

# LLM-Powered Test Case Generation for Detecting Bugs in Plausible Programs

Kaibo Liu<sup>1</sup>, Zhenpeng Chen<sup>2,\*</sup>, Yiyang Liu<sup>1</sup>, Jie M. Zhang<sup>3</sup>, Mark Harman<sup>4</sup>,  
Yudong Han<sup>1</sup>, Yun Ma<sup>1</sup>, Yihong Dong<sup>1</sup>, Ge Li<sup>1,\*</sup>, Gang Huang<sup>1,5</sup>

<sup>1</sup>Peking University, <sup>2</sup>Nanyang Technological University,

<sup>3</sup>King's College London, <sup>4</sup>University College London

<sup>5</sup>National Key Laboratory of Data Space Technology and System

{liukb, hanyd, mayun, lige, hg}@pku.edu.cn, zhenpeng.chen@ntu.edu.sg

{ptr1479, dongyh}@stu.pku.edu.cn, jie.zhang@kcl.ac.uk, mark.harman@ucl.ac.uk

## Abstract

Detecting tricky bugs in plausible programs, those that pass existing test suites yet still contain bugs, remains a significant challenge in software testing. To address this problem, we propose TrickCatcher, an LLM-powered approach to generating test cases for uncovering bugs in plausible programs. TrickCatcher operates in three stages: First, it uses an LLM to generate program variants based on the program under test (PUT) and its specification. Second, it employs an LLM to construct an input generator from the specification for producing test inputs. Finally, these inputs are executed on both the PUT and its program variants to detect inconsistencies in their outputs. We evaluate TrickCatcher on two datasets, Tricky-Bugs and EvalPlus, which include 366 human-written and 151 AI-generated plausible programs with tricky bugs. TrickCatcher achieves recall, precision, and F1 scores that are 1.80 $\times$ , 2.65 $\times$ , and 1.66 $\times$  those of the state-of-the-art baselines, respectively. Code and data used are available at <https://github.com/RinCloud/TrickCatcher>.

## 1 Introduction

Validating that programs meet a given specification that defines their intended functionality is critical, with software testing serving as the primary approach to achieving this goal. Central to software testing lies in the use of a *test suite*, a collection of test cases designed to validate the program under test (PUT). Each test case consists of a *test input*, sampled from the input space of the PUT, and a *test oracle*, which specifies the expected output based on the program's specification.

Programs that pass all test cases are considered *plausible programs* (Liu et al., 2024b; Chen et al., 2024a), but plausibility does not equate to correctness. Such plausible programs may still harbor subtle bugs (Liu et al., 2023b; Tambon et al., 2024; Gu

\*Corresponding authors

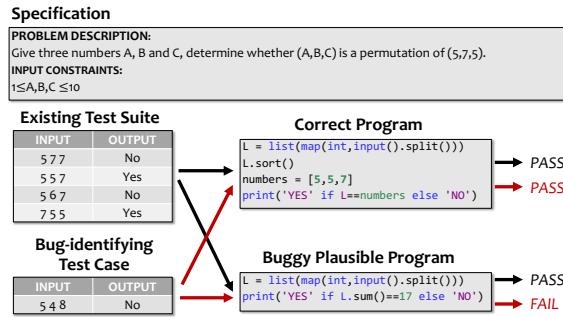


Figure 1: A motivating example.

et al., 2024), often logical corner cases, that escape detection by test suites. We refer to these elusive bugs as *tricky bugs*. For example, Figure 1 shows a real-world plausible program from an online judge platform<sup>1</sup> that passes an existing test suite yet conceals a tricky bug. A correct program should check whether the set of the three input numbers is {5, 7, 5}, while the buggy plausible program only checks if the sum of the three numbers is 17.

In fact, tricky bugs are surprisingly common. A recent study (Liu et al., 2023b) identified 3,440 such bugs in human-written programs deemed correct by Online Judge platforms, underscoring their prevalence even in scenarios where code has been thoroughly reviewed and tested. Despite their significant impact, existing research has largely overlooked the development of testing approaches specifically designed to uncover tricky bugs in plausible programs, leaving a critical gap in current testing practices.

To fill the gap, we propose TrickCatcher, an LLM-powered test case generation approach for detecting tricky bugs in plausible programs. TrickCatcher combines LLMs and differential testing to accurately generate test inputs and test oracles. It consists of three steps: *program variant generation*,

<sup>1</sup>[https://atcoder.jp/contests/abc042/tasks/abc042\\_a](https://atcoder.jp/contests/abc042/tasks/abc042_a)

*test input generation*, and *differential testing*. In the first two steps, we use LLMs to generate various program variants and test inputs of the PUT. In the third step, we continually feed generated test inputs to both the PUT and program variants, searching for inconsistencies in program outputs.

A straightforward approach is to directly use the specification as input for LLMs to generate program variants and test inputs. However, our preliminary experiments reveal two major limitations of this naive approach: ❶ *Low correctness of program variants*. While LLMs can generate correct programs for simple tasks based on specifications (Chen et al., 2021), their performance deteriorates with complex tasks (e.g., competition-level programs). The resulting variants often contain errors, reducing the effectiveness of differential testing. ❷ *Low correctness of test inputs*. Although LLMs can generate valid test inputs for simple formats (e.g., two integers), they struggle with inputs requiring constraints (e.g., a square matrix with monotonically increasing rows). Our experiments show that when test inputs are generated directly from specified input constraints, 40.10% of the generated test inputs are invalid. Invalid inputs fail to expose bugs, as they result in undefined program behavior, and worse, they can cause *false positives*, incorrectly marking a working program as buggy.

To tackle these challenges, we propose the following three solutions in TrickCatcher:

**❶ PUT-guided program variant generation.** Instead of relying solely on specifications, TrickCatcher provides the PUT alongside the specification in the LLM prompt. The LLM is tasked with analyzing the PUT and generating repaired program variants if necessary. These variants are then filtered using existing test cases to exclude those that fail. By leveraging the PUT as a foundation, TrickCatcher steers the LLM toward generating meaningful modifications rather than creating implementations from scratch, significantly improving the quality of the program variants.

**❷ Generator-based input generation.** Instead of directly generating test inputs from specifications, TrickCatcher instructs the LLM to create an input generator (e.g., a Python script) that adheres to specified constraints and then uses the generator to generate test inputs. This approach separates logical reasoning from input generation, allowing TrickCatcher to overcome the reasoning limitations of LLMs. As a result, it significantly improves the validity of the generated inputs, achieving a higher

proportion of valid test cases.

**❸ Diversity-driven differential testing.** Recognizing that program variants can inherit similar bugs from the PUT, TrickCatcher departs from the traditional majority-voting principle, which assumes the most frequent output to be correct. Instead, it prioritizes diversity in test outputs to construct test oracles. This counterintuitive approach enhances the ability to detect subtle discrepancies and uncover tricky bugs that majority voting might overlook.

We evaluate TrickCatcher on two datasets, TrickyBugs (Liu et al., 2024b) and EvalPlus (Liu et al., 2023a), which contain 366 human-written and 151 AI-generated plausible programs with tricky bugs, respectively. TrickyBugs includes both C++ and Python programs, while EvalPlus focuses on Python. TrickCatcher is compared against three representative baselines, and the results demonstrate its superior performance in recall, precision, and F1 score, achieving up to **1.80 $\times$** , **2.65 $\times$** , and **1.66 $\times$**  of the best baseline, respectively. In particular, TrickCatcher achieves F1 scores of 41.31%, 42.35%, and 51.34% on TrickyBugs (C++), TrickyBugs (Python), and EvalPlus, significantly outperforming the best baseline’s F1 scores of 24.95%, 36.20%, and 35.76%. An ablation study further confirms that each component of TrickCatcher contributes meaningfully to its overall performance.

## 2 Related Work

**Traditional test case generation.** Traditional test case generation methods primarily rely on search-based (McMinn, 2004, 2011) and symbolic execution-based approaches (Baldoni et al., 2018), with popular tools such as EvoSuite (Fraser and Arcuri, 2011), Pynguin (Lukasczyk and Fraser, 2022), and KLEE (Cadar et al., 2008) exemplifying these approaches. However, these traditional approaches cannot automatically parse program specifications, which are often written in natural language and crucial for bug detection. In contrast, TrickCatcher leverages LLMs to interpret and utilize program specifications effectively.

**LLM-based test case generation.** Recently, LLMs have been widely adopted in test case generation approaches (Liu et al., 2024a), such as ChatTester (Yuan et al., 2024), TestPilot (Schäfer et al., 2024), ChatUnitTest (Chen et al., 2024b), and SymPrompt (Ryan et al., 2024). However, these approaches differ from TrickCatcher as they primarily focus on improving test coverage rather than

detecting bugs. In terms of bug detection, Differential Prompting (DP) (Li et al., 2023) represents the state-of-the-art in LLM-based test case generation. As a differential testing approach, DP also generates program variants to identify potential bugs. The key differences between DP and TrickCatcher are as follows: (1) TrickCatcher focuses on plausible programs, which enables it to consider the existing test suite when generating inputs, whereas DP, which is not designed for plausible programs, does not; (2) TrickCatcher uses both the PUT and its specification for program variant generation, while DP relies solely on the inferred specification; (3) TrickCatcher employs LLMs to generate input generators, which are then used to produce test inputs, in contrast to DP, which directly generates inputs from the specification; and (4) TrickCatcher introduces diversity-driven differential testing, whereas DP uses majority voting, a traditional approach in differential testing, for test oracle construction. We have implemented a variant of DP to make it applicable to plausible programs and compared it with TrickCatcher. Moreover, we thoroughly evaluate these designs of TrickCatcher in Section 6.3.

### 3 Problem Definition

In this section, we define the problem of generating test cases to detect bugs in plausible programs.

Given a program specification  $S$ , it defines an intended mapping  $f$  from the input space  $I$  to the output space  $O$ . For any input  $in \in I$ , the correct output (test oracle) is  $out = f(in)$ . The goal of bug-identifying test case generation is to generate an input  $int$  along with its corresponding correct output  $f(int)$ , forming the test case  $(int, f(int))$ , such that the program under test (PUT) produces an incorrect output, i.e.,  $f_{PUT}(int) \neq f(int)$ .

If a program  $P_0$  passes all test cases in a test suite  $T_0$ , we call  $P_0$  a *plausible program* (relative to  $T_0$ ). However, if  $P_0$  still contains a bug, we refer to it as a *buggy plausible program* (Chen et al., 2024a; Gu et al., 2024), and the bugs in  $P_0$  are termed *tricky bugs* (Liu et al., 2023b, 2024b).

The objective of this paper is to generate test cases for a plausible program  $P_0$  based on its specification  $S$ . A failed test case is considered a potential bug identifier as it reveals a discrepancy between the program’s output and the expected output (i.e.,  $f_{PUT}(in_i) \neq f(in_i)$ ). However, not every failed test case necessarily indicates a bug in the program, as failures can also result from errors

in the test case itself.

For a test case  $(in, out)$  to be valid, it must meet the following two conditions: **(1) Validity of the test input:** The input  $in$  must belong to the valid input space  $I$ . **(2) Correctness of the test oracle:** The output  $out$  must satisfy  $out = f(in)$ , where  $f$  is the mapping specified by  $S$ .

If a test case  $(in, out)$  fails, and both conditions are satisfied, the test case is a *True Positive* (TP), indicating the presence of a bug in the PUT. If the test case fails to meet one or both conditions, it results in a false alarm, which is categorized as a *False Positive* (FP).

## 4 Our Approach: TrickCatcher

Figure 2 provides an overview of TrickCatcher. It takes the program specification, the PUT, and the existing test suite as inputs, and outputs a set of test cases designed to detect bugs in the PUT. The workflow of TrickCatcher consists of three key steps: PUT-guided program variant generation, generator-based test input generation, and diversity-driven differential testing. The first two steps use LLMs to generate multiple program variants and test inputs for the PUT, while the third step iteratively feeds these test inputs into both the PUT and its program variants, searching for inconsistencies in the outputs and constructing potential bug-identifying test cases. Each step is explained in detail below.

### 4.1 Program Variant Generation

The first step of TrickCatcher involves generating program variants of the PUT. Both the program specification and the PUT are provided to the LLM, which is prompted to assess whether the PUT contains any bugs based on the specification. If the LLM detects a potential bug, it is tasked with repairing the program and generating a corrected version. The prompt used is illustrated in Figure 3.

The key advantage of PUT-guided program generation is its ability to leverage both the program specification and the PUT. Since the PUT is a plausible program, it already exhibits a certain level of correctness, particularly for the input space covered by the existing test suite. By generating variants based on the PUT, the LLM is more likely to produce high-quality variants with a reduced risk of introducing new bugs compared to generating variants directly from the specification alone.

To improve the correctness of the program variants, TrickCatcher filters out any variants that fail

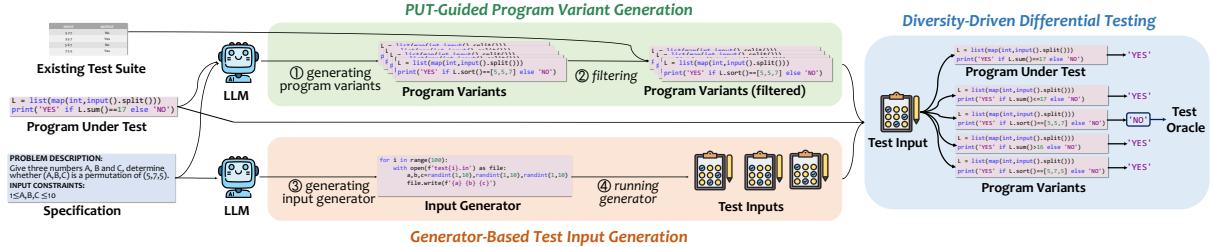


Figure 2: Overview of TrickCatcher.

```
**INSTRUCTION**:
You are a professional coding competition participant, skilled at identifying bugs and logic flaws in code.
You will receive a description of a coding problem, and a piece of code attempting to solve the problem.
Your task is to find whether there is any bug or logic flaw in the code, if any, please repair the code.
Please reply with ONLY the COMPLETE REPAIRED CODE (rather than code fragments) without any other content.

**PROBLEM DESCRIPTION**:
{The specification of the coding task}

**CODE**:
{Source code of PUT}
```

Figure 3: Prompt for generating program variants.

the existing test suite. This filtering step makes effective use of the information provided by the test suite, ensuring that only high-quality program variants are retained for the subsequent steps of differential testing.

## 4.2 Test Input Generation

```
**INSTRUCTION**:
The following is a description of a coding problem, please write an input generator for this problem (DO NOT generate outputs).
The generated inputs should meet the input constraints of the problem description.
Please reply with ONLY the code without any other content.

You can use the Python library {library name} if necessary, here are some examples of how to use the library, which may be helpful:
{Few-shot examples to use the library}

**PROBLEM DESCRIPTION**:
{The specification of the coding task}
```

Figure 4: Prompt for generating test input generator.

The second step of TrickCatcher involves generating test inputs. The main challenge here is ensuring that the generated test inputs are valid, meaning that they satisfy the required input constraints.

Directly generating test inputs based on the specification using LLMs often results in invalid inputs, as LLMs have limited reasoning capabilities, particularly when the specified constraints are complex. To address this challenge, we adopt a two-step approach: first, we prompt the LLM to summarize the constraints and translate them into code. Then, we use the generated code to produce valid test inputs.

Specifically, TrickCatcher proposes generator-based input generation. The LLM is tasked with creating a test input generator, which is then executed to produce the inputs. In our approach, the generator is specified as a Python script, as shown in the prompt in Figure 4. To enhance the generator's capabilities, we can provide the LLM with a library of functions through few-shot learning examples, enabling it to efficiently learn how to use the library. We choose the Python library CYaRon in our experiment, and the library can be easily replaced by adjusting the few-shot examples.

### Algorithm 1 Diversity-driven differential testing

**INPUT:** PUT, Set of program variants  $\mathcal{P}$ , Set of generated inputs  $\mathcal{I}$

**OUTPUT:** Test cases

```

1: testCases  $\leftarrow \emptyset$ 
2: for input  $\in \mathcal{I}$  do
3:   diff  $\leftarrow \emptyset$ 
4:   output0  $\leftarrow f_{PUT}(\text{input})$ 
5:   for P  $\in \mathcal{P}$  do
6:     output  $\leftarrow f_P(\text{input})$ 
7:     if output  $\neq$  output0 then
8:       Add output to diff
9:     end if
10:   end for
11:   if diff  $\neq \emptyset$  then
12:     oracle  $\leftarrow$  most frequent element in diff
13:     test  $\leftarrow (\text{input}, \text{oracle})$ 
14:     Add test to testCases
15:   end if
16: end for
17: return testCases
```

## 4.3 Differential Testing

The third step of TrickCatcher involves constructing a test oracle through differential testing. As shown in Figure 2, TrickCatcher feeds the generated test inputs to both the PUT and the generated program variants, searching for inconsistencies in their outputs.

The algorithm is detailed in Algorithm 1. TrickCatcher introduces diversity-driven differential testing: if a program variant produces an output dif-

ferent from the PUT’s output, we take the variant’s output as the test oracle (the correct output). If multiple outputs differ from the PUT’s, the most frequent output is selected as the oracle. If all outputs match the PUT, the input is discarded, and the next input is tested.

This approach is counterintuitive, as developers typically rely on majority voting in differential testing (Liu et al., 2023b). The rationale behind our algorithm is that while the LLM may correctly replicate parts of the PUT, it can also be misled by the PUT. As a result, program variants may inherit the same bugs as the PUT. Our experiments show that it is common for some variants to produce the same erroneous output as the PUT. Thus, we place greater trust in program variants that differ from the PUT’s output.

## 5 Evaluation

### 5.1 Research Questions (RQs)

We aim to comprehensively evaluate TrickCatcher by answering the following RQs.

**RQ1:** How effective are the test cases generated by TrickCatcher in detecting bugs in plausible programs?

**RQ2:** How many false positives does TrickCatcher generate when applied to correct programs?

**RQ3:** How do the different components contribute to the final performance of TrickCatcher?

**RQ4:** How does the number of program variants impact the effectiveness of TrickCatcher?

**RQ5:** How does the difficulty of coding tasks impact the effectiveness of TrickCatcher?

### 5.2 Datasets

We evaluate TrickCatcher using two datasets: *TrickyBugs* (Liu et al., 2024b), which tests its ability to detect bugs in human-written plausible programs, and *EvalPlus* (Liu et al., 2023a), which assesses its effectiveness on AI-generated plausible programs.<sup>2</sup>

In total, we use 366 human-written and 151 AI-generated plausible programs as PUTs. Below are brief descriptions of the two datasets:

- **TrickyBugs** contains hundreds of coding tasks from an online judge platform, with plausible programs submitted by real participants. Although these programs pass the existing test suite, they contain bugs, and the dataset provides additional

<sup>2</sup>The two datasets are under the MIT and Apache 2.0 licenses, respectively.

test cases to detect them. We use 251 C++ and 115 Python plausible programs from this dataset.

- **EvalPlus** is a code generation benchmark including 164 Python coding tasks, each with base and extra test cases. We filter programs from EvalPlus’s pre-generated LLM code samples<sup>3</sup> to obtain buggy plausible programs that pass the base test cases but fail the extra ones. The final set includes 151 coding tasks, each with an AI-generated plausible program.

### 5.3 Evaluation Metrics

We have defined TP and FP in Section 3. To distinguish between TPs and FPs, we need to know the correct outputs and validity of the generated test inputs. For correct outputs, both datasets provide canonical programs, and we use the outputs from these canonical programs as the reference for correctness. To validate the test inputs, we use the provided Python checkers for the EvalPlus dataset, while for the TrickyBugs dataset, we manually verify the input validity.

We further define precision, recall, and F1-score:

- **Precision** is defined as  $\frac{\#TP}{\#TP + \#FP}$ . It determines the practicality of test generation approaches (Liu et al., 2023c). The higher the precision, the fewer false positive test cases developers need to check before confirming a true bug.

- **Recall** is defined as  $\frac{\#TP}{\#TP + \#FN}$ . Note that for buggy PUTs, all negatives are false negatives; for correct PUTs, all negatives are true negatives.

- **F1 score** is defined as  $\frac{2 \times \text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}}$ . Since recall and precision often exhibit an inverse relationship, the F1-score provides a harmonic mean to balance them, making it a widely used metric.

To ensure the reliability of our results, we conduct multiple runs of our experiments and calculate the average metric values across these runs. Detailed information on the repetition methods is provided in the Appendix B.

### 5.4 Baselines

We use three representative methods as baselines to compare with TrickCatcher (TC).

- **DirectChat (CHAT)**: It provides the LLM with the PUT and the specification, then asks it to generate bug-identifying test cases directly.

- **Differential Prompting Plus (DPP)**: As described in Section 2, Differential Prompting (Liu et al., 2023) is a state-of-the-art test case generation

<sup>3</sup><https://github.com/evalplus/evalplus/releases/tag/v0.1.0>

Table 1: (RQ1) Effectiveness of different methods in detecting bugs in plausible programs.  $k$  denotes the number of generated program variants. R, P, and F denote recall, precision, and F1 score. Bold numbers indicate the best F1 scores. Overall, TrickCatcher shows the best effectiveness in detecting bugs in both human-written and AI-generated plausible programs.

$k$	TrickyBugs (C++)			TrickyBugs (Python)			EvalPlus			
	R	P	F	R	P	F	R	P	F	
CHAT	-	3.78	6.31	4.27	3.77	8.85	5.29	1.21	8.28	2.12
APR	-	16.46	34.58	22.30	10.20	36.64	15.96	41.39	49.97	45.28
DPP	2	20.96	26.37	23.35	32.54	40.79	36.20	23.36	53.54	32.52
	4	19.46	32.22	24.27	28.72	46.32	35.46	23.01	61.16	33.45
	6	17.68	41.60	24.81	26.17	53.03	35.05	22.80	72.71	34.71
	8	16.31	53.09	24.95	24.41	62.02	35.03	22.55	80.51	35.23
	10	15.50	60.80	24.71	23.22	70.26	34.90	22.29	90.36	35.76
	2	25.06	69.91	36.90	23.74	78.95	36.51	32.78	85.09	47.33
TrickCatcher	4	27.74	69.69	39.68	27.17	77.81	40.28	35.35	84.41	49.83
	6	28.65	69.45	40.57	28.69	77.81	41.92	36.32	83.85	50.69
	8	29.19	69.34	41.09	29.09	77.81	42.35	36.90	83.38	51.16
	10	29.38	69.57	<b>41.31</b>	29.09	77.81	<b>42.35</b>	37.14	83.14	<b>51.34</b>
	Average	55.73% $\uparrow$	62.54% $\uparrow$	63.44% $\uparrow$	2.01% $\uparrow$	43.23% $\uparrow$	15.16% $\uparrow$	56.56% $\uparrow$	17.19% $\uparrow$	45.83% $\uparrow$
Improvement	Best vs. Best	80.13% $\uparrow$	31.04% $\uparrow$	65.57% $\uparrow$	10.61% $\downarrow$	90.76% $\uparrow$	16.99% $\uparrow$	66.62% $\uparrow$	7.99% $\downarrow$	43.57% $\uparrow$
	Worst vs. Worst	19.56% $\uparrow$	165.11% $\uparrow$	58.03% $\uparrow$	2.24% $\uparrow$	12.37% $\uparrow$	4.61% $\uparrow$	40.33% $\uparrow$	58.93% $\uparrow$	45.50% $\uparrow$

approach. While it is not designed or evaluated for detecting bugs in plausible programs, we recognize its potential in this area and use it as a baseline. The original method requires inferring the specification from the PUT; however, for a fair comparison, we use the ground truth specification instead. Thus, this modified version of Differential Prompting is referred to as Differential Prompting Plus (DPP).

• **Automated Program Repair (APR).** We introduce an additional baseline, which is an automated program repair (APR) method. This method provides the LLM with the PUT and specification, asking it to generate repair patches. In this scenario, correct patches are true positives, incorrect plausible patches are false positives, and other patches are negatives. This baseline corresponds to the first step of TrickCatcher, and we include it to show that TrickCatcher’s bug detection capability is not solely dependent on the LLM-generated repair patches.

## 5.5 Implementation

We use gpt-3.5-turbo-0125 as the LLM for implementing TrickCatcher and the baselines, striking a balance between performance and cost to conduct our extensive evaluation within budget constraints.

## 6 Results

### 6.1 RQ1: Performance on Buggy Plausible Programs

Table 1 shows the effectiveness of TrickCatcher and baselines in generating bug-identifying test cases

for TrickyBugs and EvalPlus datasets. The last three rows highlight the improvements of TrickCatcher compared to the best baseline, DPP. To ensure a comprehensive comparison, we use three distinct methods: “Average”, “Best vs. Best”, and “Worst vs. Worst.” The “Average” comparison computes the mean value across all different  $k$  values. “Best vs. Best” uses the  $k$  values with the best F1 scores for comparison. For example, DPP achieves its best F1 score (24.95) on TrickyBugs (C++) at  $k = 8$ , while TrickCatcher reaches its best F1 score (41.31) at  $k = 10$ ; we then compute the improvement of TrickCatcher ( $k = 10$ ) over DPP ( $k = 8$ ). “Worst vs. Worst” follows a similar approach but chooses the  $k$  values with the worst F1 scores.

The evaluation results demonstrate TrickCatcher’s superior performance in recall, precision, and F1 score, achieving up to  $1.80\times$ ,  $2.65\times$ , and  $1.66\times$  of DPP, respectively. Furthermore, TrickCatcher achieves the highest F1 score across all datasets. Specifically, TrickCatcher achieves F1 scores of 41.31%, 42.35%, and 51.34% on TrickyBugs (C++), TrickyBugs (Python), and EvalPlus, respectively, significantly outperforming DPP’s F1 scores of 24.95%, 36.20%, and 35.76%.

**Ans. to RQ1:** TrickCatcher shows the best effectiveness in detecting bugs in both human-written and AI-generated plausible programs. The recall, precision, and F1 score achieved by TrickCatcher are  $1.80\times$ ,  $2.65\times$ , and  $1.66\times$  those of the state-of-the-art baseline.

Table 2: (RQ3) Results of ablation study. “PG”, “IG”, and “DT” represent different ways to perform program variant generation, input generation, and differential testing, respectively. “R”, “P”, and “F” represent recall, precision, and F1 score, respectively.

Pattern	PG	IG	DT	k=2			k=4			k=6			k=8			k=10		
				R	P	F	R	P	F	R	P	F	R	P	F	R	P	F
1	Basic	Basic	Basic	.21	.26	.23	.19	.32	.24	.18	.42	.25	.16	.53	.25	.16	.61	.25
2	Filtered	Basic	Basic	.20	.27	.23	.19	.34	.24	.17	.45	.25	.16	.50	.25	.16	.48	.24
3	Filtered	Basic	Ours	.23	.60	.33	.22	.59	.32	.22	.59	.32	.22	.59	.32	.22	.59	.32
4	Ours	Basic	Ours	.22	.48	.30	.24	.47	.32	.24	.47	.32	.25	.47	.32	.25	.47	.32
5	Filtered	Ours	Ours	.26	.76	<b>.38</b>	.26	.76	.38	.26	.76	.38	.26	.76	.38	.26	.76	.38
6	Ours	Ours	Ours	.25	.70	.37	.28	.70	<b>.40</b>	.29	.69	<b>.41</b>	.29	.69	<b>.41</b>	.29	.70	<b>.41</b>

## 6.2 RQ2: Performance on Correct Programs

We further evaluate TrickCatcher and baseline methods on correct programs (i.e., canonical programs provided by the datasets) to assess whether they introduce false positives (FPs). We categorize the FPs into two types: incorrect oracles and invalid inputs. Since APR does not generate test cases, it is excluded from this RQ. Given that EvalPlus provides official checkers for input validity, we focus on this dataset for RQ2.

Figure 5 presents the results. The total number of FPs generated by TrickCatcher is significantly lower (up to 16 $\times$ ) compared to DPP and CHAT. Notably, TrickCatcher produces no FP due to invalid inputs, demonstrating that its generator-based input generation method can effectively ensure valid inputs. In contrast, most FPs from DPP are due to invalid inputs. Additionally, the majority of FPs from CHAT are caused by incorrect oracles, highlighting that LLMs struggle to directly generate accurate test oracles for given test inputs.

**Ans. to RQ2:** TrickCatcher generates up to 16 $\times$  fewer false positives for correct programs compared to state-of-the-art methods.

## 6.3 RQ3: Ablation Study

We conduct an ablation study to evaluate the contributions of different components to TrickCatcher’s performance. Due to the page limit, we focus on TrickyBugs(C++) dataset.

Table 2 presents the results of the ablation study. In the table, “PG”, “IG”, and “DT” refer to different ways for program variant generation, input generation, and differential testing, respectively.

- For program generation, “Basic” generates pro-

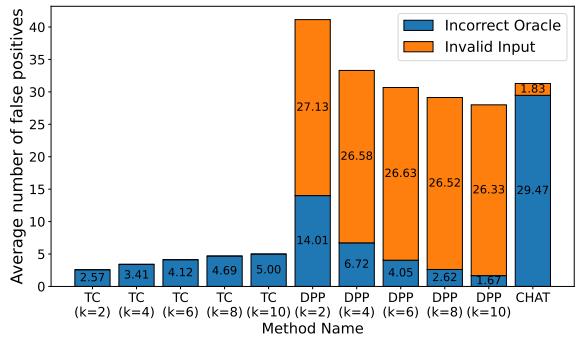


Figure 5: (RQ2) False positives generated by each approach for correct programs. Lower values indicate better performance. TrickCatcher generates significantly fewer false positives compared to the other methods.

gram variants solely based on the specification and does not filter them using the existing test suite. The “Filtered” approach also generates variants in the same way as “Basic” but then filters them using the existing test suite. “Ours” refers to TrickCatcher’s PUT-guided program generation.

- For input generation, the “Basic” approach directly uses the LLM to generate test inputs based on the specification, while “Ours” employs TrickCatcher’s generator-based input generation method.
- For differential testing, “Basic” follows the majority voting rule for determining the test oracle. “Ours” implements TrickCatcher’s diversity-driven differential testing.

Pattern 6 in the table represents the complete TrickCatcher approach.

The results demonstrate the great contribution of each component of TrickCatcher: PUT-guided program generation (by comparing Patterns 3, 5, and 6), generator-based generation (by comparing

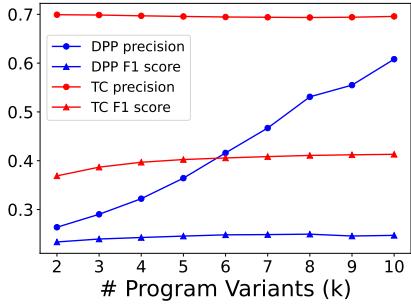
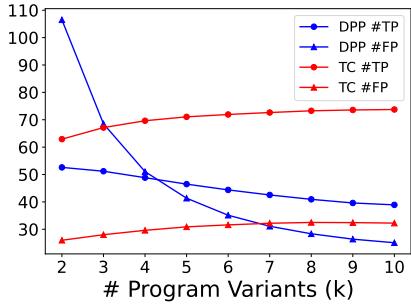


Figure 6: (RQ4) Impact of the number of generated program variants  $k$ . TrickCatcher consistently maintains stable precision and F1 scores.

Patterns 4 and 6), and diversity-driven differential testing (by comparing Patterns 2 and 3).

**Ans. to RQ3:** Each component of TrickCatcher (i.e., PUT-guided program generation, generator-based test generation, and diversity-driven differential testing) contributes to its final performance.

#### 6.4 RQ4: Impact of Program Variant Number

To better understand how the number of program variants  $k$  impacts the effectiveness of TrickCatcher and DPP, We analyze the number of FPs, the number of TPs, precision, and F1 score as  $k$  changes for the TrickyBugs (C++) dataset.

Figure 6 shows the results. We can observe that TrickCatcher’s TP, FP, precision, and F1 score are superior to DPP in most cases. For precision and F1 score, TrickCatcher, even in the worst case, outperforms DPP in the best case. Furthermore, the performance of DPP fluctuates significantly with changing  $k$ , while the performance of TrickCatcher remains consistently stable and excellent, further demonstrating the practicality of TrickCatcher.

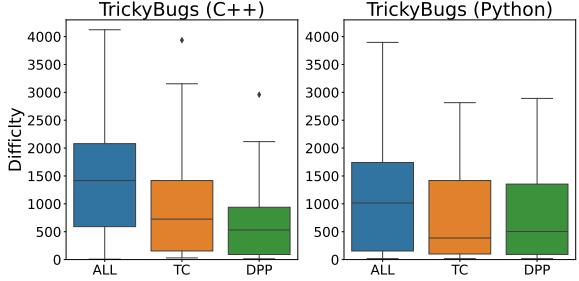


Figure 7: (RQ5) The distribution of the difficulty of all tasks and the tasks where TC and DPP perform well.

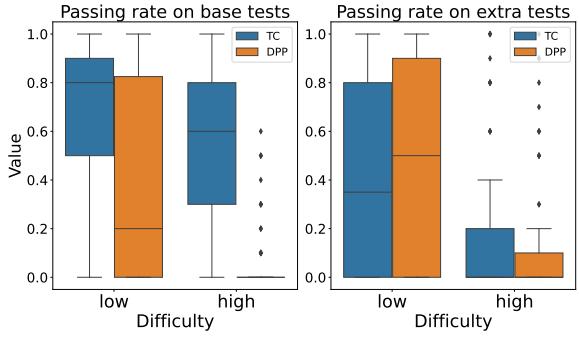


Figure 8: (RQ5) Average passing rate of the generated program variants on base test cases and extra test cases. The data is grouped by difficulty, “low” represents the lower 50%, and “high” represents the higher 50%.

**Ans. to RQ4:** The performance of TrickCatcher remains consistently stable and high with different numbers of program variants.

#### 6.5 RQ5: Impact of Task Difficulty

We further explore the impact of the difficulty of coding tasks on the effectiveness of TrickCatcher. Since the TrickyBugs dataset provides difficulty information, we focus on it here.

Figure 7 shows the results. “ALL” is the difficulty distribution of all tasks in the dataset. The next two box plots represent the difficulty distribution of the tasks where the corresponding method performs well (defined as achieving a precision greater than 0.5). We can find that when the difficulty of the coding task is low, the performance of the two methods is similar (see the right figure). However, for more difficult coding tasks, TrickCatcher outperforms DPP (see the left figure). Figure 8 shows another interesting result that more program variants generated by TrickCatcher pass the base tests, and this difference is more pronounced in high-difficulty tasks. This difference suggests that TrickCatcher introduces fewer new bugs when

generating program variants, thereby improving the final performance.

**Ans. to RQ5:** TrickCatcher demonstrates a more significant improvement over DPP on the PUTs for more difficult coding tasks.

## 7 Discussion

### 7.1 Usefulness of Buggy Program Variants

Here we discuss an interesting finding that **buggy program variants can also contribute to generating true positive test cases**. We refer to the variants that have produced the correct oracle for any true positive test case as *useful variants*. During our evaluation, we find that 23.2% (TrickyBugs) and 15.0% (EvalPlus) of the useful variants are actually buggy (A program variant is buggy if it has ever produced any wrong output that is different from the canonical solution).

The performance comparison between TC and APR in Table 1 also supports the conclusion that buggy variants can be useful, as TrickCatcher achieves better recall than APR in most cases.

The findings imply that there is a certain logical complementarity between these buggy variants and the buggy PUT. So, even if a variant is not entirely correct, it may still contribute to generating a bug-identifying test case. This observation aligns with the points we made in Section 5.4 that the ability of TrickCatcher to detect bugs is not solely derived from LLM-based program repair, TrickCatcher can also leverage buggy variants to generate correct test oracles.

### 7.2 Method Generalization Capability

We also use another model, deepseek-v3, for experiments on EvalPlus dataset to verify the generalization capability of TrickCatcher. The evaluation results are shown in Table 3.

Table 3: Evaluation results (EvalPlus) with different language models.

Model	Recall	Precision	F1 score
deepseek-v3(k=5)	44.26	90.94	59.54
deepseek-v3(k=10)	44.01	90.43	59.21
gpt3.5-turbo(k=5)	35.97	84.12	50.39
gpt3.5-turbo(k=10)	37.14	83.14	51.34

The experimental results demonstrate the generalization capability of TrickCatcher, and we can

also find that the stronger the underlying model, the better the performance.

## 8 Conclusion

We propose TrickCatcher, an LLM-powered test case generation approach for detecting bugs in plausible programs. We evaluate TrickCatcher on both human-written and AI-generated plausible programs. The results show that TrickCatcher achieves up to 1.80 $\times$ , 2.65 $\times$ , and 1.66 $\times$  the recall, precision, and F1 score of the state-of-the-art baseline, respectively. Additionally, the ablation study demonstrates that each component of TrickCatcher contributes to its performance.

## Limitations

The first limitation is that, due to budget constraints, we use two models, gpt-3.5-turbo and deepseek-v3, for evaluation. However, we believe that utilizing more advanced LLMs could further enhance the performance of TrickCatcher. The second limitation is the inherent uncertainty in the behavior of LLMs. To mitigate this, we performed multiple repetitions and averaged the results to ensure a more reliable evaluation. The third limitation concerns the risk of data leakage. However, the TrickyBugs dataset we used was released after gpt-3.5-turbo-0125, and EvalPlus explicitly prohibits its use for training LLMs. Moreover, the poor performance of the three LLM-based baselines further suggests that data leakage is not a main concern in our evaluation.

## Acknowledgements

This research is supported by the National Key R&D Program under Grant No.2023YFB4503801, the National Natural Science Foundation of China under Grant No.62192733, 62192730, and the Major Program (JD) of Hubei Province (No.2023BAA024). Jie M. Zhang is supported by the ITEA Genius and ITEA GreenCode projects, funded by InnovateUK.

## References

- Roberto Baldoni, Emilio Coppa, Daniele Cono D’elia, Camil Demetrescu, and Irene Finocchi. 2018. A survey of symbolic execution techniques. *ACM Computing Surveys*, 51(3):1–39.
- Cristian Cadar, Daniel Dunbar, and Dawson R. Engler. 2008. KLEE: unassisted and automatic generation of

- high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation*, pages 209–224.
- 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. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*.
- Mouxiang Chen, Zhongxin Liu, He Tao, Yusu Hong, David Lo, Xin Xia, and Jianling Sun. 2024a. B4: Towards optimal assessment of plausible code solutions with plausible tests. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, pages 1693–1705.
- Yinghao Chen, Zehao Hu, Chen Zhi, Junxiao Han, Shuiguang Deng, and Jianwei Yin. 2024b. Chatunitest: A framework for llm-based test generation. In *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering*, pages 572–576.
- Gordon Fraser and Andrea Arcuri. 2011. Evosuite: Automatic test suite generation for object-oriented software. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, pages 416–419.
- Alex Gu, Wen-Ding Li, Naman Jain, Theo Olausson, Celine Lee, Koushik Sen, and Armando Solar-Lezama. 2024. The counterfeit conundrum: Can code language models grasp the nuances of their incorrect generations? In *Findings of the Association for Computational Linguistics ACL 2024*, pages 74–117.
- Tsz-On Li, Wenxi Zong, Yibo Wang, Haoye Tian, Ying Wang, Shing-Chi Cheung, and Jeff Kramer. 2023. Nuances are the key: Unlocking ChatGPT to find failure-inducing tests with differential prompting. In *Proceedings of the 38th IEEE/ACM International Conference on Automated Software Engineering*, pages 14–26.
- Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. 2023a. Is your code generated by ChatGPT really correct? Rigorous evaluation of large language models for code generation. In *Proceedings of the Thirty-seventh Conference on Neural Information Processing Systems*.
- Junwei Liu, Kaixin Wang, Yixuan Chen, Xin Peng, Zhenpeng Chen, Lingming Zhang, and Yiling Lou. 2024a. Large language model-based agents for software engineering: A survey. *CoRR*, abs/2409.02977.
- Kaibo Liu, Yudong Han, , Yiyang Liu, Jie M. Zhang, Zhenpeng Chen, Federica Sarro, Gang Huang, and Yun Ma. 2024b. TrickyBugs: A dataset of corner-case bugs in plausible programs. In *Proceedings of the 21st International Conference on Mining Software Repositories*, pages 113–117.
- Kaibo Liu, Yudong Han, Jie M. Zhang, Zhenpeng Chen, Federica Sarro, Mark Harman, Gang Huang, and Yun Ma. 2023b. Who judges the judge: An empirical study on online judge tests. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, page 334–346.
- Zhongxin Liu, Kui Liu, Xin Xia, and Xiaohu Yang. 2023c. Towards more realistic evaluation for neural test oracle generation. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, page 589–600.
- Stephan Lukasczyk and Gordon Fraser. 2022. Pynuin: Automated unit test generation for python. In *Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: Companion Proceedings*, pages 168–172.
- Phil McMinn. 2004. Search-based software test data generation: A survey. *Software Testing, Verification & Reliability*, 14(2):105–156.
- Phil McMinn. 2011. Search-based software testing: Past, present and future. In *Proceedings of the 2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops*, pages 153–163.
- Gabriel Ryan, Siddhartha Jain, Mingyue Shang, Shiqi Wang, Xiaofei Ma, Murali Krishna Ramanathan, and Baishakhi Ray. 2024. Code-aware prompting: A study of coverage-guided test generation in regression setting using LLM. *Proceedings of the ACM on Software Engineering*, 1(FSE):951–971.
- Max Schäfer, Sarah Nadi, Aryaz Eghbali, and Frank Tip. 2024. An empirical evaluation of using large language models for automated unit test generation. *IEEE Transactions on Software Engineering*, 50(1):85–105.
- Florian Tambon, Amin Nikanjam, Le An, Foutse Khomh, and Giuliano Antoniol. 2024. Silent bugs in deep learning frameworks: An empirical study of Keras and TensorFlow. *Empirical Software Engineering*, 29(1):10.
- Zhiqiang Yuan, Mingwei Liu, Shiji Ding, Kaixin Wang, Yixuan Chen, Xin Peng, and Yiling Lou. 2024. Evaluating and improving ChatGPT for unit test generation. *Proceedings of the ACM on Software Engineering*, 1(FSE):1703–1726.

## A Category of Test Cases

Based on the problem definition in Section 3, we define a comprehensive category of test cases:

### 1 Test cases that correctly identify a bug ( $T_c$ ).

For any  $t = (in, out) \in T_c$ , we have  $f(in) = out$  and  $PUT(in) \neq out$ . This is exactly the test case we want, which effectively identifies a bug in PUT.

### 2 Test cases with right test oracles but do not identify any bug ( $T_r$ ).

For any  $t = (in, out) \in T_r$ ,

we have  $f(in) = out$  and  $PUT(in) = out$ . Test cases of this kind are trivial, neither positively nor negatively significant for bug identification.

**③ Test cases with wrong test oracles ( $T_w$ ).** For any  $t = (in, out) \in T_w$ , we have  $f(in) \neq out$ . These test cases are erroneous. They may lead to *false negative*, where  $PUT(in) = out$  but  $PUT(in) \neq f(in)$ . They may also lead to *false positive*, where  $PUT(in) \neq out$  but  $PUT(in) = f(in)$ . False positives are more detrimental than false negatives because false positives can undermine the credibility and practicality of the entire method.

**④ Test cases with invalid input ( $T_{err}$ ).** For any  $t = (in, out) \in T_{err}$ , we have  $in \notin$  input space  $I$ . These test cases are erroneous. For any invalid test input, no test oracle exists, and all program behaviors are undefined. These test cases might also lead to false positives.

Any test case should fall into one of the above four categories.

For convenience, we define two additional types of test cases: We define **passed test cases ( $T_p$ )**. For any  $t = (in, out) \in T_p$ , we have  $PUT(in) = out$ . A passed test case  $t$  could belong to  $T_r$ ,  $T_w$ , or  $T_{err}$ . Then we define **failed test cases ( $T_f$ )**. For any  $t = (in, out) \in T_f$ , we have  $PUT(in) \neq out$ . A failed test case  $t$  could belong to  $T_c$ ,  $T_w$ , or  $T_{err}$ .

We focus on identifying functional bugs within plausible programs; therefore, we do not consider other program behaviors such as timeouts or crashes, which fall outside the scope of our study.

## B Experiment Repetition

To make our evaluation results more reliable, we repeat the experiments multiple times to reduce the impact of randomness from LLMs.

For CHAT, each response of the LLM is a test case. We repeatedly sample 100 test cases and compute the average number of TPs and FPs.

For APR, each response of the LLM is a patch. We repeatedly sample 10 patches and compute an average number of TPs and FPs.

For DPP and TC, the randomness comes from program generation and input generation. We first repeatedly sample 100 inputs and 10 program variants, then keep only the filtered program variants (if the method in the first step is TC or filtered). Next, we use a combinatorial approach to conduct extensive repeated experiments. For example, we have 10 filtered program variants for a PUT and set

the parameter  $k$  (number of generated programs) as 4. Furthermore, we randomly select 4 out of the 10 program variants each round, resulting in a total of  $C_{10}^4 = 210$  rounds of results. For each round, we first compute the average number of TPs and FPs among the 100 generated inputs, getting an average precision and recall for this round. We average the results of the 210 rounds to obtain the final result.