

Nuances are the Key: Unlocking ChatGPT to Find Failure-Inducing Tests with Differential Prompting

Tsz-On Li^{a,b}, Wenxi Zong^c, Yibo Wang^c, Haoye Tian^d, Ying Wang^{c,a*}, Shing-Chi Cheung^{a,b*}, Jeff Kramer^e

^aThe Hong Kong University of Science and Technology, Hong Kong, China

^bGuangzhou HKUST Fok Ying Tung Research Institute, Guangzhou, China

^cNortheastern University, Shenyang, China

^dUniversity of Luxembourg, Luxembourg, Luxembourg

^eImperial College London, London, United Kingdom

{toli,scc}@cse.ust.hk, iamwenxiz@163.com, yibowangcz@outlook.com, haoye.tian@uni.lu,
wangying@swc.neu.edu.cn, j.kramer@imperial.ac.uk

Abstract—Automated detection of software failures is an important but challenging software engineering task. It involves finding in a vast search space the failure-inducing test cases that contain an input triggering the software fault and an oracle asserting the incorrect execution. We are motivated to study how far this outstanding challenge can be solved by recent advances in large language models (LLMs) such as ChatGPT. However, our study reveals that ChatGPT has a relatively low success rate (28.8%) in finding correct failure-inducing test cases for buggy programs. A possible conjecture is that finding failure-inducing test cases requires analyzing the subtle differences (nuances) between the tokens of a program’s correct version and those for its buggy version. When these two versions have similar sets of tokens and attentions, ChatGPT is weak in distinguishing their differences.

We find that ChatGPT can successfully generate failure-inducing test cases when it is guided to focus on the nuances. Our solution is inspired by an interesting observation that ChatGPT could infer the intended functionality of buggy code if it is similar to the correct version. Driven by the inspiration, we develop a novel technique, called Differential Prompting, to effectively find failure-inducing test cases with the help of the compilable code synthesized by the inferred intention. Prompts are constructed based on the nuances between the given version and the synthesized code. We evaluate Differential Prompting on QuixBugs (a popular benchmark of buggy programs) and recent programs published at Codeforces (a popular programming contest portal, which is also an official benchmark of ChatGPT). We compare Differential Prompting with two baselines constructed using conventional ChatGPT prompting and PYNGUIN (the state-of-the-art unit test generation tool for Python programs). Our evaluation results show that for programs of QuixBugs, Differential Prompting can achieve a success rate of 75.0% in finding failure-inducing test cases, outperforming the best baseline by 2.6X. For programs of Codeforces, Differential Prompting’s success rate is 66.7%, outperforming the best baseline by 4.0X.

Index Terms—failure-inducing test cases, large language models, program intention inference, program generation

I. INTRODUCTION

“Testing leads to failure, and failure leads to understanding.”
- Burt Rutan

*Ying Wang and Shing-Chi Cheung are the corresponding authors.

Problem and Challenges. Finding failure-inducing test cases is a main objective in software testing. However, finding such test cases is challenging in practice. First, it is because a developer needs to search for a test input that can trigger a program failure, while the search space of such a test input is vast [1]. Second, the developer needs to construct a test oracle to automatically detect a program failure, while constructing a test oracle is often an undecidable problem [2].

Recent studies [3], [4] have reported the potential of using large language models (LLMs) like ChatGPT [5] for software engineering tasks, such as automated program repair. In this paper, we conduct the first study to investigate the use of ChatGPT in finding failure-inducing test cases for a given program. We do not focus on finding test cases that achieve high code coverage because it is not necessarily correlated with fault detection abilities. In addition, finding test cases whose assertions are incorrectly generated is not useful even if they achieve high code coverage. Our study aims to find test cases with failure-inducing test inputs and correctly specified assertions. However, using ChatGPT to find such test cases is challenging.

We experiment using ChatGPT in a direct manner with both buggy and correct Python programs in QuixBugs [6], which is a common benchmark used by recent works [4], [3] to study the use of LLMs for SE tasks. We prompt ChatGPT in two steps, following a recently proposed prompting convention [3]. For each program, we first ask ChatGPT to determine whether a given program contains bugs. When it replies yes, we ask it to generate a failure-inducing test case. Our experiment yields two findings: first, ChatGPT has a low success rate (28.8%) in finding failure-inducing test cases for buggy programs; second, ChatGPT often considers correct programs buggy and suggests failure-inducing test cases.

A possible explanation of the findings is that ChatGPT is essentially a Transformer (a Deep Learning model architecture commonly adopted by LLM) [5], which is found to be insensitive to nuances (i.e., subtle differences between

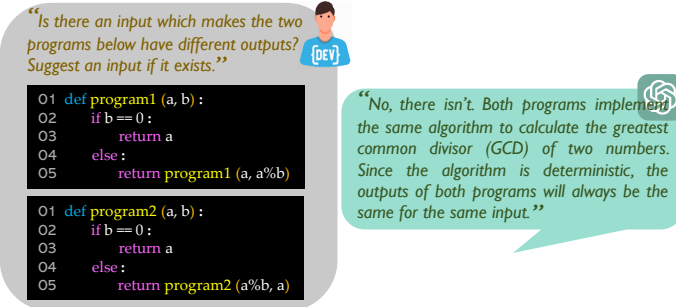


Fig. 1: An illustrative example for ChatGPT’s weakness.

two similar sequences of tokens) [7], [8], [9], [10]. In other words, when a sequence of input tokens varies slightly, LLMs often make similar inferences. This property allows LLMs to tolerate some noise in input tokens. Nevertheless, a bug is essentially a nuance between a buggy program and its fixed version [11], [12], [13]. Hence, an LLM (including ChatGPT) cannot reliably detect the existence of bugs in a program and is therefore weak in finding correct failure-inducing test cases.

Figure 1 shows an illustrative example that consists of two buggy implementations of greatest common divisor. We ask ChatGPT to generate an input that can make the two programs have different outputs. When $a=12$ and $b=20$, the outputs of the two programs differ. However, ChatGPT responds that no input can make the two programs have different outputs. ChatGPT shows its insensitivity to the presence of nuances in this example.

Insights. To tackle the weakness of ChatGPT in finding failure-inducing test cases, we leverage two insights. First, the task of finding a failure-inducing test case for a program can be much facilitated if its intention is known. It enables us to focus our search for failure-inducing test cases on those that exhibit behaviors different from the program intention.

Second, the weakness of ChatGPT at recognizing nuances can facilitate ChatGPT to infer a program’s intention, because such a weakness allows ChatGPT to ignore the presence of a bug (i.e., nuances) when inferring the program’s intention. To validate this insight, we conduct an experiment on QuixBugs by requesting ChatGPT to infer a buggy program’s actual intention. Our experiment result shows that ChatGPT correctly infers the actual intention of 91.0% buggy programs.

Approach. Leveraging these two insights, we propose Differential Prompting, a new paradigm for finding failure-inducing test cases. Differential Prompting divides the task of finding failure-inducing test cases into three sub-tasks: program intention inference, program generation, and differential testing. Specifically, given a program-under-test (PUT), Differential Prompting first asks ChatGPT to infer the intention of a PUT. ChatGPT is then prompted to generate multiple compilable programs that have the same intention as the PUT. Essentially, these programs are alternative implementations of the PUT. Since these programs may also be buggy, we consider only those test cases that deliver consistent results across these programs. Here, we assume that these generated programs are unlikely to commonly suffer from the same bug. Therefore, if one of these test cases induces PUT to deliver a different result, the test case will likely be failure-inducing. In other words, we apply differential testing be-

tween PUT and the programs generated by ChatGPT based on the inferred program intention. By doing so, Differential Prompting bypasses the weakness of ChatGPT at recognizing nuances, while leveraging its capability of inferring program intention. Note that Differential Prompting may find incorrect failure-inducing test cases in the few scenarios when programs generated by ChatGPT commonly suffer from the same bug. We will evaluate this and suggest measures to mitigate this in our evaluation.

Evaluation. We evaluate Differential Prompting on QuixBugs and Codeforces [14]. Codeforces publicly releases all buggy or correct programs submitted by contest participants. We select seven programs that were submitted after September 2021 (the cut-off date of ChatGPT’s training dataset) for evaluation, in order to mitigate the threat of data leakage. We compare Differential Prompting against two baselines: ChatGPT and Pynguin [15] (the state-of-the-art unit test generation tool for Python program). Our evaluation results show that for buggy programs of QuixBugs, the success rate of Differential Prompting in finding failure-inducing test cases is 75.0%, outperforming the best baselines (28.8%) by 2.6X. For correct programs of QuixBugs, Differential Prompting only incorrectly suggests a failure-inducing test case for 5.0% of the programs. In comparison, the two baselines incorrectly suggest a failure-inducing test case for every correct program. For buggy programs of Codeforces that have similar complexity with buggy programs of QuixBugs, Differential Prompting’s success rate is 66.7%, outperforming the best baselines (16.7%) by 4.0X. For the correct programs, Differential Prompting only incorrectly suggests failure-inducing test cases for 1 out of 7 programs, while the baselines incorrectly suggest failure-inducing test cases for all the seven programs.

Contributions. To summarize, this paper makes the following three contributions:

- **Originality:** *To the best of our knowledge, we conducted the first study to investigate ChatGPT’s effectiveness in finding failure-inducing test cases.* Our findings help understand ChatGPT’s limitations in doing so, and inspire future studies in this area.
- **Technique:** *We propose Differential Prompting, a new paradigm for finding failure-inducing test cases.* Differential Prompting unlocks ChatGPT’s power in detecting software failure, by formulating the task of finding failure-inducing test cases into program intention inference, program generation, and differential testing. By doing so, Differential Prompting turns ChatGPT’s weakness into a strength in finding failure-inducing test cases. The evaluation result shows that Differential Prompting notably outperforms the state-of-the-art baselines on QuixBugs and Codeforces.
- **Evaluation:** *We implement Differential Prompting and evaluate it on QuixBugs and Codeforces.* Our evaluation result shows that Differential Prompting’s success rate outperforms the best baseline by 2.6X on QuixBugs and 4.0X on Codeforces. We make our research artifact and experimental data available.

Promising Application Scenarios. We acknowledge that Differential Prompting in this paper works best in finding failure-inducing test cases for simple programs (i.e., programs

that have less than 100 lines of code). However, the improvement that it offers over state-of-the-art approaches is an essential step towards finding failure-inducing test cases for larger software. Larger software is inevitably associated with more sophisticated program intention/requirement, which can be difficult to describe precisely in natural language. In addition, the input search space can be much larger. As a result, failure-inducing test cases are more difficult to find.

Besides, Differential Prompting’s effectiveness in finding failure-inducing test cases also benefits education for software engineering. Specifically, our evaluation result implies that failure-inducing test cases found by Differential Prompting can accurately detect failures of programming beginners’ programs (which are usually simple programs). Previous studies [16], [17], [18] on software engineering education suggest that those failure-inducing test cases can facilitate a programming beginner’s understanding of programming concepts, and help the beginner effectively learn debugging practices. Hence, like many software engineering techniques proposed for programming assignments [19], [20], [21], Differential Prompting can be adopted by beginners for learning programming.

II. PRELIMINARIES

We use a running example in Figure 1 to illustrate the task of finding failure-inducing test cases. In the example, `program1` and `program2` differ in their last return statement, which is `return program1(a,a%b)` for the former and `return program2(a%b,a)` for the latter. Essentially, they swap the order of arguments. This makes the output of `program1` and `program2` different for certain inputs. For instance, when $a = 12$ and $b = 20$, `program1` outputs 12 while `program2` incorrectly outputs 0. However, `program1` intends to implement a greatest common divisor (GCD) program. Hence, given $a = 12$ and $b = 20$, the output of `program1` should be 4.

A test case found by Differential Prompting, ChatGPT or Pinguin [15] can fall into one of the five categories.

- **Correct failure-inducing test case (FT-IA).** An *FT-IA* consists of (1) a failure-revealing test input I that triggers PUT to give an incorrect output, and (2) a correct assertion A that identifies the incorrect output. So, both failure-inducing input and assertion are correctly generated. An example is a test case that executes `program1(12,20)` and asserts its expected output to be 4. It correctly reveals a bug in `program1`. A failing test triggered by an *FT-IA* is considered a *true failure*.
- **Coincidental failure-inducing test case (FT-Ia)** An *FT-Ia* is similar to an *FT-IA* except its assertion a incorrectly specifies the expected output. A failure is coincidentally observed because the incorrect output given by the PUT happens to be different from the incorrectly specified expected output in the assertion. In other words, the failure-inducing input is correctly generated but the assertion is not. An example is a test case that executes `program1(12,20)` but incorrectly asserts its expected output to be a value (e.g., 2) other than 4 (correct output) and 12 (incorrect output of PUT). A failing test triggered by an *FT-Ia* is considered a *coincidental failure*. An *FT-Ia* does not help fault diagnosis and program repair. It rejects correct patches.

- **False failure-inducing test case (FT-ia)** An *FT-ia* reports a false alarm. The test case consists of (1) a non-failure-inducing test input i , and (2) an assertion a that incorrectly asserts the expected output. So, both the failure-inducing input and the assertion are incorrectly generated. An example is a test case that executes `program1(17,0)` and incorrectly asserts its expected output to be 18. A failing test triggered by an *FT-ia* is considered a *false failure*.
- **Passing test (PT).** A *PT* results in a passing test.
- **Illegal argument test case (IT).** An *IT* consists of an illegal test input violating the designated argument type.

Note that a test case with a correct assertion (A) and a non-failure-inducing test input (i) cannot trigger a failure. Such a test case is either *PT* or *IT*. Out of the five categories, an *FT-IA* is considered a **correct failure-inducing test case**, while test cases in the other categories are considered **incorrect failure-inducing test cases**. The correctness criterion allows us to design an evaluation metric (e.g., *success rate*) to compare the effectiveness of Differential Prompting with the baselines (see §IV-A1).

III. METHODOLOGY

Differential Prompting is designed to correctly find failure-inducing test cases for a PUT. Figure 2 shows the workflow of Differential Prompting: it accepts a PUT and outputs either a failure-inducing test case or a message that it cannot find a failure-inducing test case. Differential Prompting consists of two main components: Program Generator (§III-A) and Test Case Generator (§III-B).

A. Program Generator

Program Generator’s objective is to generate multiple “reference versions” of a PUT, each of which provides an alternative implementation of the PUT. However, our experiment on QuixBugs programs finds that only 6.8% of the programs generated by ChatGPT are correct when it is prompted to generate bug-free versions of the QuixBugs programs (see §IV-B2). Therefore, such a strawman approach to generate reference versions does not work well. In the following, we present a more effective mechanism for Program Generator.

1) *Overview of Program Generator:* As pointed out in §I, ChatGPT can be insensitive to code nuances. It is a double-edged sword. On the one hand, it hinders ChatGPT from identifying bugs arising from these nuances. On the other hand, it lets ChatGPT infer the intention of a PUT despite the presence of these bugs. Leveraging this insight, Program Generator divides the task of generating reference versions into two steps. It first leverages ChatGPT to infer the intention of a PUT. Then, it leverages ChatGPT to generate multiple compilable reference versions of the PUT based on the inferred intention. These reference versions will be used by Test Case Generator for differential testing. By doing so, Program Generator no longer relies on the generation of correct patched programs from buggy code, and bypasses the weakness using the strawman approach.

We consider a reference version generated by ChatGPT from the inferred intention is *good* if the reference version does not suffer from the same bug(s) as PUT. It is a necessary condition for Differential Prompting to deduce the correct

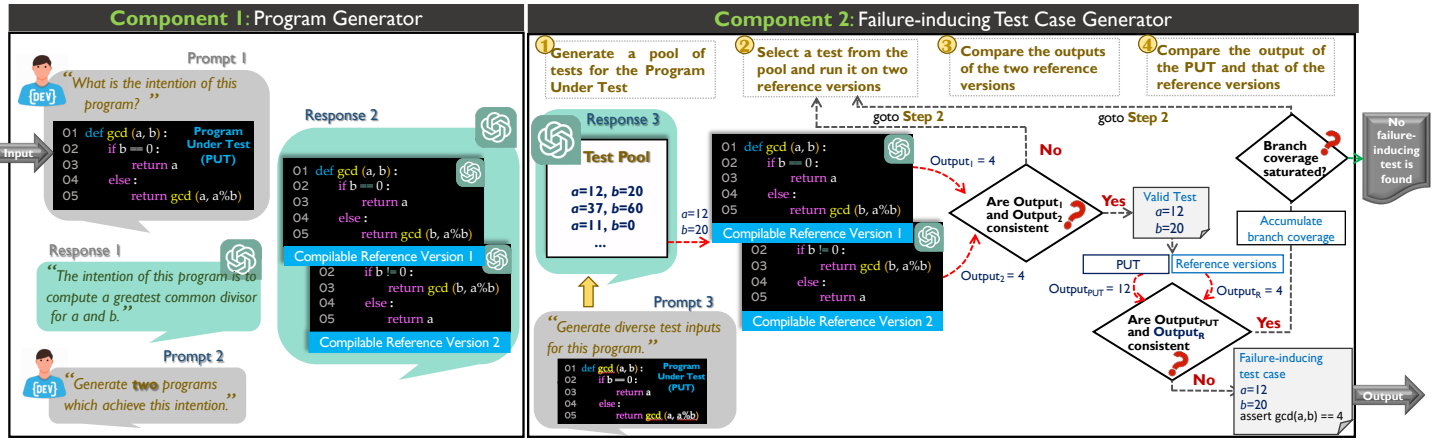


Fig. 2: Workflow of Differential Prompting

expected output of a failure-inducing test case using the reference version. Our experiment results show that 74.6% of the reference versions generated by Program Generator using the inferred intention are good (see §IV-C). The relaxation of the correctness requirement for generated reference versions allows Differential Prompting to successfully find failure-inducing test cases for most program subjects in QuixBugs [6], significantly outperforming the baselines (see §IV-A).

2) *Illustration of Program Generator’s workflow:* Component 1 in Figure 2 illustrates Program Generator’s workflow. Suppose that the buggy `gcd(a,b)` function described in §II is the PUT, Program Generator first requests ChatGPT to infer the intention (as shown in *Prompt 1*). After ChatGPT returns an inferred intention (*Response 1*), Program Generator requests ChatGPT to generate multiple compilable reference versions based on the inferred intention (as shown in *Prompt 2*). ChatGPT then generates the reference versions (named “Reference Version 1” and “Reference Version 2” in *Response 2*). Note that the number of reference versions to generate is a parameter of Differential Prompting. Differential Prompting by default generates two reference versions because two is the minimal number of reference versions needed to validate the correctness of an expected output (more discussion in §III-B2). Our evaluation shows that Differential Prompting can effectively find FT-IA using the default setting (§IV-A).

B. Test Case Generator

A Test Case Generator is designed to find a failure-inducing test case for PUT by conducting differential testing between PUT and the reference versions generated by Program Generator. Test Case Generator works in three steps: *generating test input for PUT* (§III-B3), *inferring expected output using reference versions* (§III-B2) and *applying differential testing* (§III-B3).

1) *Step 1: Generating test input:* To perform differential testing, the first step is to generate test inputs that can trigger diverse behaviors (e.g., branches) of a PUT. Test Case Generator leverages ChatGPT to perform the generation because recent studies [22], [23] reveal that LLMs have the potential to generate more diverse and valid test inputs than conventional approaches (e.g., PYNGUIN). Our experiment results (Figure 3) also show that ChatGPT generates significantly more failing tests than PYNGUIN, and notably fewer illegal test

inputs. Hence, Differential Prompting prompts ChatGPT to generate diverse test inputs instead of relying on Pynguin. Note that Differential Prompting prompts ChatGPT to “generate diverse test input” instead of “generate test inputs that result in different outputs between PUT and reference versions”, because the later prompt requires ChatGPT to identify a nuance between PUT and reference versions. As shown in our illustrative example (§I) and our experimental result (§IV-C), ChatGPT is not effective in telling the differences between two similar pieces of code.

Step 1 works as follows. Given a PUT, Test Case Generator requests ChatGPT to generate diverse tests for the PUT (*Prompt 3* of Figure 2). ChatGPT then returns a set of test inputs (*Response 3*). Note that the number of test inputs returned by ChatGPT may vary across conversations. In the evaluation, we repeat the experiment ten times. Test Case Generator then chooses the test inputs in turn and performs Step 2 (§III-B2) until it finds a failure-inducing test case. It discards a chosen test input if the input leads to an inconsistent output across the reference versions or cannot trigger a failure in PUT.

2) *Step 2: Inferring an expected output:* In this step, Test Case Generator infers the expected output of a PUT with respect to the test input chosen in Step 1. Since reference versions generated by ChatGPT can have bugs, they can induce incorrect failure-inducing tests. Differential Prompting addresses this issue by using only those test inputs that lead to the same output for all reference versions. Here, the strategy assumes that the chances that all reference versions commonly suffer from the same bug are low, i.e., the reference versions are sufficiently diversified. The degree of diversity can be increased by generating more reference versions or enlarging the temperature setting of ChatGPT. Our evaluation (Figure 3 and Figure 4) by using two reference versions shows the effectiveness of this strategy: Differential Prompting has a low probability of returning FT-IA or FT-ia. Essentially, Differential Prompting returns these two types of test cases only when both reference versions commonly have the same bug.

Therefore step 2 works as follows. Test Case Generator first passes the test input in turn to all reference versions generated by Program Generator. Test Case Generator then inspects whether the output returned by each reference version is the same. If so, Test Case Generator regards such an output

as the expected output. Test Case Generator makes at most k attempts (the default value of k is 10). If Differential Prompting cannot find a failure-inducing test case after k attempts, it reports failure-inducing test cases not found.

3) *Step 3: Differential testing*: In this step, Test Case Generator inspects whether the output of PUT and the consistent output of the reference versions are the same. If not, it essentially reveals a potential failure of the PUT. Hence, when Test Case Generator detects an output difference, Test Case Generator constructs a failure-inducing test case with the test input found in Step 1, and the expected output inferred in Step 2, and then reports this failure-inducing test case. Otherwise, the test input is considered non-failure-inducing. So, Test Case Generator rolls back to Step 1 and chooses another test input (§III-B2). Before rolling back to Step 1, Test Case Generator records the lines of code exercised by the current test input, in order to compute the branch coverage accumulated by all test inputs that have been exercised on a PUT. When the branch coverage has reached 100% or saturated, Test Case Generator considers a PUT has been adequately tested. Hence, Test Case Generator stops finding a failure-inducing test case, and reports failure-inducing test cases not found.

IV. EVALUATION

To evaluate Differential Prompting’s effectiveness and usefulness, we study four research questions. For the study of each research question, we first introduce the experiment design (e.g., subject selection, evaluation metrics, and baseline construction), followed by experimental results and discussions.

- **RQ1 (Finding FT-IA for QuixBugs)**: *Can correct failure-inducing test cases for QuixBugs programs be effectively found?*
- **RQ2 (Inferring program intention)**: *Can program intention be effectively inferred?*
- **RQ3 (Generating reference versions)**: *Can reference versions be effectively generated?*
- **RQ4 (Finding FT-IA for Codeforces)**: *Can correct failure-inducing test cases for recent Codeforces programs be effectively found?*

All experiments were conducted on a Linux computer running AMD Ryzen 7 5800 8-Core Processor 3.40 GHz and 16GB RAM. All interactions with ChatGPT, such as sending requests to ChatGPT or receiving responses from ChatGPT, are performed via ChatGPT’s API of version *gpt-3.5-turbo-0301*. We conduct the experiment via ChatGPT’s API instead of its web interface because ChatGPT’s API allows us to explicitly specify the model version (e.g., *gpt-3.5-turbo-0301*), which is useful for result reproduction.

A. RQ1: Finding FTs for QuixBugs

1) *Experiment setup*: We consider two baselines to compare their effectiveness in finding failure-inducing test cases:

- **BASECHATGPT**: The first baseline is to prompt ChatGPT directly for failure-inducing test cases. This baseline prompts ChatGPT in two steps: initially, it checks with ChatGPT if a PUT contains bugs. Upon an affirmative response, it further requests ChatGPT to generate a failure-inducing test case. We refer to this baseline as BASECHATGPT. Like

a recent related study [3], the two-step prompting convention does not assume any knowledge if a given program is buggy, emulating a common real-life situation.

- **PYNGUIN**: The second baseline is PYNGUIN [15], the state-of-the-art unit test generation tool for Python.

Dataset. We evaluate Differential Prompting and the baselines on QuixBugs [6], which consists of 40 pairs of buggy and patched Python programs; each implements a commonly used algorithm, such as *breath-first-search* and *mergesort*. Each buggy program contains one bug. The programs have been adopted by recent works [3], [4] to study the use of LLMs for software engineering tasks. We select all the 80 QuixBugs programs as evaluation subjects because they implement algorithms that are common building blocks of real-life software [4], [6].

Although QuixBugs also consists of Java programs, our evaluation focuses on Python programs for two main reasons. First, Python is one of the most popular programming languages [24]. Second, recent studies [25], [4], [26] show that ChatGPT has potential in performing various software engineering tasks for Python programs. Some of these tasks (e.g., program generation [26]) are closely related to Differential Prompting. Hence, the findings of these studies indicate that Differential Prompting has potential in finding failure-inducing test cases for Python programs. We leave Differential Prompting’s evaluation on Java programs to future works.

Note that, we remove all the code comments or problem descriptions for each program documented by QuixBugs, to avoid their interference with Differential Prompting’s performance of inferring program intentions. In fact, the code comments or problem descriptions of each program are considered the ground truth of the program’s intention in RQ2 (§IV-B).

Comparison. To mitigate experimentation randomness, we repeat the experiment ten times for Differential Prompting and BASECHATGPT, and record the number of failure-inducing test cases in each category found. The number of test cases found each time can be either 0 or 1. For the *success rate* for buggy/correct programs, we compute the ratio of number of correct failure-inducing test cases found by each technique for the buggy/correct programs to the total number of times that the technique has been executed ($400 = 40 \text{ programs} \times 10 \text{ runs}$).

Unlike Differential Prompting and BASECHATGPT which return at most one test case each time, PYNGUIN returns multiple test cases. It is because PYNGUIN is a coverage-guided test generation technique designed to generate enough test cases to achieve code coverage. The current version of PYNGUIN that we can publicly access does not support parameters to limit the number of generated test cases or specify stopping conditions for test generation (e.g., coverage threshold or time budget). Hence, for a fair comparison, we repeat the experiment of applying PYNGUIN to each program ten times. In each experiment, we repeatedly execute PYNGUIN until it generates ten test cases. If PYNGUIN generates more than ten test cases, we select the top ten test cases. The number of FT-IA found by PYNGUIN for a program is calculated as the average number of FT-IA found in the concerned ten experiments. We further round off all the average numbers into integers so

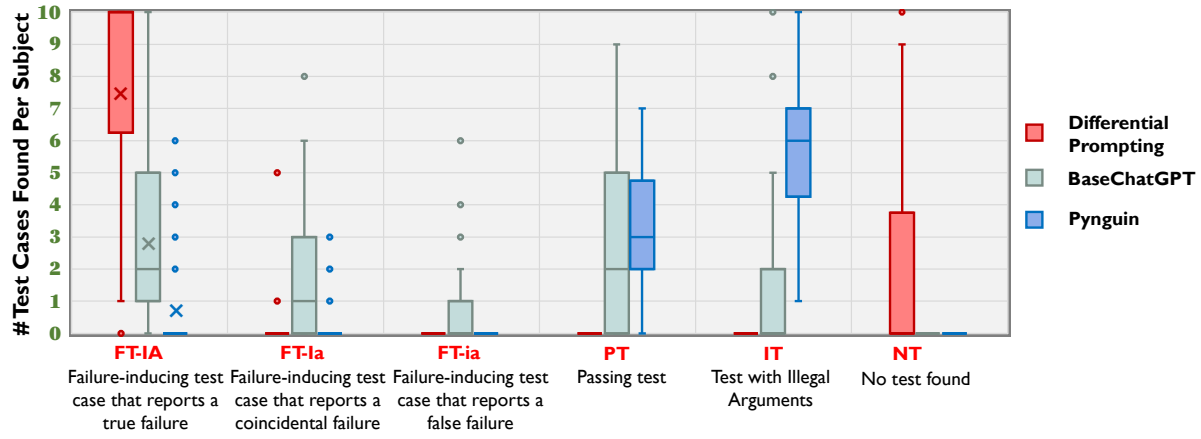


Fig. 3: Effectiveness of Differential Prompting and the baselines in finding failure-inducing test cases for buggy programs of QuixBugs. The vertical axis represents the number of test cases found by Differential Prompting or a baseline for a program subject in ten executions. The cross marks in the FT-IA column indicate the average number of FT-IA found by the three techniques.

that the presentation of PYNGUIN’s performance is consistent with that of Differential Prompting or BASECHATGPT. We calculate the number of other failure-inducing test cases (e.g., *FT-Ia*) found by PYNGUIN similarly. The *success rate* of PYNGUIN is computed in the same way as that of Differential Prompting and BASECHATGPT.

Apart from *success rate*, we propose another evaluation metric called *accuracy* to assess the three techniques’ effectiveness in finding FT-IA. Specifically, *accuracy* is calculated as the total number of FT-IA found by a technique for all targeted subjects (e.g., all buggy programs) divided by the total number of all test cases found by the technique for these subjects. Essentially, *accuracy* is adapted from a popular evaluation metric called *Precision* (i.e., a ratio calculated by dividing the number of true positives with the total number of true positives and false positives). However, since PYNGUIN’s main objective is to generate test cases that achieve high code coverage instead of finding failure-inducing test cases, directly using *Precision* as an evaluation metric for PYNGUIN may lead to confusion. Hence, *accuracy* is proposed to avoid such confusion, while sharing a similar implication with *Precision*.

2) *Results and findings*: Figure 3 compares the effectiveness of the three techniques in finding failure-inducing test cases for buggy programs. The cross marks in the FT-IA column indicate the average number of FT-IA found by the three techniques for all the forty program subjects. Particularly, the average number is calculated as the total number of FT-IA found for the forty subjects divided by forty. Hence, *success rate* can be calculated by dividing the average number of FT-IA by ten (i.e., dividing the total number of FT-IA by four hundred).

Overall, Differential Prompting’s *success rate* is 75.0%, 2.6X as BASECHATGPT (28.8%) and 10.0X as PYNGUIN (7.5%). In terms of *accuracy*, Table II shows that Differential Prompting’s accuracy for buggy code is 98.0%, outperforms the best baseline (28.8%) by 3.4X. This result indicates test cases returned by Differential Prompting have a high probability to be correct failure-inducing test cases (i.e., FT-IA).

Differential Prompting can achieve a significantly higher *success rate* and *accuracy* because Differential Prompting’s

TABLE I: Effectiveness of the three techniques in finding correct failure-inducing test cases (i.e., FT-IA) for programs of QuixBugs. Note that test cases other than FT-IA are considered incorrect failure-inducing test cases.

Buggy programs		
	#programs that a technique finds ten FT-IA	#programs that a technique finds at least one FT-IA
Differential Prompting	22	37
Base-ChatGPT	2	32
Pynguin	0	8
	#programs that a technique finds ten incorrect failure-inducing test cases	#programs that a technique finds at least one incorrect failure-inducing test cases
Differential Prompting	0	2
Base-ChatGPT	8	38
Pynguin	32	40
Correct programs		
	#programs that a technique finds ten incorrect failure-inducing test cases	#programs that a technique finds at least one incorrect failure-inducing test cases
Differential Prompting	0	2
Base-ChatGPT	36	40
Pynguin	40	40

workflow makes use of ChatGPT’s strength and bypasses its weakness. Specifically, §IV-B shows that Differential Prompting correctly infers an intention for 91.0% buggy programs of QuixBugs, and has a success rate of 74.6% in generating reference versions that can reveal the buggy programs’ failures §IV-C.

Table I further delves into the three techniques’ effectiveness in finding FT-IA. Essentially, Differential Prompting finds at least one *FT-IA* in ten executions for 37 out of 40 buggy programs, and finds ten *FT-IA* for 22 out of 40 buggy programs. In contrast, BASECHATGPT (the best baseline) is restricted by ChatGPT’s weakness in identifying nuance, so it finds ten *FT-IA* only for 2 out of 40 buggy programs. For PYNGUIN, it finds only few FT-IA. Indeed, most test cases it finds are *PT* and *IT*, and none of the *PT* consists of failure-inducing test inputs.

TABLE II: Accuracy of the three techniques on QuixBugs programs and Codeforces programs. Accuracy is calculated as the number of FT-IA divided by the number of all test cases (§IV-A1). It has a similar implication as *Precision*.

	Accuracy (buggy programs only)	Accuracy (buggy and correct programs)
QuixBugs programs		
Differential Prompting	98.0%	94.6%
BaseChatGPT	28.8%	14.3%
Pynguin	7.5%	3.8%
Codeforces programs		
Differential Prompting	87.9%	80.1%
BaseChatGPT	7.1%	3.6%
Pynguin	0.0%	0.0%

We analyze the few cases where Differential Prompting fails to find failure-inducing test cases successfully and observe two situations. The *first situation* occurs when Differential Prompting cannot find a failure-inducing test input. For instance, Differential Prompting finds *FT-IA* in only three executions for depth-first-search. In the remaining seven executions, Differential Prompting cannot generate a test input that reaches the buggy branch using ChatGPT. Specifically, to reach the buggy branch, a test input needs to contain a graph of at least one cycle. Without guidance, Differential Prompting (and also BASECHATGPT) does not always generate such a test input. This situation also occurs in the other three buggy programs *detect-cycle*, *quick-sort*, and *topological-ordering*, where Differential Prompting accomplishes a relatively low *success rate*.

The *second situation* occurs when the two reference versions generated by ChatGPT suffer from the same bug. As a result, Differential Prompting can return incorrect failure-inducing test cases (e.g., *lcs-length* and *wrap* return two and one *FT-ia* respectively). §IV-C further delves into this reason and methods for mitigation.

🔑 Finding 1: For buggy programs of *QuixBugs*, Differential Prompting’s success rate is 75.0%, 2.6X as BASECHATGPT (28.8%) and 10.0X as Pynguins (7.5%).
🔑 Implication: Differential Prompting is effective in finding failure-inducing test cases for buggy programs.

Figure 4 compares the effectiveness of Differential Prompting and the baselines on correct programs of *QuixBugs*. The figure shows that Differential Prompting has notably lower frequencies in finding incorrect failure-inducing test cases (i.e., *FT-ia*, *PT*, and *IT*). Table I further looks into the three techniques’ performance in avoiding incorrect failure-inducing test cases for correct programs. Essentially, Differential Prompting returns incorrect failure-inducing test cases for only two programs. These two programs are *lcs-length* and *wrap*, which Differential Prompting also returns incorrect failure-inducing test cases for their buggy version (§IV-B2 discusses these two cases). Meanwhile, the baselines return incorrect failure-inducing test cases for all the forty programs.

Overall, Table II shows that Differential Prompting’s accuracy on both buggy and correct programs (94.6%) is comparable with that on buggy programs only (98.0%). Moreover, Differential Prompting’s accuracy on buggy and correct programs outperforms the best baseline (14.3%) by 6.6X.

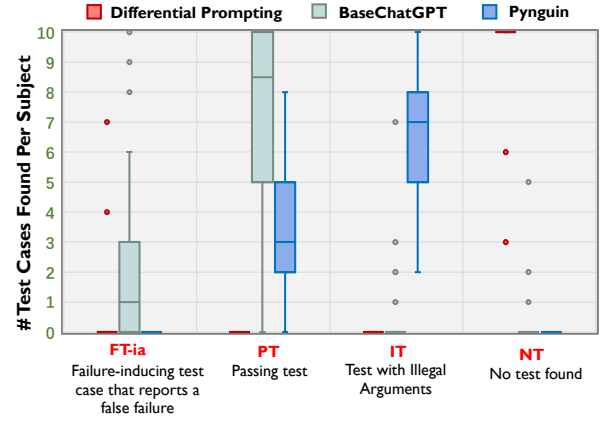


Fig. 4: Effectiveness of Differential Prompting and the baselines in finding failure-inducing test cases for correct programs of *QuixBugs*. The vertical axis represents the number of test cases found by Differential Prompting or a baseline for a program subject in ten executions.

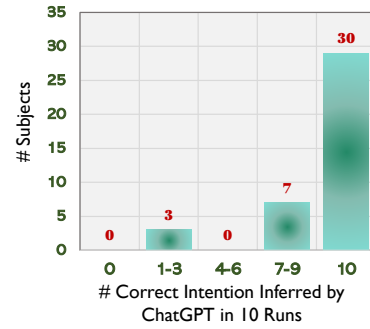


Fig. 5: ChatGPT’s effectiveness in inferring program intention.

🔑 Finding 2: Differential Prompting’s accuracy on buggy and correct programs (94.6%) outperforms the best baseline (14.3%) by 6.6X.
🔑 Implication: Differential Prompting has a high probability of returning correct failure-inducing test cases (i.e., returning *FT-IA* for buggy programs, and returning no failure-inducing test case for correct programs).

B. RQ2: Inferring Program Intention

1) *Experiment setup:* To evaluate the likelihood that ChatGPT is able to infer a program’s intention correctly even if the program is buggy, we conduct a manual analysis. We treat the problem description documented in the comments of each program in *QuixBugs* as the ground truth to assess the correctness of the 400 intentions inferred by ChatGPT in RQ1. Following an open coding procedure [27], two authors who have substantial software development experience separately examine whether each inferred intention is consistent with the documented problem description. Discrepancies between the two authors are discussed until a consensus is reached. Upon completion of the manual inspection, we observed Cohen’s Kappa to be 82.3%, which is considered an “almost perfect agreement” by Landis et al. [28]. We then calculate ChatGPT’s success rate in inferring intentions as the number of correct intentions divided by 400 (i.e., total number of intention returned by ChatGPT in RQ1).



2) *Results and findings:* Figure 5 shows that ChatGPT infers a correct intention for a majority of buggy programs.

Specifically, Differential Prompting correctly infers ten intentions for 30 out of 40 subjects. Overall, Differential Prompting’s success rate in inferring intention is 91.0% (364 out of 400). This result justifies our insight that ChatGPT is good at inferring program intention despite the presence of program bugs.

We further look into the three subjects (`lcs-length`, `wrap` and `next_palindrome`) where Differential Prompting infers only few (1-3) correct intentions. Specifically, the intentions inferred by Differential Prompting for these subjects often describe the actual program semantics of the incorrect implementation. However, these intentions are incorrect, missing important details about the program’s intended functionality.

For instance, `lcs-length` aims to solve the problem of finding the longest common substring of two input strings (“common substrings” are defined as consecutive characters that exist in both input strings). The buggy PUT intends to address this problem using dynamic programming. However, the PUT implements an incorrect dynamic programming algorithm: the implemented statement is `dp[i, j]=dp[i-1, j]+1`, while the correct statement should be `dp[i, j]=dp[i-1, j-1]+1`. Hence, the PUT often finds common characters that are not the longest, if not the shortest.

The intention inferred by ChatGPT correctly points out that the PUT implements a dynamic programming algorithm for finding the longest common characters. However, the intention does not mention whether the characters should be consecutive or not. Hence, the reference versions generated based on this intention often output the longest common characters that are not consecutive (i.e., these reference versions are bad). A similar situation occurs for `wrap` and `next_palindrome`. §IV-C provides further discussion about this case and a possible mitigation strategy.

 **Finding 3:** Differential Prompting’s success rate in inferring intention is 91.0% (364 out of 400).
 **Implication:** ChatGPT can often infer the intention of a program despite the presence of bugs.

C. RQ3: Generating Reference Versions

1) *Experiment setup:* To evaluate Differential Prompting’s effectiveness in generating good reference versions, we study the number of good reference versions generated by Differential Prompting: a reference version is considered *good* if the reference version does not suffer from the same bug(s) as the PUT (discussed in §III-A1).

Specifically, recalled that in RQ1, Differential Prompting has been applied to each buggy program 10 times. Each time, Differential Prompting generates two reference versions (§III-A2). Hence, for each buggy program, Differential Prompting has generated 20 reference versions in total. We consider a reference version of a buggy program to be good if it passes the failure-inducing test cases provided by QuixBugs for the program.

In this RQ, Differential Prompting’s *baseline* is a strawman approach that directly prompts ChatGPT to generate reference versions. The strawman approach prompts ChatGPT in two steps. First, it asks ChatGPT whether a PUT has bugs. Upon

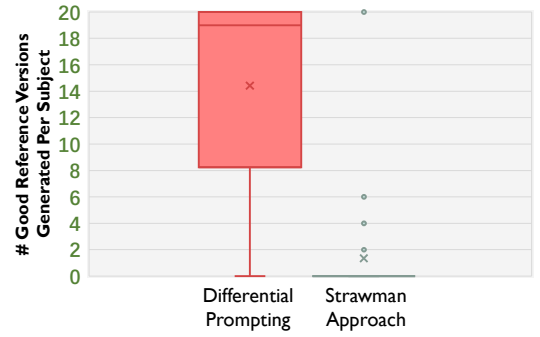


Fig. 6: Effectiveness of Differential Prompting and the baseline in generating good reference versions. The vertical axis represents the number of good reference versions generated by Differential Prompting or the baseline for a subject in ten executions. Note that Differential Prompting or the baseline generates two reference versions in each execution. The cross marks in the figure indicate the average number of good reference versions generated by the two techniques.

affirmative response, it further asks ChatGPT to generate two bug-fixed implementations of the PUT (same as Differential Prompting that in each execution, Differential Prompting also generates two reference versions). This two-step prompting convention emulates a common real-life scenario (similar to BASECHATGPT introduced in RQ1). Following our experiment setup in RQ1, we apply the strawman approach to each program 10 times. Our *evaluation metric* for this RQ is *success rate* in generating good reference versions: the *success rate* is calculated as the number of good reference versions generated by a technique for the forty buggy programs divided by eight hundred (which is the total number of reference versions generated by the technique for the forty buggy programs).



2) *Results and findings:* Figure 6 compares the effectiveness of Differential Prompting and the baseline in generating reference versions. The cross marks in the figure indicate the average number of good reference versions generated by the two techniques for all the forty program subjects. Specifically, the average number is calculated by dividing the total number of good reference versions by forty. Hence, the *success rate* can be calculated by dividing the average number by twenty (i.e., dividing the total number of good reference versions by eight hundred).

Overall, Differential Prompting’s success rate is 74.6%, outperforms the baseline (6.8%) by 11.0X. Besides, Differential Prompting generates 20 good reference versions for 20 subjects, while the baseline generates 20 good reference versions for only two subjects. It is because Differential Prompting generates reference versions through a PUT’s intention (instead of its implementation), and hence bypasses ChatGPT’s weakness in identifying nuances (§IV-B2). In contrast, the strawman approach generates reference versions from a PUT’s implementation, hence it confronts ChatGPT’s limitation in identifying nuances. Figure 6 shows that the strawman approach cannot generate any good reference version for a majority of (34 out of 40) subjects. Specifically, in 89.2% of the executions using the strawman approach, ChatGPT responds with “no bug is found”. In 6.8% of the executions, ChatGPT cannot confirm whether a bug exists in a PUT and

responds with “*More information is required*”. The result is in line with our conjecture that ChatGPT is insensitive to nuances (e.g., bugs) in input tokens (§I).

For the subjects that Differential Prompting only generates a low number (e.g., nearly zero) of good reference versions (e.g., `lcs_length`, `next_palindrome`, `wrap`), we found the main reason is that Differential Prompting has inferred an incorrect intention (§IV-B). For instance, the intention inferred by Differential Prompting for `lcs_length` often misses important details, causing Differential Prompting to generate reference versions that always output an incorrect value. Hence, for `lcs_length`, Differential Prompting generates many (18 out of 20) bad reference versions.

However, we found that by providing Differential Prompting with the correct intention (i.e., the documented description). Differential Prompting has a much higher probability of generating a good reference version. For instance, with a correct intention, the number of good reference versions for `lcs_length` generated by Differential Prompting increases from 2 to 20 out of 20. We also observed a similar increase for `next_palindrome` and `wrap`. This result essentially implies inferring a correct intention is crucial. One possible solution is adopting in-context learning (e.g., few-shot prompting), which has shown great potential in improving ChatGPT’s reasoning capability. [29]

 **Finding 4:** *Differential Prompting’s success rate in generating good reference versions is 74.6%, outperforms the baseline (6.8%) by 11.0X.*
 **Implication:** *Differential Prompting is effective in generating good reference versions.*

D. RQ4: Finding FT-IA for Codeforces

1) *Experiment setup:* To address the validity threat due to data leakage discussed in § IV-E1, we conduct an evaluation on Codeforces programs released after the cutoff date of ChatGPT’s training dataset.

Specifically, we selected programs from a contest for programming beginners (named *Codeforces Round 835*, held on 21, November 2022). *Codeforces Round 835* is the educational contest most recent to the date that ChatGPT was launched (i.e., 30, November 2022). The contest contains seven programming problems of different difficulties. For each programming problem, we choose a buggy version and a correct version based on two criteria. First, the correct program has to be the bug-fixed version of the buggy program, so that we can compare Differential Prompting’s success rate and false positive rate fairly. Second, both programs are implemented in Python. If there is more than one candidate satisfying both criteria, we randomly choose one.

We use the two baselines as adopted for RQ1 (§IV-A): BASECHATGPT and PYNGUIN. We apply Differential Prompting and the baselines to each program 10 times and record the number of failure-inducing test cases found in each category, following the experiment setup in RQ1. We adopt the *success rate* and *accuracy* defined in RQ1 as the evaluation metric of this RQ.

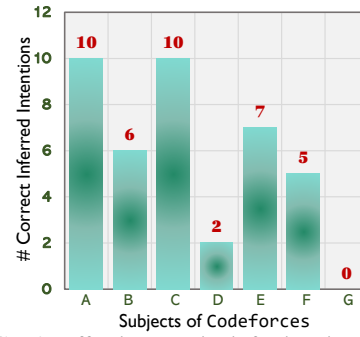


Fig. 7: ChatGPT’s effectiveness in inferring intentions for buggy programs of Codeforces.

2) *Results and findings:* Table III shows that for buggy programs of Codeforces, Differential Prompting’s success rate (41.0%) outperforms the best baseline (7.0%) by 5.9X. In addition, for both buggy and correct programs, Differential Prompting returns incorrect failure-inducing test cases for only one subject (Challenging Valleys), while BASECHATGPT returns incorrect failure-inducing test cases for all seven subjects. Besides, PYNGUIN finds no correct failure-inducing test cases in any executions.

Note that Differential Prompting’s success rate on Codeforces is lower than that on QuixBugs. Table II also shows that Differential Prompting’s accuracy is notably lower on Codeforces than that on QuixBugs. We observe that such discrepancies can be caused by differences in complexities/difficulties between programs of Codeforces and QuixBugs.

Specifically, Table III shows the seven selected Codeforces programs. In fact, Codeforces has assigned a label (a letter from A to G) to each program: the label represents the complexity level of a program. We also present these labels in Table III. Particularly, the program labeled with A is the simplest and the program labeled with G is the most difficult. Essentially, Differential Prompting performs the best (i.e., finds ten FT-IA) for A, and its performance gradually decreases for programs with increasing difficulties. We also find that Differential Prompting’s performance in inferring intentions decreases for programs with increasing difficulties (Figure 7).

We further analyze Codeforces’ programs to understand how program complexity may affect Differential Prompting’s performance. Programs with label D to G are rated with a difficulty level of at least 1000 in Codeforces. According to studies [30], [31] of Codeforces’ rating scheme, programs that are rated 1000 or above focus on applying advanced data structures or algorithmic techniques (e.g., divide-and-conquer, dynamic programming) for solving complex problems.

In comparison, programs with label A to C are rated with a difficulty level of 800, according to the studies [30], [31], they focus on implementing common algorithms (e.g., A implements the computation of medium number). Hence, A to C have similar complexities with QuixBugs’ programs, because QuixBugs also focuses on implementing common algorithms adopted by real-world programs [6], such as quicksort, detect-cycle, reverse-linked-list.

We find that Differential Prompting’s success rate over A to C (66.7%) is not significantly less than its success rate over the QuixBugs programs (75.0%). The results suggest that Differential Prompting is able to find failure-inducing

TABLE III: The effectiveness of Differential Prompting and the baselines in finding failure-inducing test cases for Codeforces.

Buggy programs	Failing test which triggers a true failure (FT-IA)			Failing test which triggers a coincidental failure (FT-Ia)			Failing test which triggers a false failure (FT-ia)			Passing test (PT)			Illegal test input (IT)		
	Differential Prompting	Base-ChatGPT	PYNGUIN	Differential Prompting	Base-ChatGPT	PYNGUIN	Differential Prompting	Base-ChatGPT	PYNGUIN	Differential Prompting	Base-ChatGPT	PYNGUIN	Differential Prompting	Base-ChatGPT	PYNGUIN
Buggy program															
A: Medium Number	10	4	0	0	2	0	0	1	0	0	2	2	0	1	8
B: Atilla's Favorite	6	1	0	0	5	0	0	3	0	0	0	2	0	1	8
C: Advantage	4	0	0	0	5	0	0	4	0	0	1	4	0	0	6
D: Challenging Valleys	3	0	0	0	6	0	4	2	0	0	0	0	0	2	10
E: Binary Inversions	0	0	0	0	0	0	0	5	0	0	0	6	0	4	4
F: Quests	6	0	0	0	2	0	0	8	0	0	0	2	0	0	8
G: SlavicG's Favorite	0	0	0	0	0	0	0	8	0	0	2	5	0	0	5
Average	4.1	0.7	0.0	0.0	2.9	0.0	0.6	4.4	0.0	0.0	0.7	3.0	0.0	1.1	7.0
Correct Program															
A: Medium Number	0	0	0	0	0	0	0	2	0	0	3	2	0	5	8
B: Atilla's Favorite	0	0	0	0	0	0	0	5	0	0	5	3	0	0	7
C: Advantage	0	0	0	0	0	0	0	7	0	0	3	4	0	0	6
D: Challenging Valleys	0	0	0	0	0	0	3	0	0	0	1	0	0	8	10
E: Binary Inversions	0	0	0	0	0	0	0	6	0	0	0	2	0	4	8
F: Quests	0	0	0	0	0	0	0	7	0	0	1	1	0	2	9
G: SlavicG's Favorite	0	0	0	0	0	0	0	3	0	0	1	4	0	5	6
Average	0.0	0.0	0.0	0.0	0.0	0.0	0.4	4.3	0.0	0.0	2.0	2.3	0.0	3.4	7.7


[†] Each cell of the table shows the number of FT-IA, FT-Ia, FT-ia, FT-IA and IT found by Differential Prompting or the baselines for each subject of Codeforces.


[‡] For each subject, Differential Prompting and the baselines are run ten times.

[§] The total number of test cases found by Differential Prompting for a subject can be less than ten, because Differential Prompting may not return any test case in an execution.

test cases for Codeforces programs with similar complexity as QuixBugs programs. Similarly, Differential Prompting's accuracy over A to C is 100% because no incorrect failure-inducing test case has been found. It is comparable with that on QuixBugs programs (over 94.6%, see Table I).

However, Differential Prompting may not work well on complex programs. An interesting future research direction is to explore a reduction methodology, allowing the use of Differential Prompting on the simpler code snippets reduced from a complex program. Experimental results suggest that Differential Prompting is effective to find for those programs the failure-inducing test cases, that the state-of-the-art test generation technique and a strawman approach of using ChatGPT are unlikely able to find.

 **Finding 5:** For selected Codeforces programs that have similar complexity with QuixBugs programs, Differential Prompting's success rate on these Codeforces programs (66.7%) is comparable to its success rate on QuixBugs programs (75.0%).

 **Implication:** Differential Prompting performs comparably on QuixBugs and Codeforces programs with similar complexity.

E. Threat to Validity

Our evaluation can be subject to several validity threats.

1) *Data leakage:* "Data leakage" refers to the problem that an evaluation is conducted on a dataset that has been included in ChatGPT's training dataset. In this case, overfitting may occur and cause evaluation results to be biased. For instance, Differential Prompting's evaluation is conducted on QuixBugs, a public benchmark released in 2017 [6]. It is possible that the benchmark has been used to train ChatGPT whose training cutoff date is September 2021. If so, the performance of both BASECHATGPT and Differential Prompting may be overestimated.

Given the low success rate (28.8%) of BASECHATGPT in finding failure-inducing test cases, the data leakage threat with ChatGPT is likely to be insignificant. In addition, we mitigated the threat with additional experiments on Codeforces

programs, which were created after the training cut-off date of ChatGPT. The performance of Differential Prompting on the Codeforces programs is comparable with that on the QuixBugs programs.

2) *Generalizability:* Evaluation is made only on QuixBugs and Codeforces programs. These programs are simple programs that have less than one hundred lines of code. Hence, there would be a concern that our evaluation result on Differential Prompting may not be generalized to large real-world software. Nonetheless, as pointed out by Weimer et al. [32], large real-world software is often made up of small programs. Hence, our evaluation result reveals Differential Prompting's potential of being applied to large real-world software. In addition, Differential Prompting's enhancement over baselines is a crucial step towards finding failure-inducing test cases for large real-world software.

Another possible validity threat is that our evaluation result based on *gpt-3.5-turbo-0301* may not be generalized to other LLMs such as GPT-4. Nevertheless, we choose *gpt-3.5-turbo-0301* because it is the only LLM providing an API interface that is not subject to limited access. Besides, the technical report [33] released by OpenAI shows that GPT-4, the state-of-the-art LLM, does not outperform ChatGPT (the backbone model of *gpt-3.5-turbo-0301*) for code-related tasks. Hence, our results can provide a useful reference for future related studies.

3) *Reproducibility:* Our evaluation is conducted using *gpt-3.5-turbo-0301*. One possible threat is that the evaluation results presented may not be reproducible after the model has been deprecated in the future. Nevertheless, since Differential Prompting targets at addressing the fundamental limitations of LLMs and conventional approaches (e.g., Pynguin) in finding failure-inducing test cases (§III), Differential Prompting's enhancement over existing approaches is unlikely subject to specific LLMs. Besides, as discussed in §IV-E2, more advanced LLMs (e.g., GPT-4) do not necessarily outperform *gpt-3.5-turbo-0301*. Hence, the results and findings in our paper can still provide useful references even *gpt-3.5-turbo-0301* is deprecated.

V. DISCUSSION AND FUTURE WORK

Compared to baselines, Differential Prompting has a notably higher success rate in finding failure-inducing test cases (§IV-A). We analyze the few cases where Differential Prompting cannot find failure-inducing test cases and find that the bugs are often located at a buggy branch which can be reached only by specific test input values (see §IV-A).

A possible enhancement is to augment the prompting with test coverage guidance. For instance, Differential Prompting can conversationally inform ChatGPT about uncovered statements, and request ChatGPT to generate test inputs to cover those statements. In doing so, ChatGPT can have a higher probability of suggesting failure-inducing test inputs.

Besides augmenting the prompting, Differential Prompting can be readily adapted to Coverage-based Test Generation by 1) leveraging state-of-the-art coverage-based test generation techniques, which generate a set of diverse test cases, and 2) constructing the corresponding test oracles for these test cases using the reference versions.

Apart from test input generation, in this paper, Differential Prompting focuses on finding failure-inducing test cases for relatively simple programs. A future study is to decompose a large program into small programs, and then deduce failure-inducing test cases from the failure-inducing test cases found for these small programs.

VI. RELATED WORK

Related works mainly fall into the three categories below.

A. Finding failure-inducing test inputs

There are many pieces of works that study the problem of test input generation (e.g., coverage-guided test input generation [15], [34], symbolic execution [35] etc). However, not all of them focus on finding failure-inducing test inputs (e.g., Pyguin [15] focuses on generating test inputs that achieve high code coverage). To the best of our knowledge, the most closely-related works would be those target the problem of failing test reproduction [36], [37], [38], failing test augmentation [39], [40], or fuzzing [41], [42], [43]. Failing test reproduction studies the problem of generating failure-inducing test inputs from bug reports. Failing test augmentation studies the problem of generating failure-inducing test inputs based on existing failure-inducing test cases. Failing test augmentation is useful for fault localization. Essentially these works are orthogonal to Differential Prompting, because these works assume a PUT is buggy and additional information (e.g., bug reports or failing tests) is available. In contrast, Differential Prompting is provided with a PUT only (without having knowledge of whether the PUT is buggy or not).

Regarding fuzzing, fuzzing techniques focus on inducing specific types of failures (e.g., crash, security vulnerabilities) that the corresponding test oracles have been pre-defined [41]. In comparison, Differential Prompting is not restricted to the detection of specific failure types. Furthermore, it does not require test oracles to be pre-defined.

B. Addressing Test Oracle problem

Automated test oracle construction is a longstanding challenge. Several paradigms have been proposed to address this challenge, such as metamorphic testing [44], [45], [46],

[47] and differential testing [48], [49], [50]. However, these paradigms often have limited application scenarios [2]. A recent technique SEER [2] aims to address this limitation by proposing a framework to train Deep Learning models for inferring a PUT's behaviors. SEER and Differential Prompting are orthogonal because Differential Prompting focuses on leveraging existing commercial LLMs (e.g., ChatGPT) to address the test oracle problem, instead of training new models (which could be cost-prohibitive for LLMs).

C. Studies of ChatGPT and other LLMs

After ChatGPT has been released, several studies are proposed to study its effectiveness in tackling diverse software engineering problems. Dominik, et al. [3] studies ChatGPT's effectiveness in generating patches. We adopt the prompting convention proposed by this work as the baseline of Differential Prompting in our evaluation (§IV-C2). There are also other works studying ChatGPT's (or other LLMs') capabilities for various software engineering tasks, including bug repair [25], [3], fuzzing [22], code generation [51], [52], code summarization [53], software testing education [54], and vulnerability detection [55], [56]. To the best of our knowledge, no existing works study ChatGPT's (or other LLMs') capability in finding failure-inducing test cases. This paper fills the research gap.

VII. CONCLUSION

In this paper, we propose Differential Prompting, the first paradigm for finding failure-inducing test cases using ChatGPT. The insight is that the program intention inferred by ChatGPT is insensitive to nuances in code. It allows ChatGPT to infer correct intention from a buggy program. With this insight, Differential Prompting finds failure-inducing test cases into three steps: program intention inference, program generation, and differential testing. Our evaluation result shows that Differential Prompting significantly outperforms state-of-the-art baselines in finding failure-inducing test cases.

VIII. DATA AVAILABILITY

We provide a reproduction package at <https://differential-prompting.github.io/> to facilitate future research. The package includes (1) a dataset containing 470 intentions inferred by Differential Prompting, 940 reference versions, and all failure-inducing test cases found by Differential Prompting and the baselines (2) an available tool Differential Prompting, and (3) seven Codeforces programs.

ACKNOWLEDGEMENTS

We would like to thank the anonymous reviewers for their comments and suggestions. We would also like to thank DL library developers for analyzing our reported issues. This work is supported by the National Science Foundation of China (Grant No. 61932021, 62141210), the Hong Kong Research Grant Council/General Research Fund (Grant No. 16205722), the Hong Kong Research Grant Council/Research Impact Fund (Grant No. R5034-18), the Fundamental Research Funds for the Central Universities (Grant No. N2217005), and Open Fund of State Key Lab. for Novel Software Technology, Nanjing University (KFKT2021B01).

REFERENCES

- [1] G. Fraser, M. Staats, P. McMinn, A. Arcuri, and F. Padberg, “Does automated unit test generation really help software testers? a controlled empirical study,” *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 24, no. 4, pp. 1–49, 2015.
- [2] A. R. Ibrahimzada, Y. Varli, D. Tekinoglu, and R. Jabbarvand, “Perfect is the enemy of test oracle,” in *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2022, pp. 70–81.
- [3] D. Sobania, M. Briesch, C. Hanna, and J. Petke, “An analysis of the automatic bug fixing performance of chatgpt,” *arXiv preprint arXiv:2301.08653*, 2023.
- [4] C. S. Xia and L. Zhang, “Conversational automated program repair,” *arXiv preprint arXiv:2301.13246*, 2023.
- [5] “Chatgpt: Optimizing language models for dialogue,” <https://openai.com/blog/chatgpt/>, 2023, accessed: 2023-04-01.
- [6] D. Lin, J. Koppel, A. Chen, and A. Solar-Lezama, “QuixBugs: A multi-lingual program repair benchmark set based on the quixey challenge,” in *Proceedings Companion of the 2017 ACM SIGPLAN international conference on systems, programming, languages, and applications: software for humanity*, 2017, pp. 55–56.
- [7] Z. Zhang, Y. Wu, H. Zhao, Z. Li, S. Zhang, X. Zhou, and X. Zhou, “Semantics-aware bert for language understanding,” in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 34, no. 05, 2020, pp. 9628–9635.
- [8] J. Zhang, W.-C. Chang, H.-F. Yu, and I. Dhillon, “Fast multi-resolution transformer fine-tuning for extreme multi-label text classification,” *Advances in Neural Information Processing Systems*, vol. 34, pp. 7267–7280, 2021.
- [9] T. Jiang, D. Wang, L. Sun, H. Yang, Z. Zhao, and F. Zhuang, “Lightxml: Transformer with dynamic negative sampling for high-performance extreme multi-label text classification,” in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 35, no. 9, 2021, pp. 7987–7994.
- [10] W.-C. Chang, H.-F. Yu, K. Zhong, Y. Yang, and I. S. Dhillon, “Taming pretrained transformers for extreme multi-label text classification,” in *Proceedings of the 26th ACM SIGKDD international conference on knowledge discovery & data mining*, 2020, pp. 3163–3171.
- [11] J. Patra and M. Pradel, “Semantic bug seeding: a learning-based approach for creating realistic bugs,” in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2021, pp. 906–918.
- [12] L. Tan, C. Liu, Z. Li, X. Wang, Y. Zhou, and C. Zhai, “Bug characteristics in open source software,” *Empirical software engineering*, vol. 19, pp. 1665–1705, 2014.
- [13] K. Liu, A. Koyuncu, D. Kim, and T. F. Bissyandé, “Avatar: Fixing semantic bugs with fix patterns of static analysis violations,” in *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2019, pp. 1–12.
- [14] “Codeforces,” <https://codeforces.com/>, 2023, accessed: 2023-04-01.
- [15] S. Lukaszczuk and G. Fraser, “Pynguin: Automated unit test generation for python,” in *Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: Companion Proceedings*, 2022, pp. 168–172.
- [16] E. Lahtinen, K. Ala-Mutka, and H.-M. Järvinen, “A study of the difficulties of novice programmers,” *Acm sigcse bulletin*, vol. 37, no. 3, pp. 14–18, 2005.
- [17] C. Kelleher and R. Pausch, “Lowering the barriers to programming: A taxonomy of programming environments and languages for novice programmers,” *ACM Computing Surveys (CSUR)*, vol. 37, no. 2, pp. 83–137, 2005.
- [18] J. Perretta, A. DeOrio, A. Guha, and J. Bell, “On the use of mutation analysis for evaluating student test suite quality,” in *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2022, pp. 263–275.
- [19] J. Yi, U. Z. Ahmed, A. Karkare, S. H. Tan, and A. Roychoudhury, “A feasibility study of using automated program repair for introductory programming assignments,” in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, 2017, pp. 740–751.
- [20] S. Bhatia, P. Kohli, and R. Singh, “Neuro-symbolic program corrector for introductory programming assignments,” in *Proceedings of the 40th International Conference on Software Engineering*, 2018, pp. 60–70.
- [21] Y. Hu, U. Z. Ahmed, S. Mehtaev, B. Leong, and A. Roychoudhury, “Refactoring based program repair applied to programming assignments,” in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2019, pp. 388–398.
- [22] Y. Deng, C. S. Xia, C. Yang, S. D. Zhang, S. Yang, and L. Zhang, “Large language models are edge-case fuzzers: Testing deep learning libraries via fuzzgpt,” *arXiv preprint arXiv:2304.02014*, 2023.
- [23] M. Schäfer, S. Nadi, A. Eghbali, and F. Tip, “Adaptive test generation using a large language model,” *arXiv preprint arXiv:2302.06527*, 2023.
- [24] R. Widyasari, S. Q. Sim, C. Lok, H. Qi, J. Phan, Q. Tay, C. Tan, F. Wee, J. E. Tan, Y. Yieh *et al.*, “Bugsinpy: a database of existing bugs in python programs to enable controlled testing and debugging studies,” in *Proceedings of the 28th ACM joint meeting on european software engineering conference and symposium on the foundations of software engineering*, 2020, pp. 1556–1560.
- [25] C. S. Xia and L. Zhang, “Keep the conversation going: Fixing 162 out of 337 bugs for \$0.42 each using chatgpt,” *arXiv preprint arXiv:2304.00385*, 2023.
- [26] J. Liu, C. S. Xia, Y. Wang, and L. Zhang, “Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation,” *arXiv preprint arXiv:2305.01210*, 2023.
- [27] J. W. Creswell, *Qualitative Inquiry and Research Design: Choosing Among Five Approaches (3rd Edition)*. SAGE Publications, Inc., 2013.
- [28] J. R. Landis and G. G. Koch, “The measurement of observer agreement for categorical data,” *biometrics*, pp. 159–174, 1977.
- [29] S. Feng and C. Chen, “Prompting is all your need: Automated android bug replay with large language models,” *arXiv preprint arXiv:2306.01987*, 2023.
- [30] A. Ebtekar and P. Liu, “An elo-like system for massive multiplayer competitions,” *arXiv preprint arXiv:2101.00400*, 2021.
- [31] “How to interpret contest ratings,” <https://codeforces.com/blog/entry/68288>, 2023, accessed: 2023-04-01.
- [32] W. Weimer, T. Nguyen, C. Le Goues, and S. Forrest, “Automatically finding patches using genetic programming,” in *2009 IEEE 31st International Conference on Software Engineering*. IEEE, 2009, pp. 364–374.
- [33] OpenAI, “Gpt-4 technical report,” 2023. [Online]. Available: <https://arxiv.org/pdf/2303.08774.pdf>
- [34] G. Fraser and A. Arcuri, “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*, 2011, pp. 416–419.
- [35] R. Baldoni, E. Coppa, D. C. D’elia, C. Demetrescu, and I. Finocchi, “A survey of symbolic execution techniques,” *ACM Computing Surveys (CSUR)*, vol. 51, no. 3, pp. 1–39, 2018.
- [36] W. Jin and A. Orso, “Bugredux: Reproducing field failures for in-house debugging,” in *2012 34th International Conference on Software Engineering (ICSE)*. IEEE, 2012, pp. 474–484.
- [37] M. Soltani, P. Derakhshanfar, A. Panichella, X. Devroey, A. Zaidman, and A. van Deursen, “Single-objective versus multi-objective optimization for evolutionary crash reproduction,” in *Search-Based Software Engineering: 10th International Symposium, SSBSE 2018, Montpellier, France, September 8-9, 2018, Proceedings 10*. Springer, 2018, pp. 325–340.
- [38] M. Soltani, P. Derakhshanfar, X. Devroey, and A. Van Deursen, “A benchmark-based evaluation of search-based crash reproduction,” *Empirical Software Engineering*, vol. 25, pp. 96–138, 2020.
- [39] Z. Zhang, Y. Lei, X. Mao, M. Yan, and X. Xia, “Improving fault localization using model-domain synthesized failing test generation,” in *2022 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2022, pp. 199–210.
- [40] G. An and S. Yoo, “Human-in-the-loop fault localisation using efficient test prioritisation of generated tests,” *arXiv preprint arXiv:2104.06641*, 2021.
- [41] V. J. Manès, H. Han, C. Han, S. K. Cha, M. Egele, E. J. Schwartz, and M. Woo, “The art, science, and engineering of fuzzing: A survey,” *IEEE Transactions on Software Engineering*, vol. 47, no. 11, pp. 2312–2331, 2019.
- [42] S. Saha, L. Sarker, M. Shafiuazzaman, C. Shou, A. Li, G. Sankaran, and T. Bultan, “Rare path guided fuzzing,” in *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2023, pp. 1295–1306.
- [43] J. Liang, Y. Jiang, M. Wang, X. Jiao, Y. Chen, H. Song, and K.-K. R. Choo, “Deepfuzzer: Accelerated deep greybox fuzzing,” *IEEE Transactions on Dependable and Secure Computing*, vol. 18, no. 6, pp. 2675–2688, 2019.
- [44] T. Y. Chen, S. C. Cheung, and S. M. Yiu, “Metamorphic testing: a new approach for generating next test cases,” *arXiv preprint arXiv:2002.12543*, 2020.
- [45] Y. Tian, S. Ma, M. Wen, Y. Liu, S.-C. Cheung, and X. Zhang, “To what extent do dnn-based image classification models make unreliable inferences?” *Empirical Software Engineering*, vol. 26, no. 5, p. 84, 2021.

- [46] J. Cao, M. Li, Y. Li, M. Wen, S.-C. Cheung, and H. Chen, "Semmt: a semantic-based testing approach for machine translation systems," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 31, no. 2, pp. 1–36, 2022.
- [47] S. Segura, G. Fraser, A. B. Sanchez, and A. Ruiz-Cortés, "A survey on metamorphic testing," *IEEE Transactions on software engineering*, vol. 42, no. 9, pp. 805–824, 2016.
- [48] W. M. McKeeman, "Differential testing for software," *Digital Technical Journal*, vol. 10, no. 1, pp. 100–107, 1998.
- [49] R. B. Evans and A. Savoia, "Differential testing: a new approach to change detection," in *The 6th Joint Meeting on European software engineering conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering: Companion Papers*, 2007, pp. 549–552.
- [50] Y. Chen, T. Su, C. Sun, Z. Su, and J. Zhao, "Coverage-directed differential testing of jvm implementations," in *proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2016, pp. 85–99.
- [51] H. Tian, W. Lu, T. O. Li, X. Tang, S.-C. Cheung, J. Klein, and T. F. Bissyandé, "Is chatgpt the ultimate programming assistant—how far is it?" *arXiv preprint arXiv:2304.11938*, 2023.
- [52] P. Vaithilingam, T. Zhang, and E. L. Glassman, "Expectation vs. experience: Evaluating the usability of code generation tools powered by large language models," in *Chi conference on human factors in computing systems extended abstracts*, 2022, pp. 1–7.
- [53] T. Ahmed and P. Devanbu, "Few-shot training llms for project-specific code-summarization," *arXiv preprint arXiv:2207.04237*, 2022.
- [54] S. Jalil, S. Rafi, T. D. LaToza, K. Moran, and W. Lam, "Chatgpt and software testing education: Promises & perils," *arXiv preprint arXiv:2302.03287*, 2023.
- [55] H. Pearce, B. Tan, B. Ahmad, R. Karri, and B. Dolan-Gavitt, "Can openai codex and other large language models help us fix security bugs?" *arXiv preprint arXiv:2112.02125*, 2021.
- [56] C. S. Xia, Y. Wei, and L. Zhang, "Practical program repair in the era of large pre-trained language models," *arXiv preprint arXiv:2210.14179*, 2022.