6B-mono, are trained on a large corpus of Python code.

All these systems are stated to be trained on a large corpus of open-source source code. For instance, Copilot is noted to include the vast majority of GitHub's open-source code. GPT-J and GPT-Neo are trained on the Pile dataset [30], a diverse language modeling dataset. We download their pre-trained models from huggingface [4] and run the code completion locally for all LLMs except Copilot. LLMs support different search (sampling) strategies to generate code completions, such as beam search, temperature sampling. These factors make our testing pipeline less deterministic. Thus, in our experiments, we follow Copilot's interface to only choose the completion result with the highest confidence score as its "output." For other tested models, we disable their sampling strategy.

In all, LLM-based code completion systems presented in Table III, to the best of our knowledge and experience, represent the best systems available to the public. We tentatively explored code completion solutions based on conventional machine learning or rule-based approaches. We clarify that these conventional methods were seen to produce much worse and shallow code completion outputs compared with these LLM-based modern systems.

V. Findings

In evaluation, we mainly explore the following research questions. **RQ1**: Can CCTest generate high-quality and structure-consistent prompt variants? **RQ2**: How effective is CCTest in detecting code completion defects? **RQ3**: To what extent can CCTest enhance the quality of code completion outputs? We answer these RQs in subsections below.

A. *RQ1: Effectiveness on Input Generation*

Answering **RQ1** requires assessing the quality of mutated prompts. At this step, for each seed prompt, we generate mutants and first check whether they pass the parsing. In particular, we generate up to 9 mutants for each prompt, leading to a total of 19,898 mutant prompts generated on top of 2,910 seed prompts. We use the standard `ast` module in Python to parse all transformed Python prompts into abstract syntax trees to check their validity. All generated mutant prompts are parsed without any error, indicating that they are grammatically valid.

TABLE IV
Distribution of structural consistency scores.

Distance	$[0, 0.05]$	$[0.05, 0.1]$	$[0.1, 0.15]$	$[0.15, 0.2]$	$[0.2, 0.9]$	$[0.9, 1.0]$
Freq. (%)	90.16	8.30	0.92	0.30	0.32	0
Cumulative Freq. (%)	90.16	98.46	99.38	99.68	100.0	100.0

To illustrate the structure consistency of mutated prompts, we use `pycode-similar` [8], a well-performing tool to calculate Python code similarity. This similarity metric is based on AST structures, which is different from Levenshtein edit similarity that we use in deciding the "outliers". Let the prompt AST have n nodes, where m nodes are matched toward nodes on the mutated prompt's AST. `pycode-similar` returns ratio $\frac{m}{n}$, denoting how similar two programs are. We report the distribution of the "distance score" (distance is computed as $1 - \frac{m}{n}$) in Table IV. For the vast majority of mutated prompts (over 98%), structural-level distance is less than 0.1, meaning over 90% of AST nodes are matched. Recall that several PSC transformations (Sec. III-A) introduce only identifier-level changes. As expected, the distance scores between their mutated and seed prompts are zero, implying that program structures are retained. A few cases have slightly higher distances. With manual inspection, we find that it is primarily because the size of original prompts is short, leading to an increased distance $1 - \frac{m}{n}$ where n is small.

Answer to RQ1: CCTest can generate large-scale, grammatically valid, and structurally consistent prompt mutants.

B. *RQ2: Bug Detection*

To answer **RQ2**, we first launch testing to detect code completion defects. Table V reports our findings. Note that we use two datasets for testing, including 4,909 prompts from the LeetCode dataset and 17,899 prompts from the CodeSearchNet dataset. In Table V, "3003 + 12101" in the first "#Outliers $T = 1$" cell means that 3,003 outliers are found using the LeetCode mutated inputs, whereas 12,101 outliers are found using the CodeSearchNet mutated inputs. The same format applies for the "#No Results" column.

First, as shown in the second column of Table V, while nearly all mutated prompts can be processed, we still find 58 (0.03%) mutated prompts that trigger "no response" for the tested code completion systems. As expected, such cases are rare, given the high capability and comprehensiveness of LLM-based production code completion systems.

For the outlier detection evaluation, we assess the performance under different values of the hyper-parameter T. As clarified in Sec. III-B, CCTest leverages a threshold T to form the testing oracle: a code completion output is deemed an outlier if it has a small similarity score with T code completion outputs mutated from the same seed prompt. Therefore, T is an integer ranging from 1 to the total number of schemes (9 in our implementation). Table V reports the number of uncovered outliers in accordance with different models and thresholds. Overall, out of 182,464 test cases, a substantial number of defects are found under all five configurations. As expected, the increment of T decreases the number of detected defects (outliers). As we illustrated in Algorithm 1, the selection stringency for outliers depends on the threshold T, where a higher T represents a stricter standard. Also, different from some robustness testing papers (e.g., [31]) that merely detecting mutated inputs that lead to outlier predictions, bad completions are derived from either original (what users start with) or mutated prompts in this study. For instance, when $T = 9$, among all 33,540 (5912 + 27628) bad completions, 3,008 (8.9%) are outputs derived from the original prompts. That is, a reasonable portion of "bad completions" are from original prompts.

Copilot outperforms the other seven LLMs, given fewer inconsistency defects (293+2184 for $T = 9$). However, Copilot has the most "no response" failures (41 out of 58). We suspect that Copilot will refuse to return any output when its model in the remote server fails to generate code snippets with valid syntax or high confidence scores. Despite the small number of "no response" failures, all the other cases can be processed by code completion systems to produce non-trivial outputs.

TABLE V
OVERVIEW OF OUTLIER DETECTION RESULTS. FOR EACH SYSTEM, WE USE 4909 LEETCODE PROMPTS AND 17899 CODESEARCHNET PROMPTS TO TEST.

System	**#No Results**	**#Outliers**				
		T= 1	T= 3	T= 5	T= 7	T= 9
Copilot	4+37	3003 + 12101	1347 + 7928	803 + 5570	559 + 3899	293 + 2184
CodeParrot	1+0	4798 + 17605	4379 + 16118	3631 + 13368	2359 + 9023	904 + 3778
CodeParrot-small	2+0	4812 + 17611	4469 + 16069	3776 + 13454	2470 + 9344	1033 + 4009
GPT-J	0+0	4606 + 17280	3776 + 15073	2832 + 12005	1786 + 7875	586 + 3174
GPT-NEO-13B	1+0	4729 + 17427	4088 + 15495	3311 + 12536	2120 + 8462	794 + 3463
GPT-NEO-125M	0+2	4734 + 17509	4281 + 15556	3654 + 12907	2523 + 8966	1079 + 3700
Codegen-2B-mono	2+9	4661 + 17221	3794 + 15112	2731 + 12241	1638 + 8460	639 + 3761
Codegen-6B-mono	0+0	4578 + 17016	3575 + 14595	2493 + 11546	1452 + 7866	584 + 3559
Total	10+48	35921 + 133770	29709 + 115946	23231 + 93627	14907 + 63895	5912 + 27628

TABLE VI
ASSESSING CCTEST'S FINDINGS WITH MANUAL INVESTIGATION.

	T=1	T=3	T=5	T=7	T=9
TP	216	342	429	537	689
FN	26	38	70	90	104
Precision	0.270	0.427	0.536	0.671	0.861
Recall	0.892	0.900	0.859	0.856	0.868
F1 score	0.414	0.579	0.660	0.752	0.865

Manual Validation. At this step, we first measure the true positive (TP) rate of the outliers found under different thresholds T. For each pair of $\langle T, model\rangle$, we randomly sample 100 cases from two datasets, resulting in a total of 4,000 ($5 \times 8 \times 100$) cases. The first two authors check each case to manually decide if an outlier is TP or false positive (FP). Our manual inspection results are presented in the "TP" row of Table VI. It shows that with the increment number of T, the number of TPs keeps increasing. Particularly, when $T = 9$, out of 800 positive findings of CCTEST, we have only 111 ($800 - 689$) cases that are FPs. To better understand our findings, we further analyze those remaining 111 cases and summarize the following two main reasons for FPs.

(i) The completion results are highly robust for the majority of mutants. To select an outlier, Algorithm 1 will cross compare the outputs from the same seed program. However, it would be possible that all other results are exactly the same except for one mutant, which would be treated as an "outlier", even if it is only slightly different from the others. Such cases count for the 59.46% of FPs. (ii) No "mainstream" results. The completion results for some vague prompts would vary drastically, and sometimes the vast majority, if not all completion outputs, appear to be different. During the manual inspection of such cases, authors would not deem any mutants as "outliers." Such cases count for 26.13% of FPs.

Recall that our test oracle uses string-level distance metrics to decide outliers, instead of directly using program structure-level distance metrics. We have clarified why string-level distance is preferable and practical to structure-level distance in **Alternative *Sim* Metrics** paragraph of Sec. III-A. From the above manual inspection, it is evident that using string-level distance metrics does not primarily add FPs; we find that when two completion outputs exhibit a high distance in string-level metrics, they generally look distinct from program structures as well. Overall, empirical results here further support using string-level distance metrics in deciding outliers.

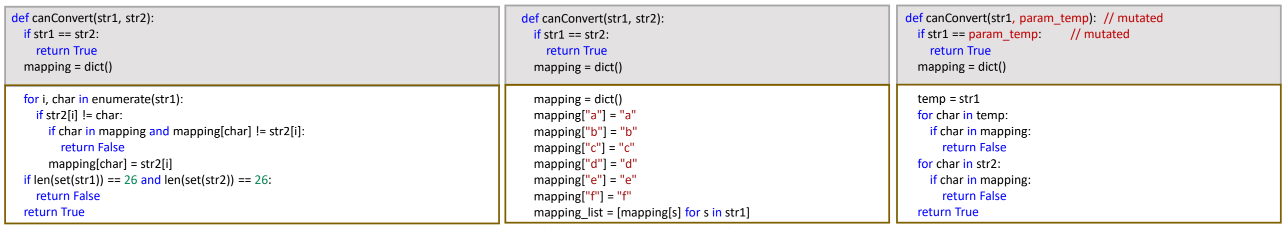
The above study explores TP/FP over 4,000 "positive" findings. As a testing tool, CCTEST cannot avoid false negatives (FNs). Nevertheless, measuring FNs help understand the potential of CCTEST. Hence, at this step, we randomly select 4,000 negative samples from CCTEST's findings. Two authors manually check the FN and true negative (TN) cases (the procedure follows how we confirm TP/FP). We report the number of FNs in Table VI. We accordingly compute the precision, recall, and F1 scores. It is observed from the "FN" row of Table VI that the number of FNs increases when T grows. This is reasonable, given that the higher T is, the more likely CCTEST may miss some findings. Nevertheless, CCTEST becomes more accurate when T grows, as reflected by the F1 scores in Table VI. In short, when $T = 9$, CCTEST achieves reasonably high accuracy. We interpret the results as overall encouraging and reasonable, and we recommend using $T = 9$ as the default setup in practice.

TABLE VII
DISTRIBUTION OF SCHEMES WHICH TRIGGER OUTLIERS WHEN $T = 9$.
LC REPRESENTS LEETCODE AND CNS REPRESENTS CODESEARCHNET.

Pass name	IRR	GRA_R	GRA_C	REP_R	REP_C	INI	RTF	REL_R	REL_C
#variants on LC	253	598	594	601	600	601	38	505	506
#variants on CNS	153	2297	2297	2207	2207	2295	314	1914	1918
outliers on LC (%)	3.51	17.03	17.57	11.35	8.38	12.7	0.54	17.03	11.89
outliers on CNS (%)	0.27	15.41	18.11	11.62	14.59	11.62	1.62	14.32	12.43

Potency of PSC Transformations. We measure the potency of all nine PSC transformations about their contributions to uncovering outliers. For both LeetCode and CodeSearchNet, we report the distribution of generated variants and triggered outliers in Table VII. Due to limited space, we report the average distribution across all code completion systems; see [12] for each system. Overall, all PSC transformations reasonably facilitate uncovering outliers, even though they are performed toward different levels of program hierarchical representation (e.g., identifier level or basic block level). In particular, nearly all schemes, except "IRR" and "RTF", have comparable contributions. IRR has a relatively minor application scope, given that IRR looks for specific arithmetic operations like "+=" that may not be pervasively used. Similarly, RTF requires the existence of an `if` branch with a relatively simple condition, which may not be available in our test cases. As a consequence, these two schemes result in less amount of uncovered outliers, as shown in the last two rows in TableVII.

Case Studies. Table V illustrates that the inconsistency bugs in code completion systems are pervasive, and numerous defects can be found even for highly permissive consistency thresholds. We present two representative cases in Fig. 5 and Fig. 6. Fig. 5 presents a case such that the completion output of the seed prompt largely deviates from the ground truth. When only given



```
def canConvert(str1, str2):
  if str1 == str2:
    return True
  mapping = dict()

  for i, char in enumerate(str1):
    if str2[i] != char:
      if char in mapping and mapping[char] != str2[i]:
        return False
      mapping[char] = str2[i]
  if len(set(str1)) == 26 and len(set(str2)) == 26:
    return False
  return True
```

(a) LeetCode 1153 code completion result.

```
def canConvert(str1, str2):
  if str1 == str2:
    return True
  mapping = dict()

  mapping = dict()
  mapping["a"] = "a"
  mapping["b"] = "b"
  mapping["c"] = "c"
  mapping["d"] = "d"
  mapping["e"] = "e"
  mapping["f"] = "f"
  mapping_list = [mapping[s] for s in str1]
```

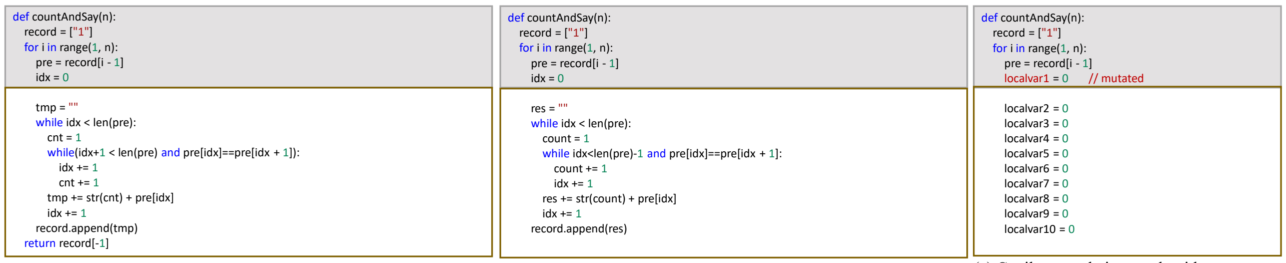
(b) GPT-J completion result with original prompt, detected as an outlier.

```
def canConvert(str1, param_temp):  // mutated
  if str1 == param_temp:       // mutated
    return True
  mapping = dict()

  temp = str1
  for char in temp:
    if char in mapping:
      return False
  for char in str2:
    if char in mapping:
      return False
  return True
```

(c) GPT-J completion result with prompt transformed by REP_R.

Fig. 5. Outlier case study. The code completion output of the seed prompt is largely deviated, whereas a mutated prompt results in much better code completion output. The "completion result" (code in white) in (a) means ground truth.



```
def countAndSay(n):
  record = ["1"]
  for i in range(1, n):
    pre = record[i - 1]
    idx = 0

    tmp = ""
    while idx < len(pre):
      cnt = 1
      while(idx+1 < len(pre) and pre[idx]==pre[idx + 1]):
        idx += 1
        cnt += 1
      tmp += str(cnt) + pre[idx]
      idx += 1
    record.append(tmp)
  return record[-1]
```

(a) LeetCode 38 completion result.

```
def countAndSay(n):
  record = ["1"]
  for i in range(1, n):
    pre = record[i - 1]
    idx = 0

    res = ""
    while idx < len(pre):
      count = 1
      while idx<len(pre)-1 and pre[idx]==pre[idx + 1]:
        count += 1
        idx += 1
      res += str(count) + pre[idx]
      idx += 1
    record.append(res)
```

(b) Copilot completion result over the seed prompt.

```
def countAndSay(n):
  record = ["1"]
  for i in range(1, n):
    pre = record[i - 1]
    localvar1 = 0      // mutated

    localvar2 = 0
    localvar3 = 0
    localvar4 = 0
    localvar5 = 0
    localvar6 = 0
    localvar7 = 0
    localvar8 = 0
    localvar9 = 0
    localvar10 = 0
```

(c) Copilot completion result with prompt transformed by REL_R, detected as an outlier.

Fig. 6. Outlier case study. The code completion output of a mutated prompt is largely deviated, whereas a the seed prompt results in much better code completion output. The "completion result" (code in white) in (a) means ground truth.

the completion output in Fig. 6(b), it is inherently challenging to decide if the deviation between completion outputs in Fig. 6(a) and Fig. 6(b) is due to model capacity or bugs. Nevertheless, when referring to Fig. 6(c), it becomes evident that the tested code completion system, GPT-J, is *capable* of generating high-quality completion outputs with closer structure consistency to the ground truth. We only tweak a little on the parameter name for the prompt in Fig. 6(c) compared to the seed. Thus, it should be accurate to consider Fig. 6(b) as code completion bug, which may be likely repairable.

Fig. 7(a) presents another case, such that the code completion output of the seed prompt appears to be highly similar to the ground truth in Fig. 7(b). In contrast, when applying the REL_R scheme to mutate a local variable's name, the code completion output of a mutated prompt (as in Fig. 7(c)) becomes largely distinct from the ground truth, which clearly denotes a bug in the code completion system.

Time & Cost. We employ a GPU server for running the involved models locally. The server has an Intel Xeon Platinum 8276 CPU, 256 GB memory, and 4 NVIDIA A100 GPUs. Although processing time is generally not a concern, we record and report that it takes 35 GPU seconds to finish completing one prompt on average. For each seed, CCTEST takes 2.7 CPU seconds to generate nine mutated prompts and about 82 GPU seconds for code completion systems to infer and obtain the results in parallel. The outlier detection and enhancement (discussed in **RQ3**) steps take about 1.1 CPU seconds. Overall, we admit that generating prediction from many mutated prompts and "averaging" them will take time, compromising interactive response time. Nevertheless, **RQ3** illustrates that code completion outputs are of much higher quality. Moreover, for commercial models, Table V has revealed that roughly 10.86% percent of its code completion outputs are spurious even when $T = 9$. In other words, we interpret that about 10.86% percent of the code completion outputs are highly confusing to the users and are thus "wasted." This may indicate an undesirable situation and potential *financial loss*, given that modern cloud-based code completion systems may feature a "pay-as-you-go" mode, where users are charged based on how many queries they send to the services.

Answer to RQ2: CCTEST identifies numerous defects when being used to test (commercial) code completion systems, despite the varying thresholds used in deciding outliers. We recommend configuring $T = 9$ as a presumably proper threshold (with the highest TP rates) in usage.

C. RQ3: Enhancement Effectiveness

To answer **RQ3**, we first present the improved accuracy of code completion systems. Then, we assess the potency of each PSC scheme about their contributions to enhancement. Lastly, we conduct a human evaluation to verify the effectiveness of CCTEST in enhancing code completion tools.

Improved Accuracy. We report the improved accuracy with respect to both Levenshtein edit similarity [3] and BLEU score [56] in Table VIII when $T = 9$. Note that both metrics are commonly used in relevant research to assess the performance of LLMs [33, 34]; a higher edit similarity score or BLEU score between the "ground truth" and the completion output indicates better performance. To clarify, Table VIII reports the enhancement ratio. For instance, when assessing Copilot against the LeetCode dataset, let the edit similarity or BLEU score be

TABLE VIII
ENHANCEMENT RESULT WHEN $T = 9$. VALUES DENOTE IMPROVEMENT RATIOS AGAINST BASELINE, INSTEAD OF ABSOLUTE IMPROVEMENT AMOUNTS.

	Code Completion Systems								Avg. Enhancement
	Copilot	CodeParrot	CodeParrot-small	GPT-J	GPT-NEO-13B	GPT-NEO-125M	Codegen-2B-mono	Codegen-6B-mono	
BLEU (%)	21.22 + 33.98	59.96 + 50.21	38.31 + 42.85	36.14 + 46.9	40.58 + 50.04	52.74 + 54.71	39.81 + 22.96	32.84 + 21.02	40.20 + 40.33
Edit Sim (%)	4.77 + 38.77	56.74 + 64.45	53.41 + 65.74	87.93 + 80.16	102.3 + 82.8	100.69 + 110.84	52.81 + 72.93	37.48 + 67.17	62.01 + 72.85

s. We compute the enhancement ratio as $r = \frac{s'-s}{s}$, where s' is the edit similarity/BLEU score after enhancement. Similar to **RQ2**, "21.22 + 33.98" in the first Copilot cell means that the relative gain of average BLEU score on LeetCode dataset is 21.22% and on CodeSearchNet dataset is 33.98%.

Cost. As noted in **Time & Cost** paragraph in Sec. V-B, processing all prompts mutated from a seed takes about 82 GPU seconds on average, while the cost of outlier detection and enhancement selection is about 1.1 CPU seconds. Therefore, we estimate that an acceptable cost is incurred for a substantial improvement. Note, however, that commercial code completion systems may have a "pay-per-query" mode, in which each query is charged. Therefore, it may be preferable for the service provider (instead of users) to deploy CCTEST.

Potency. Let completion output o_i be the output used for enhancement, and o_i is generated using the prompt mutated from PSC transformation t_i. We deem t_i under this circumstance as the "optimal" transformation that contributes to the code completion enhancement. For both LeetCode and CodeSearchNet, we report the distribution of different transformations selected as "optimal" in TableIX. We find all PSC transformations are effectively served as "optimal" in both datasets. As clarified in the "potency" evaluation in **RQ2**, "IRR" and "RTF" have smaller application scope and fewer variants, which indicates their relatively lower proportion of being "optimal" on both datasets. However, an exception is that "IRR" contributes disproportionately well when using the LeetCode dataset. With manual inspection, we find that the programs in LeetCode are more likely to contain arithmetic operations such as "+=" compared to programs in CodeSearchNet. In general, we interpret the evaluation results are highly encouraging, showing that almost all the designed schemes manifest high applicability and effectiveness in enhancing different models.

TABLE IX
DISTRIBUTION OF "OPTIMAL" PSC TRANSFORMATIONS CONTRIBUTING TO THE OUTPUT ENHANCEMENT WHEN T IS 9.
LC REPRESENTS LEETCODE AND CNS REPRESENTS CODESEARCHNET

Pass name	IRR	GRA_R	GRA_C	REP_R	REP_C	INI	RTF	REL_R	REL_C	Total
BLEU on LC (%)	7.12	2.65	3.14	2.79	4.78	10.80	0.27	4.76	3.89	40.20
BLEU on CNS (%)	0.21	4.21	3.47	4.59	5.88	5.89	0.59	5.57	9.92	40.33
Edit Sim on LC (%)	5.32	4.81	3.60	9.03	8.13	12.59	1.37	8.46	8.70	62.01
Edit Sim on CNS (%)	0.20	5.89	6.15	8.66	8.27	13.40	1.38	14.12	14.77	72.85

Human Study. We further conduct human evaluation to inspect the quality of the enhanced completion outputs. We randomly select 60 samples and create an online questionnaire for them. We invite twelve experts, including four industrial developers and eight academy researchers with expertise in LLMs, as the participants. We provide two completion outputs for each seed program without specifying whether they are the original or the CCTEST's enhanced outputs. On a scale from 1 to 5, the participants are asked to provide a score (1 for a completely not satisfying output and 5 for a fully satisfying output).

To ensure that all participants correctly understood the meaning of the questionnaire, we prepare and insert five sanity-check (SC) test items randomly into the questionnaire, and only participants who answer all SC correctly are considered legitimate. We evenly assigned 30 real samples with five SC to each participant, and ensured that each selected sample was examined by six participants. All participants passed the sanity check and the average completion time was 45 minutes.

The human evaluation results show that the average score for the original outputs is 2.3188, whereas for CCTEST's enhanced outputs, the average score is 3.1565, representing a relative gain of 36.12%. We analyzed all their answers. Respondents believe that for 2.2% of the cases, the enhanced completion outputs look worse than the original outputs, while the remaining 97.8% treat the enhanced outputs equal to or better than the original ones. The Fleiss' Kappa score [27] for the questionnaire is 0.94, which can be interpreted as "almost perfect agreement" between human participants. These findings indicate CCTEST's effectiveness in enhancing code completion outputs.

Answer to RQ3: CCTEST improve the outputs of different code completion systems. We also find that instead of one or a few PSC transformations that significantly enhance code completion, all schemes are shown as effective.

VI. DISCUSSION

Limitations and Threats to Validity. We now discuss the validity and limitations of this work. In this research, *construct validity* denotes the degree to which our metrics reflect the correctness of code completion systems. Overall, we conduct automatic testing and manual inspection to study the outputs of de facto code completion systems. Hence, while this practical approach detects their defects and reveals chances of enhancing their outputs, a possible threat is that our testing approach cannot guarantee the correctness of code completion systems. We clarify that our work roots the same assumption as previous testing works that aim to detect flaws of deep learning-based applications with dynamic testing rather than verification.

We check code completion outputs by comparing a set of outputs that are supposed to be (visually) consistent. The evaluation shows that the focus of structural consistency effectively unveils a large number of defects. However, a possible threat is that defects can be neglected in the completion output, in case all outputs share aligned yet erroneous code patterns. We deem this a general and well-known hurdle for invariant property-based testing techniques. We leave exploring solutions to this challenge for future work.

Besides, the potential threat exists that the proposed testing and enhancing framework, CCTEST, may not adapt to other types of code completion systems. Nevertheless, we mitigate this threat to *external validity* by designing a system and algorithm independent approach. As a result, our approach is anticipated to apply to other settings outside the current scope. We believe the proposed technique is general, and we give further discussions regarding other settings in this section.

"Natural-Looking" Mutations. CCTEST designs nine PSC transformations to mutate prompts. Nevertheless, one may question if all of these mutations are common in real-world programming. Holistically, we agree that some transformations, e.g., renaming variables, may lead to potentially rare names, e.g., using `LocalVar1` as a variable name; as illustrated in under the `REL_R` scheme. Nevertheless, PSC transformations is intentionally designed as *lightweight*, for the seek of easing the follow-up outlier detection. We find that the present transformations are sufficient to expose many defects and harvest research insights. More importantly, we focus on designing *structure-preserving* transformations. Too aggressive transformations may easily break the structure consistency of mutated prompts, making our testing oracle inapplicable. Thus, while we also consider the specific program context (as noted in Table I) to enhance the "realism" of mutated prompts, realism is not our highest priority.

Overall, generating natural-looking, realistic inputs to test deep learning systems is inherently challenging. For instance, some computer vision (CV) related testing may need to use expensive generative models like GAN to generate more natural-looking images and test auto-driving systems [90]. It is unclear if those methods fit our scenarios. As a future work, we plan to study using advanced methods (e.g., LLMs) to generate mutated test inputs that are both "natural-looking" and structure-preserving. The cost may be a primary concern, as we generated around 20K mutated test inputs in the evaluation.

Cross Comparison of Code Completion Outputs. Overall, CCTEST *individually* tests each code completion system. The proposed approach constitutes program property-based testing (or metamorphic testing). With this regard, careful readers may wonder about the feasibility of conducting *differential testing* by processing the same prompt with different code completion systems and differentiating their outputs. However, we note that the code completion outputs can have drastically different representations since different code completion systems have different model training data and LLM model capacity. For instance, Copilot is seen to produce a large chunk of code snippets (with multiple statements), whereas some other well-known systems are prone to giving more succinct outputs for the same input prompts. Our preliminary exploration also shows that they manifest different tactics and translation templates in code generation. Thus, the similarity among the code completion outputs is deemed as *low* across different code completion systems. Overall, we leave it as one future work to explore practical methods to perform cross comparison, for instance, by extracting specific "semantics-level" signatures or regulating their output code patterns first.

Other Settings. CCTEST targets Python code completion, one challenging and popular task in software engineering. We believe it is feasible to migrate CCTEST to test code completion systems of other languages like C and Java. While extending CCTEST to handle other languages demands new parsers and re-implementing PSC schemes, we expect that the key technical pipeline, including mutation, outlier detection, and enhancing, are language *independent*. We thus believe migration is an engineering task rather than open-ended research problems.

VII. RELATED WORK

Testing Neural Models. Many works have applied testing methods to neural models [23, 51, 54, 55, 61, 73, 78, 87, 88, 90, 92]. The tested neural models include computer vision tasks like image classification, object detection [67, 72, 79], auto-driving [90, 92], as well as natural language processing tasks like sentiment analysis [29, 49, 65, 74], question answering [20], machine translation [35–37, 68], and specific properties like fairness [22]. Recent advances in machine translation testing inspire CCTEST. However, CCTEST addresses domain-specific challenges in mutating prompts, and for the first time, provides a systematic framework for testing code completion systems in black-box settings.

Neural Code Comprehension. Besides code completion task (the focus of CCTEST), neural models are used in other code comprehension tasks [13, 38, 62–64, 70, 84]. For instance, function naming decides the function name by summarizing the function body [10, 11, 94]. Often, code paths on the function ASTs are extracted for embedding and name prediction. Given the large volume of available paths, optimization schemes like attention are used to speed up the processing. Code classification and code search are two popular tasks. Recent methods learn from structural information (AST and CFG) for code classification [48, 50, 80] and code search [17, 32]. Tree-based convolutional neural networks and graphics neural networks are leveraged in these tasks [47, 75–77]. Software security applications have been built based on neural code comprehension, including plagiarism detection [43, 46], malware clustering [41], software component analysis [83, 86, 89], and vulnerability detection [45, 93]. CCTEST differs from these code comprehension tasks in that it focuses on testing code completion systems. While some recent works [13, 38, 62] treat neural models as a "white-box" for retraining and robustness augmentation on relatively simple DNNs, CCTEST delivers black-box repairing over LLMs.

VIII. CONCLUSION

We offer CCTEST, a testing and repairing tool for code completion systems. CCTEST uncovers thousands of defects, and its enhancement improves their output quality largely. This work may serve as a roadmap for researchers and users interested in using and improving code completion systems.

ACKNOWLEDGEMENT

We thank anonymous reviewers for their valuable feedback. HKUST authors are supported in part by a RGC ECS grant under the contract 26206520.

References

[1] Codex. https://openai.com/blog/openai-codex/.

[2] Copilot. https://github.com/features/copilot.

[3] Fuzzywuzzy. https://pypi.org/project/fuzzywuzzy/.

[4] Hugging Face. https://huggingface.co/.

[5] jurassic. shorturl.at/JTV26.

[6] Leetcode Python. https://github.com/JiayangWu/LeetCode-Python.

[7] Leetcode0285. https://leetcode.com/problems/inorder-successor-in-bst/.

[8] pycode-similar. https://pypi.org/project/pycode-similar/.

[9] Tabnine. https://www.tabnine.com/.

[10] Uri Alon, Shaked Brody, Omer Levy, and Eran Yahav. code2seq: Generating sequences from structured representations of code. *arXiv preprint arXiv:1808.01400*, 2018.

[11] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. code2vec: Learning distributed representations of code. POPL, 2019.

[12] Artifact. Cctest. https://sites.google.com/view/cctest-info, 2022.

[13] Pavol Bielik and Martin Vechev. Adversarial robustness for code. In *International Conference on Machine Learning*, pages 896–907. PMLR, 2020.

[14] Sid Black, Leo Gao, Phil Wang, Connor Leahy, and Stella Biderman. GPT-Neo: Large Scale Autoregressive Language Modeling with Mesh-Tensorflow, March 2021.

[15] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901, 2020.

[16] Max Brunsfeld. Tree-sitter. https://github.com/tree-sitter/tree-sitter.

[17] José Cambronero, Hongyu Li, Seohyun Kim, Koushik Sen, and Satish Chandra. When deep learning met code search. In Marlon Dumas, Dietmar Pfahl, Sven Apel, and Alessandra Russo, editors, *FSE*, 2019.

[18] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. Evaluating large language models trained on code. 2021.

[19] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.

[20] Songqiang Chen, Shuo Jin, and Xiaoyuan Xie. Testing your question answering software via asking recursively. In *Proceedings of the 36th IEEE/ACM International Conference on Automated Software Engineering*, ASE '21, page 104–116. IEEE Press, 2021.

[21] Tsong Y Chen, Shing C Cheung, and Shiu Ming Yiu. Metamorphic testing: a new approach for generating next test cases. Technical report, Technical Report HKUST-CS98-01, Department of Computer Science, Hong Kong ..., 1998.

[22] Zhenpeng Chen, Jie M Zhang, Max Hort, Federica Sarro, and Mark Harman. Fairness testing: A comprehensive survey and analysis of trends. *arXiv e-prints*, pages arXiv–2207, 2022.

[23] Anurag Dwarakanath, Manish Ahuja, Samarth Sikand, Raghotham M. Rao, R. P. Jagadeesh Chandra Bose, Neville Dubash, and Sanjay Podder. Identifying implementation bugs in machine learning based image classifiers using metamorphic testing. In *ISSTA*, 2018.

[24] EleutherAI. Gpt-j. https://huggingface.co/EleutherAI/gpt-j-6B.

[25] Hugging Face. Codeparrot. https://huggingface.co/codeparrot/codeparrot.

[26] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. Codebert: A pre-trained model for programming and natural languages. In Trevor Cohn, Yulan He, and Yang Liu, editors, *Findings of the Association for Computational Linguistics: EMNLP 2020, Online Event, 16-20 November 2020*, volume EMNLP 2020 of *Findings of ACL*, pages 1536–1547. Association for Computational Linguistics, 2020.

[27] Joseph L Fleiss. Measuring nominal scale agreement among many raters. *Psychological bulletin*, 76(5):378, 1971.

[28] Philip Gage. A new algorithm for data compression. *C Users Journal*, 12(2):23–38, 1994.

[29] Sainyam Galhotra, Yuriy Brun, and Alexandra Meliou. Fairness testing: testing software for discrimination. In *ACM ESEC/FSE*, pages 498–510. ACM, 2017.

[30] Leo Gao, Stella Biderman, Sid Black, Laurence Golding, Travis Hoppe, Charles Foster, Jason Phang, Horace He, Anish Thite, Noa Nabeshima, Shawn Presser, and Connor Leahy. The Pile: An 800gb dataset of diverse text for language modeling. *arXiv preprint arXiv:2101.00027*, 2020.

[31] Karan Goel, Nazneen Rajani, Jesse Vig, Samson Tan, Jason Wu, Stephan Zheng, Caiming Xiong, Mohit Bansal, and Christopher Ré. Robustness gym: Unifying the nlp evaluation landscape. *arXiv preprint arXiv:2101.04840*, 2021.

[32] Wenchao Gu, Zongjie Li, Cuiyun Gao, Chaozheng Wang, Hongyu Zhang, Zenglin Xu, and Michael R. Lyu. Cradle: Deep code retrieval based on semantic dependency learning. *Neural Networks*, 141:385–394, 2021.

[33] Daya Guo, Shuai Lu, Nan Duan, Yanlin Wang, Ming Zhou, and Jian Yin. Unixcoder: Unified cross-modal pre-training for code representation. *arXiv preprint arXiv:2203.03850*, 2022.

[34] Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, et al. Graphcodebert: Pre-training code representations with data flow. *arXiv preprint arXiv:2009.08366*, 2020.

[35] Shashij Gupta, Pinjia He, Clara Meister, and Zhendong Su. Machine translation testing via pathological invariance. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 863–875, 2020.

[36] Pinjia He, Clara Meister, and Zhendong Su. Structure-invariant testing for machine translation. In *ICSE*, 2020.

[37] Pinjia He, Clara Meister, and Zhendong Su. Testing machine translation via referential transparency. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pages 410–422. IEEE, 2021.

[38] Jordan Henke, Goutham Ramakrishnan, Zi Wang, Aws Albarghouth, Somesh Jha, and Thomas Reps. Semantic robustness of models of source code. SANER, 2022.

[39] Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. Codesearchnet challenge: Evaluating the state of semantic code search. *arXiv preprint arXiv:1909.09436*, 2019.

[40] Saki Imai. Is github copilot a substitute for human pair-programming? an empirical study. In *2022 IEEE/ACM 44th International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, pages 319–321. IEEE, 2022.

[41] Mahmoud Kalash, Mrigank Rochan, Noman Mohammed, Neil DB Bruce, Yang Wang, and Farkhund Iqbal. Malware classification with deep convolutional neural networks. In *2018 9th IFIP international conference on new technologies, mobility and security (NTMS)*, pages 1–5. IEEE, 2018.

[42] Klemens Lagler, Michael Schindelegger, Johannes Böhm, Hana Krásná, and Tobias Nilsson. GPT2: Empirical slant delay model for radio space geodetic techniques. *Geophysical research letters*, 40(6):1069–1073, 2013.

[43] Maggie Lei, Hao Li, Ji Li, Namrata Aundhkar, and Dae-Kyoo Kim. Deep learning application on code clone detection: A review of current knowledge. *Journal of Systems and Software*, 184:111141, 2022.

[44] Yujia Li, David H. Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, Thomas Hubert, Peter Choy, Cyprien de Masson d'Autume, Igor Babuschkin, Xinyun Chen, Po-Sen Huang, Johannes Welbl, Sven Gowal, Alexey Cherepanov, James Molloy, Daniel J. Mankowitz, Esme Sutherland Robson, Pushmeet Kohli, Nando de Freitas, Koray Kavukcuoglu, and Oriol Vinyals. Competition-level code generation with alphacode. *CoRR*, abs/2203.07814, 2022.

[45] Zhen Li, Deqing Zou, Shouhuai Xu, Xinyu Ou, Hai Jin, Sujuan Wang, Zhijun Deng, and Yuyi Zhong. Vuldeepecker: A deep learning-based system for vulnerability detection. *arXiv preprint arXiv:1801.01681*, 2018.

[46] Zongjie Li, Pingchuan Ma, Huaijin Wang, Shuai Wang, Qiyi Tang, Sen Nie, and Shi Wu. Unleashing the power of compiler intermediate representation to enhance neural program embeddings. ICSE, 2022.

[47] Shangqing Liu, Xiaofei Xie, Lei Ma, Jing Kai Siow, and Yang Liu. Graphsearchnet: Enhancing gnns via capturing global dependency for semantic code search. *CoRR*, abs/2111.02671, 2021.

[48] Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin Clement, Dawn Drain, Daxin Jiang, Duyu Tang, et al. Codexglue: A machine learning benchmark dataset for code understanding and generation. *arXiv preprint arXiv:2102.04664*, 2021.

[49] Pingchuan Ma, Shuai Wang, and Jin Liu. Metamorphic testing and certified mitigation of fairness violations in nlp models. In *IJCAI*, pages 458–465, 2020.

[50] Lili Mou, Ge Li, Lu Zhang, Tao Wang, and Zhi Jin. Convolutional neural networks over tree structures for programming language processing. In *Thirtieth AAAI conference on artificial intelligence*, 2016.

[51] Shin Nakajima and Tsong Yueh Chen. Generating biased dataset for metamorphic testing of machine learning programs. In *IFIP-ICTSS*, 2019.

[52] Nhan Nguyen and Sarah Nadi. An empirical evaluation of github copilot's code suggestions. In *IEEE/ACM 19th International Conference on Mining Software Repositories, MSR 2022, Pittsburgh, PA, USA, May 23-24, 2022*, pages 1–5. IEEE, 2022.

[53] Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. A conversational paradigm for program synthesis. *arXiv preprint arXiv:2203.13474*, 2022.

[54] Augustus Odena and Ian Goodfellow. Tensorfuzz: Debugging neural networks with coverage-guided fuzzing. *arXiv preprint arXiv:1807.10875*, 2018.

[55] Qi Pang, Yuanyuan Yuan, and Shuai Wang. Mdpfuzzer: Finding crash-triggering state sequences in models solving the markov decision process. ISSTA, 2022.

[56] Kishore Papineni, Salim Roukos, Todd Ward, and Wei jing Zhu. Bleu: a method for automatic evaluation of machine translation. pages 311–318, 2002.

[57] Hammond Pearce, Benjamin Tan, Baleegh Ahmad, Ramesh Karri, and Brendan Dolan-Gavitt. Can openai codex and other large language models help us fix security bugs? *arXiv preprint arXiv:2112.02125*, 2021.

[58] Hammond Pearce, Benjamin Tan, Baleegh Ahmad, Ramesh Karri, and Brendan Dolan-Gavitt. Can openai codex and other large language models help us fix security bugs? *CoRR*, abs/2112.02125, 2021.

[59] Hammond Pearce, Benjamin Tan, Prashanth Krishnamurthy, Farshad Khorrami, Ramesh Karri, and Brendan Dolan-Gavitt. Pop quiz! can a large language model help with reverse engineering? *arXiv preprint arXiv:2202.01142*, 2022.

[60] Hammond Pearce, Benjamin Tan, Prashanth Krishnamurthy, Farshad Khorrami, Ramesh Karri, and Brendan Dolan-Gavitt. Pop quiz! can a large language model help with reverse engineering? *CoRR*, abs/2202.01142, 2022.

[61] Kexin Pei, Yinzhi Cao, Junfeng Yang, and Suman Jana. DeepXplore: Automated whitebox testing of deep learning systems. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, pages 1–18, New York, NY, USA, 2017. ACM.

[62] Maryam Vahdat Pour, Zhuo Li, Lei Ma, and Hadi Hemmati. A search-based testing framework for deep neural networks of source code embedding. ICST, 2021.

[63] Md Rafiqul Islam Rabin, Nghi DQ Bui, Ke Wang, Yijun Yu, Lingxiao Jiang, and Mohammad Amin Alipour. On the generalizability of neural program models with respect to semantic-preserving program transformations. *IST*, 2021.

[64] Md Rafiqul Islam Rabin, Vincent J Hellendoorn, and Mohammad Amin Alipour. Understanding neural code intelligence through program simplification. FSE, 2021.

[65] Marco Tulio Ribeiro, Tongshuang Wu, Carlos Guestrin, and Sameer Singh. Beyond accuracy: Behavioral testing of nlp models with checklist. *arXiv preprint arXiv:2005.04118*, 2020.

[66] Sami Sarsa, Paul Denny, Arto Hellas, and Juho Leinonen. Automatic generation of programming exercises and code explanations with large language models. *arXiv preprint arXiv:2206.11861*, 2022.

[67] Jinyang Shao. Testing object detection for autonomous driving systems via 3d reconstruction. In *ICSE-Companion*, 2021.

[68] Zeyu Sun, Jie M Zhang, Mark Harman, Mike Papadakis, and Lu Zhang. Automatic testing and improvement of machine translation. ICSE, 2020.

[69] Zeyu Sun, Jie M Zhang, Yingfei Xiong, Mark Harman, Mike Papadakis, and Lu Zhang. Improving machine translation systems via isotopic replacement. In *ICSE*, 2022.

[70] Sahil Suneja, Yunhui Zheng, Yufan Zhuang, Jim A Laredo, and Alessandro Morari. Probing model signal-awareness via prediction-preserving input minimization. FSE, 2021.

[71] Yi Tay, Mostafa Dehghani, Dara Bahri, and Donald Metzler. Efficient transformers: A survey. *ACM Computing Surveys (CSUR)*, 2020.

[72] Yongqiang Tian, Shiqing Ma, Ming Wen, Yepang Liu, Shing-Chi Cheung, and Xiangyu Zhang. To what extent do dnn-based image classification models make unreliable inferences? *Empirical Software Engineering*, 26(5):1–40, 2021.

[73] Yuchi Tian, Kexin Pei, Suman Jana, and Baishakhi Ray. DeepTest: Automated testing of deep-neural-network-driven autonomous cars. ICSE '18, 2018.

[74] Sakshi Udeshi, Pryanshu Arora, and Sudipta Chattopadhyay. Automated directed fairness testing. ASE, 2018.

[75] Huaijin Wang, Pingchuan Ma, Shuai Wang, Qiyi Tang, Sen Nie, and Shi Wu. sem2vec: Semantics-aware assembly tracelet embedding. *ACM Transactions on Software Engineering and Methodology*, 2022.

[76] Huaijin Wang, Pingchuan Ma, Yuanyuan Yuan, Zhibo Liu, Shuai Wang, Qiyi Tang, Sen Nie, and Shi Wu. Enhancing dnn-based binary code function search with low-cost equivalence checking. *IEEE Transactions on Software Engineering*, 2022.

[77] Huanting Wang, Guixin Ye, Zhanyong Tang, Shin Hwei Tan, Songfang Huang, Dingyi Fang, Yansong Feng, Lizhong Bian, and Zheng Wang. Combining graph-based learning with automated data collection for code vulnerability detection. *IEEE TIFS*, 16:1943–1958, 2020.

[78] Jingyi Wang, Guoliang Dong, Jun Sun, Xinyu Wang, and Peixin Zhang. Adversarial sample detection for deep neural network through model mutation testing. ICSE, 2019.

[79] Shuai Wang and Zhendong Su. Metamorphic object insertion for testing object detection systems. In *ASE*, 2020.

[80] Yue Wang, Weishi Wang, Shafiq Joty, and Steven CH Hoi. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. *arXiv preprint arXiv:2109.00859*, 2021.

[81] Yue Wang, Weishi Wang, Shafiq R. Joty, and Steven C. H. Hoi. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. In Marie-Francine Moens, Xuanjing Huang, Lucia Specia, and Scott Wen-tau Yih, editors, *EMNLP*, 2021.

[82] Dror Weiss. Comparison between copilot and tabnine. https://twitter.com/drorwe/status/1539329682851733505.

[83] Seunghoon Woo, Sunghan Park, Seulbae Kim, Heejo Lee, and Hakjoo Oh. Centris: A precise and scalable approach for identifying modified open-source software reuse. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pages 860–872. IEEE, 2021.

[84] Noam Yefet, Uri Alon, and Eran Yahav. Adversarial examples for models of code. *Proceedings of the ACM on Programming Languages*, 4(OOPSLA):1–30, 2020.

[85] Sangwon Yu, Jongyoon Song, Heeseung Kim, Seongmin Lee, Woo-Jong Ryu, and Sungroh Yoon. Rare tokens degenerate all tokens: Improving neural text generation via adaptive gradient gating for rare token embeddings. In *ACL*, 2022.

[86] Zeping Yu, Wenxin Zheng, Jiaqi Wang, Qiyi Tang, Sen Nie, and Shi Wu. Codecmr: Cross-modal retrieval for function-level binary source code matching. *Advances in Neural Information Processing Systems*, 33:3872–3883, 2020.

[87] Yuanyuan Yuan, Qi Pang, and Shuai Wang. Unveiling hidden dnn defects with decision-based metamorphic testing. ASE, 2022.

[88] Yuanyuan Yuan, Shuai Wang, Mingyue Jiang, and Tsong Yueh Chen. Perception matters: Detecting perception failures of vqa models using metamorphic testing. CVPR, 2021.

[89] Xian Zhan, Lingling Fan, Sen Chen, Feng We, Tianming Liu, Xiapu Luo, and Yang Liu. Atvhunter: Reliable version detection of third-party libraries for vulnerability identification in android applications. In *ICSE*.

[90] Mengshi Zhang, Yuqun Zhang, Lingming Zhang, Cong Liu, and Sarfraz Khurshid. DeepRoad: GAN-based Metamorphic Testing and Input Validation Framework for Autonomous Driving Systems. In *ASE*, 2018.

[91] Zhaowei Zhang, Hongyu Zhang, Beijun Shen, and Xiaodong Gu. Diet code is healthy: Simplifying programs for pre-trained models of code. *arXiv preprint arXiv:2206.14390*, 2022.

[92] Husheng Zhou, Wei Li, Yuankun Zhu, Yuqun Zhang, Bei Yu, Lingming Zhang, and Cong Liu. Deepbillboard: Systematic physical-world testing of autonomous driving systems. ICSE, 2020.

[93] Yaqin Zhou, Shangqing Liu, Jingkai Siow, Xiaoning Du, and Yang Liu. Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks. NeuIPS, 2019.

[94] Daniel Zügner, Tobias Kirschstein, Michele Catasta, Jure Leskovec, and Stephan Günnemann. Language-agnostic representation learning of source code from structure and context. ICLR, 2021.