Impeding LLM-assisted Cheating in Introductory Programming Assignments via Adversarial Perturbation

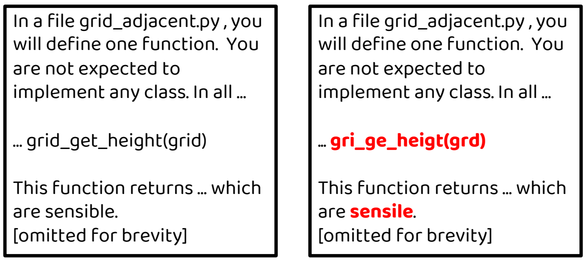
Saiful Islam Salim[*], **Rubin Yuchan Yang**[*], **Alexander Cooper**[*],
Suryashree Ray, **Saumya Debray**, **Sazzadur Rahaman**[†]
University of Arizona, Tucson, AZ, USA
{saifulislam, yuchan0401, alexanderecooper, suryashreeray, debray, sazz}@arizona.edu

Abstract

While Large language model (LLM)-based programming assistants such as CoPilot and ChatGPT can help improve the productivity of professional software developers, they can also facilitate cheating in introductory computer programming courses. Assuming instructors have limited control over the industrial-strength models, this paper investigates the baseline performance of 5 widely used LLMs on a collection of introductory programming problems, examines adversarial perturbations to degrade their performance, and describes the results of a user study aimed at understanding the efficacy of such perturbations in hindering actual code generation for introductory programming assignments. The user study suggests that *i)* perturbations combinedly reduced the average correctness score by 77%, *ii)* the drop in correctness caused by these perturbations was affected based on their detectability.

1 Introduction

Large Language Model (LLM)-based tools such as ChatGPT (OpenAI, 2024) have demonstrated an impressive ability to create high-quality code given simple prompts and have the potential for significant impact on software development (Barke et al., 2023). While there are ongoing efforts to incorporate such tools into computer science (CS) education (Jacques, 2023), integrating new technologies into educational curricula can take a long time (Hembree and Dessart, 1986; Koh and Daniel, 2022). Meanwhile, existing CS curricula are under the threat of LLM-assisted cheating and require immediate attention (Finnie-Ansley et al., 2023, 2022).



In a file grid_adjacent.py, you will define one function. You are not expected to implement any class. In all ...

... grid_get_height(grid)

This function returns ... which are sensible.
[omitted for brevity]

(a) Original prompt

In a file grid_adjacent.py, you will define one function. You are not expected to implement any class. In all ...

... **gri_ge_heigt(grd)**

This function returns ... which are **sensile**.
[omitted for brevity]

(b) Perturbed prompt

Figure 1: Removal of 5 characters from an assignment prompt caused correctness scores of the generated solutions to drop from 100% to 0% in CodeRL, Code Llama, GPT-3.5, and GitHub Copilot. For Mistral, it dropped from 33.33% to 0%.

Given that educators have little direct control over the capabilities of industrial-strength LLMs, two possible directions towards addressing this threat are (i) to detect and penalize LLM-assisted cheating; and (ii) to modify problem statements to impede LLM-assisted cheating. The first approach is problematic because it can be difficult to determine reliably whether some given content is LLM-generated or not (Hoq et al., 2023; Orenstrakh et al., 2023), and both false positives and false negatives are possible. In this paper, we explore the second option and ask the following question: *How can instructors modify assignment prompts to make them less amenable to LLM-based solutions without impacting their understandability to students?*

While there has been some work on the impact of adversarial prompts on LLMs (Wang et al., 2023a; Liu et al., 2023a), we are not aware of any research investigating adversarial strategies for impeding LLM-assisted cheating in a Blackbox setting in an academic context. To systematically study the problem, we break it into the following three steps:

Step 1. Measure the accuracy of LLMs on introductory CS programming assignments, as introductory assignments are at imminent risk (Finnie-Ansley et al., 2023).

Step 2. Develop adversarial techniques to perturb programming assignment prompts and ana-

[*] Authors contributed equally
[†] Corresponding author

Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing, pages 445–463
November 12-16, 2024 ©2024 Association for Computational Linguistics

lyze their impact on the quality of LLM-generated solutions to those problems.

Step 3. Run a user study to understand the potential for such perturbation techniques in impeding *actual* LLM-assisted cheating, focusing in particular on whether students can detect and reverse such perturbations.

An overview of these steps is presented in Figure 2. To measure the accuracy of LLM-generated code, we use the same test inputs used to evaluate student submissions. To modify problem statements in a Blackbox setting, we design a set of perturbation techniques that are informed by existing literature on adversarial perturbation (Bielik and Vechev, 2020; Rauber et al., 2017; Wang et al., 2021b; Zhao et al., 2023). We use SHAP (Lundberg and Lee, 2017) with a *surrogate* model to guide the perturbation for better *efficacy* vs. modification tradeoff. We define *efficacy* (Definition 1) for a perturbation technique to quantify the portion of lowering the LLM accuracy. To ethically conduct the user study in Step 3, we select the study group from students who have already taken the courses corresponding to the assignments used for the study.

Our findings suggest that existing LLMs generally struggle to solve assignments requiring interactions across multiple functions and classes. Our evaluation of different perturbation techniques shows a high overall success rate, causing degradation of more than 85% of the assignments for all five models (example in Figure 1). We find that high variations in solution generations strongly correlate with high success rates. Our user study with undergraduates shows that the average efficacy dropped from 15.43% to 15% when perturbations were noticed. It also suggests that subtle perturbations, i.e., substituting tokens or removing/replacing characters, when unnoticed, are likely to retain high efficacy in impeding actual solution generation. Additionally, the *detectability* of a *high-change* perturbation might not imply *reversion*. The implication is that under perturbations, students have to check and modify LLM solutions rather than adopt them unchanged – instructors can use these perturbations when preparing homework problems to reduce cases where students do not learn but use ChatGPT as is.

2 Measuring LLM Performance (Step 1)

The goal of this evaluation is to answer the following question: *How do LLMs perform on our corpus of programming assignment problems? What problems are more amenable to LLM-assisted cheating?*

2.1 Methodology

Dataset Selection and Preparation. For this study, we select programming assignments from the first two CS courses (CS1 and CS2) at the University of Arizona. These courses offer problem-solving-oriented Python programming assignments focusing on basic control structures, data structures, and algorithms (Appendix A and B). The assignments were designed by the instructors from the ground up, although we acknowledge that variants of the assignments may exist elsewhere, and previous students of the courses could have uploaded the assignments to the internet. In total, we select a set of 58 programming assignments (30 from CS1 and 28 from CS2). We discard 4 graphical user interface-based assignments from CS1, as creating test cases to check their correctness would require non-trivial efforts. Next, we divide each assignment into multiple tasks, as one assignment can contain multiple problems, and categorize them into two types: *short problems*, which require students to implement a single clearly-specified function or class; and *long problems*, which are more complex and which either require students to implement multiple functions or classes that depend on each other, or else leave the required number of functions or classes unspecified. Our corpus contains a total of 84 short problems (20 from CS1 and 64 from CS2) and 22 long problems (10 from CS1 and 12 from CS2). Examples of short and long problems are shown in Figure 4 in Appendix C. We decide not to select any programming assignments from an open dataset for several reasons. Firstly, the evaluation of open datasets might hinder the generalizability of our findings, e.g., performance on open datasets might significantly vary from the closed one where problems were curated from the ground up. Secondly, to evaluate the proposed approaches using our methodology, it is essential to have problems with accurate and reliable solutions and test cases to grade them accurately. However, we did not find any such datasets that meet this requirement.

Creating Test Oracle. We create *test oracles* to check *correctness scores* of a given assignment solution. Given a solution, a test Oracle script

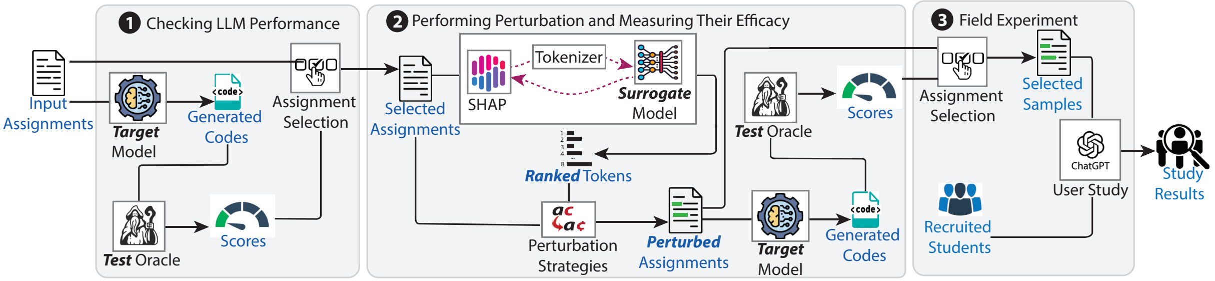


Figure 2: Overview of our study, which is conducted in three steps. Here, boxed elements indicate processing units, and unboxed elements represent input/output data. We used solid *arrows* through processing units to connect inputs to their corresponding outputs.

runs a predefined set of test cases and outputs the percentage of test cases passed by the solution. To build these scripts, we reuse the test cases obtained from the instructor. We form two groups among the authors of this paper to create and validate these test oracles. One group creates the scripts for a selected assignment set, and another validates them.

Model Selection. We consider five LLMs for this study: GPT-3.5 (OpenAI, 2022), GitHub Copilot (GitHub, 2021), Mistral (Mistral AI team, 2024), Code Llama (Rozière et al., 2023) and CodeRL (Le et al., 2022). GPT-3.5 is used behind ChatGPT, and Mistral-Large is used behind Mistral AI chat. GitHub Copilot is an IDE (e.g., JetBrains IDEs, Visual Studio, etc.) plugin developed by GitHub that is powered by OpenAI's Codex model. We select these five models for their availability to *fresh* CS students. We included Code Llama and CodeRL for their wide accessibility. The details of our code generation methods and the model versions and parameters are described in Appendix D; The most important point here is that we set any relevant parameters to values that produce the best possible solutions, upload the problem prompt into the LLM, and evaluate the solutions generated.

2.2 Results: LLM performance

We use all the short (84) and long (22) problems to evaluate the performance of the LLMs considered in our assignment corpus. For a given set of assignments, we define an LLM's performance as the average correctness scores of the corresponding solutions it generates. We generate correctness scores (the portion of the test cases that pass) with our test oracles.

Performance on CS1 Problems. The LLMs we test do not generate completely correct solutions to any of the problems in our CS1 problem set. For two short and 5 long problems, GPT-3.5 refuses to generate any solutions due to triggering academic integrity safeguards. We discuss other possible reasons for this somewhat surprising result in Section 2.3.

Table 1: LLMs' performance on CS2 problems.

Model	Short (64)			Long (12)		
	Mean	Min (Count)	Max (Count)	Mean	Min (Count)	Max (Count)
CodeRL	12.47	0 (48)	100 (3)	0.0	0 (12)	0 (12)
Code Llama	16.07	0 (49)	100 (5)	0.83	0 (11)	100 (1)
Mistral	50.09	0 (26)	100 (23)	25.31	0 (7)	100 (1)
GPT-3.5	41.60	0 (30)	100 (17)	8.33	0 (11)	100 (1)
GitHub Copilot	51.47	0 (26)	100 (24)	26.99	0 (6)	100 (2)

Performance on CS2 Problems. The performance of the LLMs on our CS2 problem set is shown in Table 1. By and large, they perform better than on the CS1 problems. CodeRL has the worst performance of the five LLMs tested: while it can construct correct solutions for some of the short problems with an average score of 12.5% for the short problems, it fails to solve any of the long problems. GPT-3.5 does somewhat better, scoring 41.6% for the short problems and 8.3% for the long problems. While Mistral's performance was closer, GitHub Copilot had the best performance, with an average score of 51.5% for the short problems and 27% for the long problems.

Finding 1: All five LLMs fail to solve CS1 problems. For CS2, GitHub Copilot performed best, with an average score of 51.5% for short and 27% for long assignments.

2.3 Discussion on the Findings

The LLMs' lack of success with CS1 problems is unexpected. Possible reasons for this include: (1) many of them are very specific problems unlikely to be of sufficient general interest to show up in code repositories and thereby appear in LLM training sets, providing a challenge for the LLMs to match the output required by the test oracles exactly; (2)

information relevant to some of the problems is provided graphically (60% CS1 problems), sometimes in the form of ASCII art (Figure 5), which was difficult for the LLMs to process; and (3) assignments are often very specific regarding names of input/output files, classes, methods, etc., and the LLMs had trouble matching these specifics. These results are at odds with other research that suggests that LLMs can be effective in solving introductory programming problems (Finnie-Ansley et al., 2022, 2023). Possible reasons for this difference include: (1) differences in the problems used in different studies, given that there is no consensus on what the specific content of CS1 and CS2 courses ought to be (Hertz, 2010); and (2) methodological differences between studies, e.g., Finnie-Ansley *et al.* manually repaired minor errors in the LLM-generated solutions (Finnie-Ansley et al., 2022) while we did not. Although the LLMs do not generate correct solutions for any of the CS1 problems, in some cases, they generate code that is *close to correct* and could potentially be massaged to a correct solution by a student.

For the CS2 problems, there is a noticeable difference between LLM performance on short problems, which involve creating a single clearly specified function or class, and long problems, which are more complex and involve interactions between multiple functions or classes. All of the LLMs generate correct solutions for some short problems but fail to generate correct solutions for others; while CodeRL fails to generate any correct solutions for any of the long problems. While Code Llama struggled too – GPT-3.5, Mistral and GitHub Copilot were able to generate correct solutions for some of the long problems. Once again, for some of the problems, the LLM-generated code is close to correct, and students could potentially massage them manually into working solutions.

3 Exploring Perturbations (Step 2)

In this section, we explore the following research question: *How can we leverage black-box adversarial perturbation techniques to impede LLM-assisted solution generation?* Towards that end, following existing literature, we design several perturbation techniques and measure their efficacy on the assignments that LLMs solved with non-zero correctness scores. For a given perturbation technique, we define its efficacy as follows.

Definition 1 (Efficacy) *The efficacy of a perturbation technique for a given assignment is the reduction of the LLM's correctness score from the base correctness score on the assignment.*

$$\mathit{Efficacy} = max\left\{0,\ 100 \times \frac{S_{no\_prtrb} - S_{prtrb}}{S_{no\_prtrb}}\right\}$$

$$\mathit{where,}$$

$$S_{no\_prtrb} = \mathit{Correctness\ with\ no\ perturbation}$$
$$S_{prtrb} = \mathit{Correctness\ with\ perturbation}$$

Given the same amount of drops in the correctness score, our efficacy favors the lower correctness score after perturbation. This is because, for example, a drop of 30% from 70% is more favorable than a drop of 30% from 100%, as the former has a more drastic impact on the overall grade.

3.1 Perturbation Methodology

We design ten perturbation techniques under two broad categories, *core* and *exploratory*.

Core perturbations. Under this category, we design seven principled techniques with four end-to-end automated perturbation strategies, *i)* synonym substitution, *ii)* rephrasing sentences, *iii)* replacing characters with *Unicode* lookalikes, and *iv)* removing contents. We apply these strategies to different perturbation units, i.e., characters, tokens, words, and sentences. Perturbation units indicate the unit of changes we make at once. Inspired by explainability-guided adversarial sample generation literature (Sun et al., 2023; Rosenberg et al., 2020), we use SHAP (SHapley Additive exPlanations) (Lundberg and Lee, 2017) with CodeRL as the *surrogate* model to select candidate units for perturbations. Specifically, we use Shapley values to compute the top-ranked tokens for perturbation. For example, for *Character (remove)* perturbation, we remove a random character from each token to generate one variant; for *Token (remove)* perturbation, we remove all 5 tokens to generate one variant, and for the synonym morphs, we may have many synonyms for one token, and generate many variants. For *Token (unicode)* perturbation, we replace all 5 tokens with Unicode characters to generate one variant. For example, we replaced *a*, *c*, and *y* with *à*, *ċ*, and *ý*, respectively. We use the token rank for all the other perturbation units except for sentences. We rank the sentences by accumulating the Shapley values of the tokens corresponding to a given sentence for sentence perturbations. We add a detailed description of each technique in the Appendix E.

Exploratory perturbations. We design three additional techniques to explore the potential of two different insights. For example, existing studies show evidence that LLMs are prone to memorizing training data (Zhang et al., 2021; Carlini et al., 2021, 2023). Thus, these models are highly sensitive to input variations (Zhang et al., 2022; Jin et al., 2022; Reynolds and McDonell, 2021). Under this hypothesis, replacing specific tokens with random strings may significantly influence performance, as such substitution may alter the context (Shi et al., 2023; Liu et al., 2023b; Wang et al., 2021b). We design a new exploratory perturbation technique to leverage this insight. Under this technique, we tweak assignments by replacing file names, function names, and class names specified in the problem statement with random words, where these names are discovered manually. Another example is that to understand the resiliency of LLMs on Unicode lookalikes (Shetty et al., 2018; Boucher et al., 2022), we create a mechanism to replace all possible characters with Unicode lookalikes in the *entire* assignment statement.

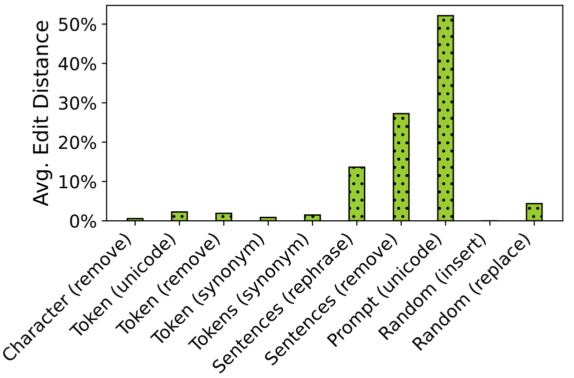


Figure 3: The average changes caused by the perturbation techniques are calculated as the edit distance between the original and the perturbed assignments.

3.2 Results: Perturbation Performance

We measure the performance of our perturbation techniques on the assignments that LLMs solved with non-zero correctness scores.

Perturbation Efficacy. Table 2 depicts the efficacy of all our perturbations. All the perturbations combined cause performance degradation in all five models for most of the assignments we tested. Combined perturbation efficacy is the average efficacy of the best perturbation technique for each problem, i.e.,

$$\text{Combined Efficacy} = \frac{1}{n}\sum_{i=1}^{n}\max\{E_i\}, \text{where,}$$

- n is the total number of problems,
- E_i is the list of efficacy scores of all the perturbation techniques on the i-th problem

The performance is mostly dictated by "remove sentence" and followed by "assignment-wide substitution with Unicodes" perturbations. However, the average *edit distance* for these two techniques is much higher, making them riskier for *detection* (Figure 3), which we discuss next.

Changes in the original prompt. A higher proportion of changes caused by a perturbation technique risks both understandability and detectability. We use the edit distance between the original and perturbed assignment statements to quantify the changes for a given perturbation technique. Note that edit distance is not the ideal method to capture the drifts (if any) caused by Unicode replacements (visual) and synonyms (conceptual); However, it gives a picture of how much the perturbed prompt was altered from the original one. Figure 3 depicts the average edit distance of the perturbation techniques on the assignments with positive efficacy (i.e., causing performance degradation). Except for sentence and prompt-wide perturbations, all the other techniques require a small (<5%) amount of perturbation to the problem statements. This is because they are performed on a small portion of characters or tokens, making them less expensive.

Finding 2: The combination of all the perturbations covers more than 90% of the problems with efficacy >80% for all five models. *High-change* perturbations have high efficacy.

Why perturbations failed? To understand why our perturbation techniques may have failed, we study the two sets of assignments where they succeeded and failed. Under the succeeded category, we select assignments where the average efficacy was high (greater than 90) for at least half of the perturbation techniques. For failed category, we select assignments with efficacy 0 for all the techniques. Next, we randomly select 10 samples for each category and study the *variety* in the generated solutions by the LLMs under various perturbation techniques. For a given assignment, we measure variety by directly comparing all the solutions and counting unique variations. We observe that the

Table 2: Average efficacy of the perturbation techniques. All the perturbations combined caused performance degradation for a significant portion of assignments, which was dictated by "`Sentence (remove)`" and "`Prompt (unicode)`" perturbations.

	CodeRL		Code Llama		Mistral		GPT-3.5		GitHub Copilot	
Perturbations	Problem Count (%)	Avg. Efficacy	Problem Count (%)	Avg. Efficacy	Problem Count (%)	Avg. Efficacy	Problem Count (%)	Avg. Efficacy	Problem Count (%)	Avg. Efficacy
`Character (remove)`	31.25	7.81	50.0	12.19	32.56	24.03	40.0	22.4	25.0	25.17
`Token (unicode)`	43.75	10.94	50.0	12.5	20.93	25.27	34.29	18.49	11.36	14.78
`Token (remove)`	25.0	6.25	56.25	20.61	20.93	18.07	37.14	17.84	34.09	43.79
`Token (synonym)`	56.25	7.65	81.25	16.57	39.53	30.56	42.86	23.81	38.64	26.83
`Tokens (synonym)`	56.25	9.17	87.5	17.73	44.19	29.25	45.71	20.95	34.09	35.1
`Sentences (rephrase)`	75.0	11.85	87.5	18.05	23.26	9.28	51.43	17.36	22.73	21.92
`Sentences (remove)`	**93.75**	14.07	68.75	15.64	**90.7**	42.98	**88.57**	30.71	**79.55**	**60.94**
`Prompt (unicode)`	**93.75**	**23.44**	**100**	**31.77**	79.07	**86.2**	54.29	**33.23**	43.18	47.36
`Random (insert)`	6.25	1.56	50	17.71	0.0	0.0	11.43	5.47	15.9	17.32
`Random (replace)`	37.5	9.11	**100**	**31.77**	**90.7**	87.86	25.71	18.68	13.64	9.11
`Combined`	**93.75**	**100**	**100**	**100**	**100**	**100**	**97.14**	**91.21**	**90.91**	**80.03**

average number of unique variations per problem is 13.9 and 26.0 for problems where perturbation failed and succeeded, respectively. To determine the uniqueness of solutions, we use AST similarity. Comparison of the ASTs of the codes that are the same except for different variable names gets a similarity score of 100, and formatting differences between solutions will be ignored. We use a threshold of 90 when determining if a program is unique.

Finding 3: High variations in generated solutions strongly correlate with high success rates for a given perturbation technique.

4 Field Experiment (Step 3)

In this step, we aim to understand how students would detect and reverse our perturbations. This would provide valuable insights into the potential of the perturbation techniques for impeding actual LLM-assisted cheating.

4.1 Methodology

User Study Design. We recruited 30 undergraduate students who had previously completed CS1 and CS2 courses from the same university to participate in this IRB-approved user study. Each participant was awarded $20 for their participation. During this study, each student was explicitly asked to use ChatGPT to solve 3 assignments over one week and submit the entire chat history in a post-study survey. After the experimentation, we asked the participants to submit their chat history with ChatGPT and observed that all of the participants used ChatGPT-3.5, except for one who used the ChatGPT-4.0 version. We discarded the data from that user.

The details of specific instructions to the students are added in Appendix G.5. We assign each assignment-perturbation pair to at least three participants to cover redundancy and diversity. *This includes no perturbation cases, too, which indicates the base performance.* Our post-study survey also asks whether students noticed anything "unusual" in the assignment description, how they validated solutions, etc. (details in Table 9). Note that for ethical reasons, we chose to run the study on students who already took the courses (Demographic information in Table 8). We discuss its impact on the outcome in Section 8.

Problem Selection. For this study, we select assignments for which the efficacy score for at least one perturbation was 80 on GPT-3.5, which powers ChatGPT. We chose 6 assignments with at least 3 perturbed versions, from this initial list, under 3 different techniques. Table 3 shows the problem and perturbation technique pairs selected for the user study. `Prompt (Original)` indicates prompt with no perturbation. We recognize that removal of content (i.e., characters, tokens, etc.) from the assignment text will be easily detected by students. To remedy this, we replace the removed content with images of the characters that were removed in an attempt to make the text look as visually identical to the original assignment as possible. We assume that students will copy and paste the text from the assignment into the ChatGPT input box, and because images do not get copied, the text pasted into ChatGPT will be pertubed. Table 10 in Appendix F shows the distributions of the number of participants for different variants of the assignments.

Analyzing the textual Responses. Answers to some of the questions in our post-study questionnaire were open-ended. Thus, to systematically

Table 3: Selected assignments and corresponding perturbation techniques for the user study. `Prompt (Original)` indicates prompt with no perturbation.

Perturbations	**Assignments**					
	#1	#2	#3	#4	#5	#6
`Prompt (original)`	✓	✓	✓	✓	✓	✓
`Character (remove)`	-	✓	-	-	-	✓
`Token (unicode)`	✓	✓	✓	-	-	✓
`Tokens (remove)`	✓	-	-	-	✓	-
`Sentences (rephrase)`	✓	-	-	-	-	-
`Sentences (remove)`	✓	✓	-	✓	-	-
`Prompt (unicode)`	✓	-	✓	✓	✓	✓
`Random (replace)`	✓	✓	✓	-	-	-

Table 4: Efficacy for each perturbation technique on the 6 problems we used for the user study.

Perturbations	**Avg. Efficacy**
`No perturbation`	**71.28** (Base Score)
`Character (remove)`	6.67%
`Token (unicode)`	18.08%
`Token (Remove)`	0.0%
`Sentence (Rephrase)`	0.0%
`Sentences (Remove)`	10.0%
`Prompt (unicode)`	31.25%
`Random (Replace)`	15.91%
`Combined Results`	**76.67%**

analyze those responses, we use thematic analysis, where the goal is to identify the concepts (known as *codebook*) and organize them under different themes (Jason and Glenwick, 2015; Quaium et al., 2023). Two authors participate in the process to avoid human bias. Our thematic analysis found that students use 5 different approaches to neutralize perturbations and 11 different approaches to validate LLM-generated solutions. We present a detailed description of the method and the codebook in the Appendix F.

Analyzing Solutions. The performance of black-box models changes over time. Without taking this into account, one might come to erroneous conclusions. For example, Figure 8 shows the performance of different model checkpoints on the assignment statements we use for the user study since we computed the efficacy with model checkpoint 0301. However, to ensure consistency in calculating the efficacy of the perturbation techniques in impeding the actual cheating, one needs to calculate the correctness scores for both the perturbed and unperturbed versions of the assignments on the *same* model checkpoints. Thus, we use the average correctness scores of unperturbed assignments to compute the average efficacy of a given perturbation technique.

4.2 Analysis Results

In this section, we present the results of our field experiment to answer the following three questions: **Q1:** *How effective are the perturbations, in general, in impeding LLM-assisted solution generation?* **Q2:** *How does the detectability affect efficacy?* and **Q3:** *What techniques do students adopt to avoid perturbations, and how do they validate their generated solutions?*

Impeding solution generation. Overall, the perturbations are effective in impeding LLM-assisted solution generation. Although most of the perturbations have an efficacy lower than 32%, in combination (selecting the best perturbation technique for each problem), their efficacy is around 77%, where the base correctness score was 71.28 (Table 4). This means perturbation techniques reduced 77% of the base score – showing promise in impeding LLM-assisted cheating. One interesting finding is that the `Prompt (unicode)` perturbation drops the models' performance significantly. While most students notice it and exercise several strategies, they fail to sidestep it.

Table 5: Comparison of average efficacy for the perturbation techniques based on whether they were detected or not. For `Token (remove)` and `Sentence (rephase)`, ChatGPT (GPT-3.5) generated correct solutions without any tweaks from the students.

Perturbations	**Noticed(%)**	**Unnoticed(%)**
`Character (remove)`	0.0	16.0
`Token (unicode)`	6.67	43.75
`Token (remove)`	0.0	0.0
`Sentences (rephrase)`	0.0	0.0
`Sentences (remove)`	16.67	0.0
`Prompt (unicode)`	35.71	0.0
`Random (replace)`	10.71	25.0
Total	15	15.43

Detectability vs. Efficacy. Broadly, participants notice *unusualness* in the assignments for all the perturbations (Table 6). In Table 5, we show the difference in efficacy based on whether the students notice a perturbation or not. Overall, the average efficacy dropped (15.43% to 15%) for detectability. Prompt/assignment-wide substitutions with Unicode lookalikes that alter a large portion of the assignment are easily noticed (Table 6). Despite the higher risk of being noticed, it still managed to deceive the model. Higher efficacies in noticed cases of perturbations, such as the removal of sentences and prompt-wide Unicode substitution, suggest that noticing the perturbation does not imply that students were able to reverse the changes, especially if reversing involves some degree of effort. Subtle perturbations, i.e., substitutions of tokens and removal of characters, showed great potential in tricking both the LLM and students, as they show higher efficacy when undetected.

Table 6: Unnoticed Ratios Across Perturbations

Perturbations	**Unnoticed / Total**
`Character (remove)`	5/12
`Token (unicode)`	4/13
`Token (Remove)`	2/7
`Sentence (Rephrase)`	2/3
`Sentences (Remove)`	4/10
`Prompt (unicode)`	2/16
`Random (Replace)`	4/11

Finding 4: Subtle perturbations, i.e., substituting tokens or removing/replacing characters, when unnoticed, are likely to retain high efficacy in impeding actual cheating.

Finding 5: The *detectability* of a *high-change* perturbation might not imply *reversion*.

Handling perturbed assignments. We learn from the post-user study questionnaire that even if students noticed perturbations, in most cases (32 out of 49), they rely on ChatGPT to bypass them (Figure 10). Other strategies they adopt are updating the assignment statement, rewriting incorrect ChatGPT-generated solutions, or writing the missing portions. The average efficacy against each of the strategies is highest at 31.11% when students impose *'Update problem statement'*, followed by *'No unusualness found'* at 15.43% and *'Expected to be bypassed'* at 9.17%. When students try *'Rewrite incorrect/missing portion'*, the perturbation efficacy is reduced to 0.

Validation apporaches. Approaches to validate the generated solutions also play a crucial role in detecting and fixing accuracy degradation. Most students report that they reviewed the generated code (72 out of 90 cases) or ran the code with the given test cases (55 out of 90 cases). Several of them report writing new test cases, too. A heatmap diagram of the validation approaches is presented in Figure 9 in Appendix F.

5 Discussion

Impact of Model Evolution on solving assignments. To understand how our results might be affected as LLMs evolve, we compared the capabilities of GPT-3.5 and GPT-4.0. Table 7 shows a comparison. It can be seen that GPT-4.0 does perform slightly better than GPT-3.5 on the CS2 problems, and while GPT-4.0 scored just over 12% on long problems and almost 16% on short problems for CS1, GPT-3.5 scored 0% on both, so GPT-4.0 evidently has some advanced capabilities that GPT-3.5 lacks.

Table 7: Performance comparison of GPT-3.5 and GPT-4.0 models on the CS introductory problems

Model	**CS1**		**CS2**		**Perturbed CS2 (Selected)**	
	Short	Long	Short	Long	Short	Long
`gpt-3.5-turbo-0301`	0.0	0.0	49.36	16.67	29.31	17.43
`gpt-4-0613`	15.71	13.11	56.14	23.57	39.23	15.72

Impact of Model Evolution on perturbations. We run GPT-4.0 on the prompts generated by some of the promising perturbation techniques from the user study, i.e., `Sentences (remove)`, `Token (unicode)`, and `Prompt (unicode)`. Out of the 1,113 prompts compared, GPT-4.0 outscored GPT-3.5 on 281 problems, while GPT-3.5 outscored GPT-4.0 on 107 problems (Table 7). We observe that GPT-3.5 has built-in safeguards for academic integrity violations. Surprisingly, GPT-4.0 seems to lack such safeguards. For example, GPT-3.5 refuses to solve 8 problems for triggering such safeguards, but GPT-4.0 refuses none. This finding is concerning because it suggests that GPT-4.0 could potentially be more amenable to misuse for LLM-assisted cheating.

6 Related Work

LLMs in Educational Problem Solving. Finnie-Ansley *et al.* found that OpenAI Codex produced high-quality solutions for a set of CS1 and CS2 programming problems (Finnie-Ansley et al., 2022, 2023). This suggests that LLM-assisted cheating in introductory programming courses has the potential to be problematic. Other studies note that LLM-generated code can be of variable quality and sensitive to small changes to the prompt; this hints at the idea that tweaking the problem prompt can affect the usefulness of LLM-generated solutions for academic dishonesty. For example, Wermelinger observes that "Sometimes Copilot seems to have an uncanny understanding of the problem ... Other times, Copilot looks completely clueless" (Wermelinger, 2023), and Jesse *et al.* discuss Codex's tendency to generate buggy code in some situations (Jesse et al., 2023). None of these works consider adversarial perturbation of prompts as a mechanism for hindering LLM-assisted cheating. Sadasivan *et al.* gives empirical evidence highlighting concerns that LLM-generated texts can easily evade current AI detection mechanisms (Sadasivan et al., 2023), underscoring the need for more advanced detection technologies that can follow the continuous advancements in LLM capabilities and ensuring the integrity of academic work.

Adversarial Attacks on Code Generation LLMs. Real-world applications relying on LLMs can be susceptible to vulnerabilities arising from adversarial attacks (Shayegani et al., 2023). Various strategies have been proposed to enhance the adversarial robustness of LLMs (Jiang et al., 2020; Shetty et al., 2018; Wang et al., 2021a), but these methods differ significantly, and there is a lack of standardization in the adversary setups used for valuation (Wang et al., 2021b). Wang *et al.*'s experiments show that, despite its relative dominance over other LLMs, ChatGPT's performance is nevertheless sensitive to adversarial prompts and is far from perfect when attacked by adversarial examples. To the best of our knowledge, our work is the first attempt at studying the ***Robustness in Education*** with adversarial attacks. Other research showed that adversarial attacks are also effective in breaking guards against generating malicious or unethical content (Zou et al., 2023; Liu et al., 2023a). Incorporating the methods suggested by (Wang et al., 2023b) for generating natural adversarial examples could be explored in the future.

7 Conclusion

High-performant LLMs pose a significant threat to enable cheating on introductory programming assignments. It investigates the potential of adversarial perturbation techniques to impede LLM-assisted cheating by designing several such methods and evaluating their efficacy in a user study. The result suggests that the combination of the perturbation indeed caused a 77% reduction in the correctness of the generated solutions, showing early promises. Our perturbations show positive results, but they might only be effective temporarily. Future techniques, including rigorous training data and protective layers in the prompting pipeline of LLMs, could counter these results. We hope our study will inspire ongoing efforts to prevent the misuse of LLMs in academic settings.

8 Limitations

Impact of running the user study with students exposed to the assignments. One possible limitation of our user study is that it was conducted on students who already took CS1 and CS2 courses; thus, the finding might not hold for target students. However, as the study aimed to see if students can detect and reverse our perturbations, we hypothesize that experienced students will be more equipped to do so than new ones. Thus, if our results suggest that a given perturbation technique is effective in impeding *reversal* for the study group, it is likely to be effective on the new students (actual target group) as well. However, if our results suggest that a perturbation technique is ineffective for the study group, it does not imply that it will be ineffective for the new students. This means our study offers a conservative estimation of the efficacy of the perturbation techniques on the students. Given that designing an ethically acceptable user study with new students is challenging, we argue this is acceptable. For example, Shalvi *et al.* (Shalvi et al., 2011) hypothesized that reducing people's ability to observe desired counterfactuals reduces lying. Thus, one can argue that exposing new students to the "ChatGPT way" of solving problems is ethically more questionable than exposing more mature students. This is because *a)* The fact that they will know they can get away might incentivize cheating, as they are likely unaware of the long-term consequences. The damage is arguably less for the students with some CS fundamental knowledge and more insights into the long-term consequences.

We also want to note that even if we ignore the ethical challenge mentioned above, designing a reasonable study with new students is challenging. For example, all CS students are required to take the courses from which we took the problems, and the problems typically address concepts that have been discussed in class. So, if we wanted students who have not seen those (or similar) problems, we would have to take non-CS students who have not taken those classes and who would not have the background to solve those problems. This implies either running the study as part of the course offering or emulating the course for the study. Given the duration and volume it needs, it will be challenging to design such a study while keeping all the other confounding factors (i.e., controlling the models used) in check. Given these challenges, we chose to use the ChatGPT interface for the user study instead of an API-based tool with the trade-off between user comfort and controllability of model parameters or versions. However, seeing how the findings hold under different user settings will be interesting. Considering the complexities and numerous factors in designing such studies, they warrant dedicated independent research efforts.

Impact of perturbation on understandability. Perturbations can affect understandability. Our work is intended to provide instructors with additional tools and techniques to deter LLM-assisted cheating; it is up to the instructor to ensure that any applied perturbations do not impact the clarity of the problem description. For example, a judicious application of the "sentence removal" perturbation technique we describe can be combined with using images to replace the semantic content of the removed sentences. Additionally, some perturbation techniques, such as "unicode replacement" and "character removal" may be easily reversed by a student who notices them, as our user study revealed. Thus for these "smart tweak" perturbations, the key requirement is to be as imperceptible as possible, to avoid detection. We also note that this is the first work to proactively deter the use of LLM-assisted cheating in the academic context, which is an urgent problem. It would be interesting to see what other approaches can be more effective for this purpose in the future or to run studies to find perturbations that do not affect students trying to solve problems honestly but do affect students who submit ChatGPT solutions. Additionally, prompts engineering to reverse the perturbation to understand their strengths can be a great complement to evaluating the strength of perturbations, together with user studies, or in cases where user studies might be infeasible to run. It would also be interesting to run follow-up studies on what factors affect comprehensibility to develop principles for designing "understandability-preserving perturbations." Investigating all these interesting questions can be both motivated and enabled by the current work.

Other limitations. We use CodeRL as the surrogate model, which might not be a close approximation of the target models. Despite this limitation, CodeRL is successful in generating perturbed samples to run our field study. Finally, we ran the user study with only 6 assignments, which might hurt the generalizability of the findings. ChatGPT provides personalized answers, which might cause variances in our results. To counter this, we added redundancy in our study design and reported average results.

9 Ethical Considerations

Our study was approved by the IRB of the designated institute. We recruited students who have already taken CS1 and CS2 to avoid academic integrity violations. Participants were compensated with a reward of $20 for their contribution. During the user study, we did not collect any personally identifiable data. Lastly, all the experiments on GPT-3.5 and Mistral models were done with premium API access. We also used GitHub Copilot under an academic subscription to ensure fair and responsible use. The replication package, which includes the data and source code, will be available to researchers on request.

Acknowledgements

We thank Genesis Elizabeth Benedith and Logan Michael Sandlin for their involvement during the initial stage of the project. We also thank the instructors of CS1 and CS2 who taught these courses at the University of Arizona over the years, including Adriana Picoral, Janalee O'Bagy, Reyan Ahmed, Russell Lewis, Todd Proebsting, and Xinchen Yu, for sharing the assignments, solutions, and syllabus with us. Finally, we thank the anonymous reviewers for their feedback on the initial draft of the paper.

References

Malik Al-Essa, Giuseppina Andresini, Annalisa Appice, and Donato Malerba. 2022. An XAI-based Adversarial Training Approach for Cyber-threat Detection. In *IEEE Intl. Conf. on Dependable, Autonomic and Secure Computing, Intl Conf on Pervasive Intelligence and Computing, Intl Conf on Cloud and Big Data Computing, Intl Conf on Cyber Science and Technology Congress, DASC/PiCom/CBDCom/CyberSciTech 2022, Falerna, Italy, September 12-15, 2022*, pages 1–8. IEEE.

Moustafa Alzantot, Yash Sharma, Ahmed Elgohary, Bo-Jhang Ho, Mani B. Srivastava, and Kai-Wei Chang. 2018. Generating natural language adversarial examples. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing, Brussels, Belgium, October 31 - November 4, 2018*, pages 2890–2896. Association for Computational Linguistics.

Shraddha Barke, Michael B. James, and Nadia Polikarpova. 2023. Grounded copilot: How programmers interact with code-generating models. *Proc. ACM Program. Lang.*, 7(OOPSLA1):85–111.

Pavol Bielik and Martin T. Vechev. 2020. Adversarial robustness for code. In *Proceedings of the 37th International Conference on Machine Learning, ICML 2020, 13-18 July 2020, Virtual Event*, volume 119 of *Proceedings of Machine Learning Research*, pages 896–907. PMLR.

Nicholas Boucher and Ross Anderson. 2023. Trojan Source: Invisible Vulnerabilities. In *32nd USENIX Security Symposium, USENIX Security 2023, Anaheim, CA, USA, August 9-11, 2023*. USENIX Association.

Nicholas Boucher, Ilia Shumailov, Ross Anderson, and Nicolas Papernot. 2022. Bad Characters: Imperceptible NLP Attacks. In *43rd IEEE Symposium on Security and Privacy, SP 2022, San Francisco, CA, USA, May 22-26, 2022*, pages 1987–2004. IEEE.

Nicholas Carlini, Daphne Ippolito, Matthew Jagielski, Katherine Lee, Florian Tramèr, and Chiyuan Zhang. 2023. Quantifying memorization across neural language models. In *The Eleventh International Conference on Learning Representations, ICLR 2023, Kigali, Rwanda, May 1-5, 2023*. OpenReview.net.

Nicholas Carlini, Florian Tramèr, Eric Wallace, Matthew Jagielski, Ariel Herbert-Voss, Katherine Lee, Adam Roberts, Tom B. Brown, Dawn Song, Úlfar Erlingsson, Alina Oprea, and Colin Raffel. 2021. Extracting Training Data from Large Language Models. In *30th USENIX Security Symposium, USENIX Security 2021, August 11-13, 2021*, pages 2633–2650. USENIX Association.

James Finnie-Ansley, Paul Denny, Brett A. Becker, Andrew Luxton-Reilly, and James Prather. 2022. The robots are coming: Exploring the implications of OpenAI Codex on introductory programming. In *ACE '22: Australasian Computing Education Conference, Virtual Event, Australia, February 14 - 18, 2022*, pages 10–19. ACM.

James Finnie-Ansley, Paul Denny, Andrew Luxton-Reilly, Eddie Antonio Santos, James Prather, and Brett A. Becker. 2023. My AI wants to know if this will be on the exam: Testing OpenAI's codex on CS2 programming exercises. In *Proceedings of the 25th Australasian Computing Education Conference, ACE 2023, Melbourne, VIC, Australia, 30 January 2023 - 3 February 2023*, pages 97–104. ACM.

GitHub. 2021. Your AI pair programmer. Accessed September 25, 2023. `https://github.com/features/copilot`.

Ray Hembree and Donald J Dessart. 1986. Effects of hand-held calculators in precollege mathematics education: A meta-analysis. *Journal for research in mathematics education*, 17(2):83–99.

Matthew Hertz. 2010. What do "cs1" and "cs2" mean? investigating differences in the early courses. In *Proceedings of the 41st ACM Technical Symposium on Computer Science Education*, SIGCSE '10, page 199–203, New York, NY, USA. Association for Computing Machinery.

Muntasir Hoq, Yang Shi, Juho Leinonen, Damilola Babalola, Collin F. Lynch, and Bita Akram. 2023. Detecting chatgpt-generated code in a CS1 course. In *Proceedings of the Workshop on Empowering Education with LLMs - the Next-Gen Interface and Content Generation 2023 co-located with 24th International Conference on Artificial Intelligence in Education (AIED 2023), Tokyo, Japan, July 7, 2023*, volume 3487 of *CEUR Workshop Proceedings*, pages 53–63. CEUR-WS.org.

Lorraine Jacques. 2023. Teaching CS-101 at the Dawn of ChatGPT. *Inroads*, 14(2):40–46.

Leonard A. Jason and David S. Glenwick. 2015. *Handbook of Methodological Approaches to Community-Based Research: Qualitative, Quantitative, and Mixed Methods*. Oxford University Press.

Kevin Jesse, Toufique Ahmed, Premkumar T. Devanbu, and Emily Morgan. 2023. Large language models and simple, stupid bugs. In *20th IEEE/ACM International Conference on Mining Software Repositories, MSR 2023, Melbourne, Australia, May 15-16, 2023*, pages 563–575. IEEE.

Haoming Jiang, Pengcheng He, Weizhu Chen, Xiaodong Liu, Jianfeng Gao, and Tuo Zhao. 2020. Smart: Robust and efficient fine-tuning for pretrained natural language models through principled regularized optimization. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*. Association for Computational Linguistics.

Woojeong Jin, Yu Cheng, Yelong Shen, Weizhu Chen, and Xiang Ren. 2022. A good prompt is worth millions of parameters: Low-resource prompt-based learning for vision-language models. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), ACL 2022, Dublin, Ireland, May 22-27, 2022*, pages 2763–2775. Association for Computational Linguistics.

Joyce Koh and Ben Daniel. 2022. Shifting online during covid-19: A systematic review of teaching and learning strategies and their outcomes. *International Journal of Educational Technology in Higher Education*, 19.

Hung Le, Yue Wang, Akhilesh Deepak Gotmare, Silvio Savarese, and Steven Chu-Hong Hoi. 2022. CodeRL: Mastering Code Generation through Pretrained Models and Deep Reinforcement Learning. In *NeurIPS*.

Aiwei Liu, Honghai Yu, Xuming Hu, Shuang Li, Li Lin, Fukun Ma, Yawen Yang, and Lijie Wen.

2022. Character-level White-Box Adversarial Attacks against Transformers via Attachable Subwords Substitution. In *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing, EMNLP 2022, Abu Dhabi, United Arab Emirates, December 7-11, 2022*, pages 7664–7676. Association for Computational Linguistics.

Bowen Liu, Boao Xiao, Xutong Jiang, Siyuan Cen, Xin He, Wanchun Dou, and Huaming Chen. 2023a. Adversarial attacks on large language model-based system and mitigating strategies: A case study on ChatGPT. *Sec. and Commun. Netw.*, 2023.

Nelson F. Liu, Kevin Lin, John Hewitt, Ashwin Paranjape, Michele Bevilacqua, Fabio Petroni, and Percy Liang. 2023b. Lost in the middle: How language models use long contexts. *CoRR*, abs/2307.03172.

Scott M. Lundberg and Su-In Lee. 2017. A Unified Approach to Interpreting Model Predictions. In *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA*, pages 4765–4774.

Mistral AI team. 2024. Mistral Large, our new flagship model. Accessed April 14, 2024. `https://mistral.ai/news/mistral-large/`.

John X. Morris, Eli Lifland, Jack Lanchantin, Yangfeng Ji, and Yanjun Qi. 2020. Reevaluating adversarial examples in natural language. In *Findings of the Association for Computational Linguistics: EMNLP 2020, Online Event, 16-20 November 2020*, volume EMNLP 2020 of *Findings of ACL*, pages 3829–3839. Association for Computational Linguistics.

OpenAI. 2022. GPT 3.5. Accessed September 25, 2023. `https://platform.openai.com/docs/models/gpt-3-5`.

OpenAI. 2024. Chatgpt (3.5) [large language model]. `https://chat.openai.com`. Accessed September 25, 2023.

Michael Sheinman Orenstrakh, Oscar Karnalim, Carlos Aníbal Suárez, and Michael Liut. 2023. Detecting LLM-Generated Text in Computing Education: A Comparative Study for ChatGPT Cases. *CoRR*, abs/2307.07411.

Adnan Quaium, Najla Abdulrahman Al-Nabhan, Masfiqur Rahaman, Saiful Islam Salim, Tarik Reza Toha, Jannatun Noor, Mainul Hossain, Nafisa Islam, Aaiyeesha Mostak, Md Shihabul Islam, Md. Masum Mushfiq, Ishrat Jahan, and A.B.M. Alim Al Islam. 2023. Towards associating negative experiences and recommendations reported by hajj pilgrims in a mass-scale survey. *Heliyon*, 9(5).

Jonas Rauber, Wieland Brendel, and Matthias Bethge. 2017. Foolbox v0.8.0: A python toolbox to benchmark the robustness of machine learning models. *CoRR*, abs/1707.04131.

Laria Reynolds and Kyle McDonell. 2021. Prompt programming for large language models: Beyond the few-shot paradigm. In *CHI '21: CHI Conference on Human Factors in Computing Systems, Virtual Event / Yokohama Japan, May 8-13, 2021, Extended Abstracts*, pages 314:1–314:7. ACM.

Ishai Rosenberg, Shai Meir, Jonathan Berrebi, Ilay Gordon, Guillaume Sicard, and Eli (Omid) David. 2020. Generating end-to-end adversarial examples for malware classifiers using explainability. In *2020 International Joint Conference on Neural Networks, IJCNN 2020, Glasgow, United Kingdom, July 19-24, 2020*, pages 1–10. IEEE.

Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, Artyom Kozhevnikov, Ivan Evtimov, Joanna Bitton, Manish Bhatt, Cristian Canton-Ferrer, Aaron Grattafiori, Wenhan Xiong, Alexandre Défossez, Jade Copet, Faisal Azhar, Hugo Touvron, Louis Martin, Nicolas Usunier, Thomas Scialom, and Gabriel Synnaeve. 2023. Code llama: Open foundation models for code. *CoRR*, abs/2308.12950.

Vinu Sankar Sadasivan, Aounon Kumar, Sriram Balasubramanian, Wenxiao Wang, and Soheil Feizi. 2023. Can ai-generated text be reliably detected?

Shaul Shalvi, Jason Dana, Michel JJ Handgraaf, and Carsten KW De Dreu. 2011. Justified ethicality: Observing desired counterfactuals modifies ethical perceptions and behavior. *Organizational behavior and human decision processes*, 115(2):181–190.

Erfan Shayegani, Md Abdullah Al Mamun, Yu Fu, Pedram Zaree, Yue Dong, and Nael Abu-Ghazaleh. 2023. Survey of vulnerabilities in large language models revealed by adversarial attacks. *arXiv preprint arXiv:2310.10844*.

Rakshith Shetty, Bernt Schiele, and Mario Fritz. 2018. A4NT: Author Attribute Anonymity by Adversarial Training of Neural Machine Translation. In *27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, August 15-17, 2018*, pages 1633–1650. USENIX Association.

Freda Shi, Xinyun Chen, Kanishka Misra, Nathan Scales, David Dohan, Ed H. Chi, Nathanael Schärli, and Denny Zhou. 2023. Large language models can be easily distracted by irrelevant context. In *International Conference on Machine Learning, ICML 2023, 23-29 July 2023, Honolulu, Hawaii, USA*, volume 202 of *Proceedings of Machine Learning Research*, pages 31210–31227. PMLR.

Ruoxi Sun, Minhui Xue, Gareth Tyson, Tian Dong, Shaofeng Li, Shuo Wang, Haojin Zhu, Seyit Camtepe, and Surya Nepal. 2023. Mate! are you really aware? an explainability-guided testing framework for robustness of malware detectors. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2023, San Francisco,* *CA, USA, December 3-9, 2023*, pages 1573–1585. ACM.

Boxin Wang, Shuohang Wang, Yu Cheng, Zhe Gan, Ruoxi Jia, Bo Li, and Jingjing Liu. 2021a. Infobert: Improving robustness of language models from an information theoretic perspective.

Boxin Wang, Chejian Xu, Shuohang Wang, Zhe Gan, Yu Cheng, Jianfeng Gao, Ahmed Hassan Awadallah, and Bo Li. 2021b. Adversarial GLUE: A multi-task benchmark for robustness evaluation of language models. In *Proceedings of the Neural Information Processing Systems Track on Datasets and Benchmarks 1, NeurIPS Datasets and Benchmarks 2021, December 2021, virtual*.

Jindong Wang, Xixu HU, Wenxin Hou, Hao Chen, Runkai Zheng, Yidong Wang, Linyi Yang, Wei Ye, Haojun Huang, Xiubo Geng, Binxing Jiao, Yue Zhang, and Xing Xie. 2023a. On the robustness of ChatGPT: An adversarial and out-of-distribution perspective. In *ICLR 2023 Workshop on Trustworthy and Reliable Large-Scale Machine Learning Models*.

Zimu Wang, Wei Wang, Qi Chen, Qiufeng Wang, and Anh Nguyen. 2023b. Generating valid and natural adversarial examples with large language models.

Michel Wermelinger. 2023. Using github copilot to solve simple programming problems. In *Proceedings of the 54th ACM Technical Symposium on Computer Science Education, Volume 1, SIGCSE 2023, Toronto, ON, Canada, March 15-18, 2023*, pages 172–178. ACM.

Lei Xu, Alfredo Cuesta-Infante, Laure Berti-Équille, and Kalyan Veeramachaneni. 2022. R&r: Metric-guided adversarial sentence generation. In *Findings of the Association for Computational Linguistics: AACL-IJCNLP 2022, Online only, November 20-23, 2022*, pages 438–452. Association for Computational Linguistics.

Chiyuan Zhang, Daphne Ippolito, Katherine Lee, Matthew Jagielski, Florian Tramèr, and Nicholas Carlini. 2021. Counterfactual Memorization in Neural Language Models. *CoRR*, abs/2112.12938.

Ningyu Zhang, Luoqiu Li, Xiang Chen, Shumin Deng, Zhen Bi, Chuanqi Tan, Fei Huang, and Huajun Chen. 2022. Differentiable prompt makes pre-trained language models better few-shot learners. In *The Tenth International Conference on Learning Representations, ICLR 2022, Virtual Event, April 25-29, 2022*. OpenReview.net.

Yunqing Zhao, Tianyu Pang, Chao Du, Xiao Yang, Chongxuan Li, Ngai-Man Cheung, and Min Lin. 2023. On evaluating adversarial robustness of large vision-language models. *CoRR*, abs/2305.16934.

Andy Zou, Zifan Wang, Nicholas Carlini, Milad Nasr, J. Zico Kolter, and Matt Fredrikson. 2023. Universal and transferable adversarial attacks on aligned language models.

A Syllabus of CS1

A.1 Course Description

An introduction to programming with an emphasis on solving problems drawn from a variety of domains. Topics include basic control and data structures, problem-solving strategies, and software development tools and techniques. Specifically, the Python programming language will be taught.

A.2 Course Objectives

By the end of the semester, you should be able to write complete, well-structured programs in Python.

A.3 Expected Learning Outcomes

Students who successfully complete this course should be able to:

- Use variables, control structures, basic data types, lists, dictionaries, file I/O, and functions to write correct 100 - 200 line programs.
- Decompose a problem into an appropriate set of functions, loops, conditionals, and/or other control flow.
- Find bugs when code is not working as expected using print statements and computational thinking skills, and will be able to understand and resolve errors.
- Write clean, well-structured, and readable code.
- Follow a provided style guide to write clean, well-structured, and readable code.
- Explain the conceptual memory model underlying the data types covered in class and demonstrate the ability to convert integers and text to and from binary.

B Syllabus of CS2

B.1 Course Description

This course provides a continuing introduction to programming with an emphasis on problem-solving. It considers problems drawn from various domains (including Computer Science). It emphasizes both the broader applicability of the relevant data structures and programming concepts, as well as the implementation of those structures and concepts in software. Topics include arrays, lists, stacks, queues, trees, searching and sorting, exceptions, classes and objects; asymptotic complexity; testing, and debugging.

B.2 Course Objectives

The course will provide a foundation in fundamental computer science concepts such as object-oriented programming, data structures and abstract data types, asymptotic worst-case complexity, program design, testing, and debugging.

B.3 Expected Learning Outcomes

Students who successfully complete this course should be able to:

- Effectively decompose simple programming problems into suitable functions.
- Comfortably write moderate-sized (100–300 line) programs incorporating a variety of control and data structures.
- Implement common data structures such as stacks, queues, linked lists, and trees and use recursive solutions when appropriate;
- Implement classes given design guidance;
- Use a provided style guide to produce clean, readable code;
- Identify and create black box and white box tests and use assertions to facilitate the testing and debugging of their programs;
- Determine the time complexity of simple algorithms and state their complexity in terms of big-O notation.

C Short and Long Problems

Figure 4 shows an example of short and long problems.

D LLM Code Generation Methodology

CodeRL. To initiate code generation with CodeRL, we first create an instance of the tokenizer and model using the HuggingFace API. To ensure obtaining the best solution, we set the *temperature* to 0 and the output token limit to its maximum allowable limit. Then, we tokenize the prompt and send it to the model. The model generates a list of tokens from the given prompt of tokens. After detokenizing the output, we get a source code, which serves as the solution to the given assignment problem.

```
In a file jaccard.py write a function jaccard(set1, set2)
↪  that takes as arguments two sets set1 and set2 and
↪  returns a floating-point value that is the Jaccard
↪  similarity index between set1 and set2.  The definition
↪  of the Jaccard similarity index is (see also: Section 2.B
↪  of the long problem spec; Wikipedia):
similarity(set1, set2)  = | set1 ∩ set2 | / | set1 ∪ set2 |

If set1 and set2 are both empty sets, their similarity is
↪  defined to be 1.0.

Examples
set1 set2 jaccard(set1, set2)
{'aaa', 'bbb', 'ccc', 'ddd'} {'aaa', 'ccc'} 0.5
{1, 2, 3} {2, 3, 4, 5} 0.4
{1, 2, 3} {4, 5, 6} 0.0
```

(a) Short problem

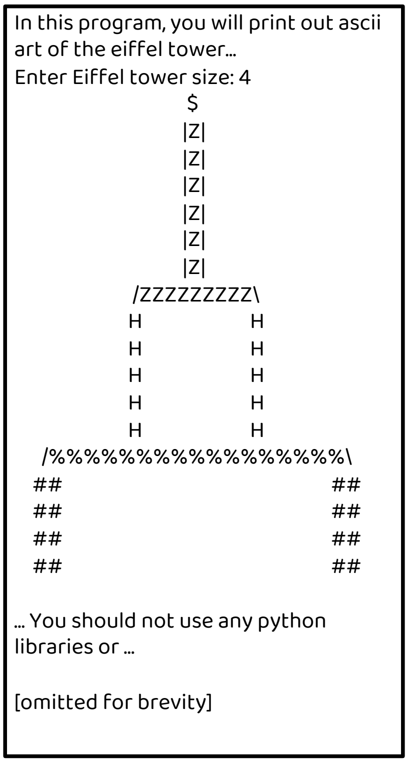
```
In a file update_board.py write the following functions:
update_board(board, mov): board is an internal
↪  representation of a board position, mov is a tuple of
↪  integers specifying a move. It returns the internal
↪  representation of the board resulting from making the
↪  move mov in board board.
update_board_interface(board_str, mov): board_str is an
↪  external representation of a board position (a string of
↪  0s and 1s), mov is a tuple of integers specifying a move.
↪  This function converts board_str to your internal
↪  representation of a board position, calls your function
↪  update_board() described above, converts the value
↪  returned by update_board() to an external representation
↪  of a board (a string of 0s and 1s), and returns the
↪  resulting string. This function thus serves as the
↪  external interface to your update_board() function.
2.3.2. Examples

board_str mov update_board_interface(board_str, mov)
110001100101011 (14, 13, 12) 110001100101100
110001100101011 (0, 1, 3) 000101100101011
0110011011 (5, 2, 0) 1100001011
```

(b) Long problem

Figure 4: Examples of short and long problems

GitHub Copilot. To generate code with Copilot, we employ PyAutoGUI to automate VS Code. The step-by-step process starts with opening VS Code in a new window and creating a new Python file. We paste the prompt into the file, surrounded by a docstring comment. Next, we ask Copilot to generate multiple variations of code in a new window using the custom keyboard shortcut. Then, we close the VS Code after saving the responses in separate files. The subsequent steps vary based on the type of problem. For short problems, we handle cases where the code can either be a standalone program generating output or a function/class definition. In the latter case, the code generation is done for that specific code. Conversely, for standalone programs, we add the “`if __name__ == '__main__':`” block at the bottom of the file and let Copilot call the generated function/class. At this point, Copilot provides inline suggestions rather than separate windows for alternatives. For longer problems, we reopen the generated code in VS Code and



```
In this program, you will print out ascii
art of the eiffel tower...
Enter Eiffel tower size: 4
              $
             |Z|
             |Z|
             |Z|
             |Z|
             |Z|
             |Z|
         /ZZZZZZZZZ\
         H         H
         H         H
         H         H
         H         H
         H         H
  /%%%%%%%%%%%%%%%%%\
 ##                       ##
 ##                       ##
 ##                       ##
 ##                       ##

... You should not use any python
libraries or ...

[omitted for brevity]
```

Figure 5: An example CS1 problem where CodeRL, GPT-3.5and GitHub Copilot scored 0%.

allow Copilot to provide up to 15 inline suggestions. However, if Copilot generates its own "`if __name__ == '__main__':`" block, we stop, as further code generation may lead to uncompilable results.

As both short and long problems can generate up to 10 solutions for a single prompt, we run all generated solutions through autograders and select the one with the highest score for evaluation. This methodology ensures efficient code generation and selection of the most appropriate solution for the given prompt.

```
Write a Python program that does the following:

<problem statement>

Please omit any explanations of the code.
```

Figure 6: Prompt to generate source code from GPT-3.5

GPT-3.5. We use the OpenAI API to generate code using GPT-3.5. Specifically, we use the `gpt-3.5-turbo-0301` model to ensure consistency throughout our experiments. Similar to CodeRL, we set the *temperature* to 0 to obtain the most optimal source code deterministically. Since GPT-3.5 is a general-purpose language model not specifically designed for code generation only, we add qualifying sentences around the prompt instructing GPT-3.5 to omit explanations and produce only code (since non-code explanatory text could induce syntax errors in the autograder). Figure 6 shows the prompt we use to generate code from GPT-3.5. This way, we exclusively receive code outputs from the model.

Mistral. We used the Mistral API to generate code using Mistral. Specifically, we used the `mistral-large-2402` model to ensure consistency throughout our experiments. Because Mistral's API is very similar to OpenAI's API, we followed the same methodology and used the same model parameters to interact with the API.

Code Llama. We used Ollama, a lightweight and extensible framework for running LLMs on local machines, to host the CodeLlama-7b-instruct model based on Meta's Llama 2. The instruct model was chosen as it is trained to output human-like answers to given queries, which we believed to be closest to ChatGPT in terms of the generated solutions. The steps include installing Ollama and simply calling **ollama run codellama:7b-instruct '<prompt>'** to generate the outputs. To the best of our knowledge, there isn't a straightforward way to tweak the parameters of the models from the provided user manuals, so we used the default model. Although the generated answers often contained comment blocks as well as codes, most outputs wrapped the code blocks with identifiable texts such as **'''**, **[PYTHON]** or **'''python**, we extracted the codes accordingly. Otherwise, we simply used the generated output.

E Descripiton of our Perturbation Techniques

E.1 Core perturbations.

Token (remove): Breaking subword tokens profoundly impacts LLM performance (Liu et al., 2022; Wang et al., 2021b). By consulting SHAP, in this technique, we remove the top 5 tokens from the assignment description and create 1 perturbed variant of a given assignment. We generated 63 short and 12 long variants in total.

Character (remove): Following the same principle as *Token (remove)* to break subwords, in this perturbation technique, we remove a random character from each of the top 5 tokens to create 1 variant. We generated 63 short and 12 long variants in total.

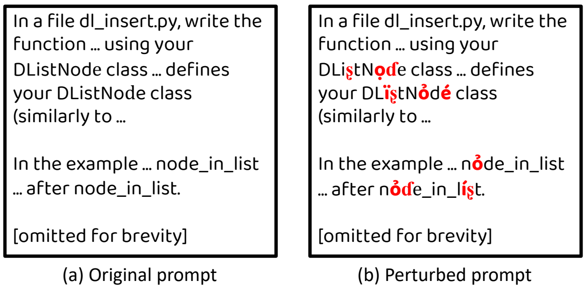
Random (insert): To break subwords, we also design another perturbation by inserting redundant characters, such as hyphens and underscores, in the top 5 tokens; similarly, we generate 1 variant of inserting redundant characters, such as hyphens and underscores, into the top tokens in the assignments. We generated 63 short and 12 long variants in total.

Sentence (remove): For sentence removal, we remove a third of the sentence from the assignment description sequentially. We chose one-third so as to not remove too much relevant information, and we removed sequential sentences to create a large hole in the information provided to the models. If the assignment description has less than 3 sentences, we remove only 1 sentence. This produces a variable number of perturbed variants. We generated 594 short and 857 long variants in total.

Sentence (rephrase): Rephrasing of sentences is known to be effective in degrading LLM performance (Xu et al., 2022; Morris et al., 2020; Alzantot et al., 2018; Wang et al., 2021b). Thus, we leverage rephrasing sentences to design this perturbation. First, we rank the sentences by accumulating the Shapley values of the tokens corresponding to a given sentence; then, we remove the top 3 sentences to create 3 independent variants. We use GPT-3.5to obtain high-quality phrases. We generated 177 short and 32 long variants in total.

Token (synonym): Tokens are the building blocks of language models, which have been used as perturbation units in context (Boucher and Anderson, 2023; Al-Essa et al., 2022; Wang et al., 2021b). Therefore, we design a perturbation technique. to substitute tokens with their synonyms. Specifically, we replace the top 5 tokens from the SHAP with their synonyms to create 5 different variants. For each top-ranked token, we replace all instances of that token in the prompt with its synonym, even if other occurrences are not top-ranked. We do this to ensure that if the token provides necessary information to the model, it cannot be obtained from another token occurrence in the assignment description. We generate contextual synonyms for a given token using GPT-3.5. We provide the sentence containing the token as the context for the GPT-3.5 model and ask for synonyms for the token. We generated 1836 short and 216 long variants in total.

Token (unicode): Recent research shows that adversarial attacks can be effective even in a black-box setting without visually altering the inputs in ways noticeable to humans, which includes replacing characters with Unicode lookalikes (Shetty et al., 2018; Boucher et al., 2022). To leverage this, we create a perturbation method to replace characters in the top 5 tokens (from SHAP) with their Unicode lookalikes to create 1 variant (Figure 7). We generated 63 short and 12 long variants in total.



In a file dl_insert.py, write the function ... using your DListNode class ... defines your DListNode class (similarly to ...

In the example ... node_in_list ... after node_in_list.

[omitted for brevity]

(a) Original prompt

In a file dl_insert.py, write the function ... using your DLiştNọɗe class ... defines your DLïştNỏdé class (similarly to ...

In the example ... nỏde_in_list ... after nỏɗe_in_líşt.

[omitted for brevity]

(b) Perturbed prompt

Figure 7: Replacing 12 characters for 5 tokens with their Unicode lookalike from an assignment prompt caused correctness scores to drop from 100% to 0% in GPT-3.5.

E.2 Exploratory Perturbations.

Tokens (synonym): To understand the potential of synonym-based perturbation, we create a new type of perturbation method to replace the top 5 tokens from the SHAP with their synonyms to create 5 different variants. However, we do not replace the top-ranked occurrences of a given token – not all occurrences in a given assignment prompt. We generated 2373 short and 223 long variants in total.

Prompt (Unicode): Similarly, to study the full potential of substituting characters with Unicode lookalikes, we apply it to the whole assignment statement under this technique. We recognize that this perturbation might easily get noticed; however, we add it to understand how detectability might impact the actual performance in the field study. We generated 63 short and 12 long variants in total.

Random (replace): Existing studies show evidence that LLMs are prone to memorizing training data (Zhang et al., 2021; Carlini et al., 2021, 2023). Thus, these models are highly sensitive to input variations, and even slight changes in the prompt may lead to substantial differences in the generated output (Zhang et al., 2022; Jin et al., 2022; Reynolds and McDonell, 2021). Under this hypothesis, replacing specific tokens with random strings may significantly influence performance, as such substitution may alter the context (Shi et al., 2023; Liu et al., 2023b; Wang et al., 2021b). We design a new exploratory perturbation technique to leverage this insight. Under this technique, we tweak assignments by replacing file names, function names, and class names specified in the problem statement with random strings, where these names are discovered manually. We store the original names and random strings, then in the code generated by the models, replace the instances of the random strings with the original names. This is to make sure that the autograders don't give a score of 0 for a good solution that uses the random string. We generated 63 short and 12 long variants in total.

F User Study

Table 8: Demography of the participants

Participants	Academic Status	Proficiency in Python (out of 5)	LLM Usage Frequency (weekly)
P1	Junior	5	Occasionally (3-5 times)
P2	Junior	4	Never
P3	Senior	5	Occasionally (3-5 times)
P4	Senior	5	Occasionally (3-5 times)
P5	Senior	5	Very frequently (More than 10 times)
P6	Senior	4	Rarely (1-2 times)
P7	Sophomore	4	Occasionally (3-5 times)
P8	Senior	4	Very frequently (More than 10 times)
P9	Sophomore	4	Occasionally (3-5 times)
P10	Senior	4	Occasionally (3-5 times)
P11	Senior	4	Regularly (6-10 times)
P12	Senior	4	Rarely (1-2 times)
P13	Sophomore	5	Occasionally (3-5 times)
P14	Senior	4	Rarely (1-2 times)
P15	Junior	4	Rarely (1-2 times)
P16	Senior	4	Rarely (1-2 times)
P17	Junior	4	Occasionally (3-5 times)
P18	Junior	4	Occasionally (3-5 times)
P19	Sophomore	4	Never
P20	Junior	3	Never
P21	Junior	5	Rarely (1-2 times)
P22	Senior	4	Never
P23	Junior	3	Rarely (1-2 times)
P24	Senior	5	Very frequently (More than 10 times)
P25	Senior	4	Never
P26	Senior	4	Regularly (6-10 times)
P27	Junior	4	Occasionally (3-5 times)
P28	Junior	3	Rarely (1-2 times)
P29	Senior	4	Very frequently (More than 10 times)
P30	Senior	4	Regularly (6-10 times)

Table 9: User Study Questions

Questions
How proficient are you in the Python programming language?
How hard did the problem seem to you while you were solving it? (For each problem)
How much time (in minutes) did you spend on this problem? (For each problem)
How did you validate the ChatGPT-generated solutions? (For each problem)
Did you notice anything unusual about the problem statement? (For each problem)
How did you avoid the "unusualness" in the problem statement while solving the problem? (For each problem)
On average, how many hours do you dedicate to coding or problem-solving per week?
How often do you utilize ChatGPT or any other Large Language Model to solve problems on a weekly basis, on average?
What other Large Language Models do you use or previously used?

F.1 Description of the thematic analysis

This approach consists of multiple stages. First, we familiarize ourselves with the collected data. We manually go through 50% (15 out of 30) responses in this stage. This allows us to perform inductive coding to identify potential codes for further analysis. In the second stage, two authors generated 16 initial codes based on their familiarity with the data. These codes are data-driven and help organize information into meaningful units. Two authors assign codes to the participants' responses to the specific questions. This coding stage is done manually. To address disagreements, the authors facilitated a consensus-based resolution while combining their coding assignments. Consensus-based resolution is considered important in qualitative studies to produce meaningful insights. In our case, there were 4 disagreements between the two raters while labeling all 30 participant's data. After that, one of the authors reviews the students' responses and corresponding conversations with ChatGPT to get the most information and update the coding. This step is iterative until saturation. We consider the coding to be saturated if no new code is assigned to the responses. Lastly, the other author validates the final coding to avoid potential bias. In the third stage, after coding the data, we start searching for themes by bringing together material under the same codes. This involves considering how codes may form broader themes that are organized hierarchically. In the fourth stage, we review and refine the potential themes.

Table 10: Distributions of the perturbation techniques and the problems in the user study

Perturbations	#Participants
Prompt (original)	18
Character (remove)	12
Token (unicode)	13
Tokens (remove)	7
Sentences (rephrase)	3
Sentences (remove)	10
Prompt (unicode)	16
Random (replace)	11

Problems	# Participants
p1	22
p2	17
p3	13
p4	13
p5	13
p6	12

Codebook for neutralizing perturbations:

- Update the given problem statement
- Rely on ChatGPT to avoid any perturbation
- Did not notice anything "unusualness"
- Rewrite the whole solution manually as the ChatGPT-generated solution is incorrect
- Rewrite a part of the solution manually

Themes and codes for validation:

- Inspecting the generated code
    - Inspect the generated code without running

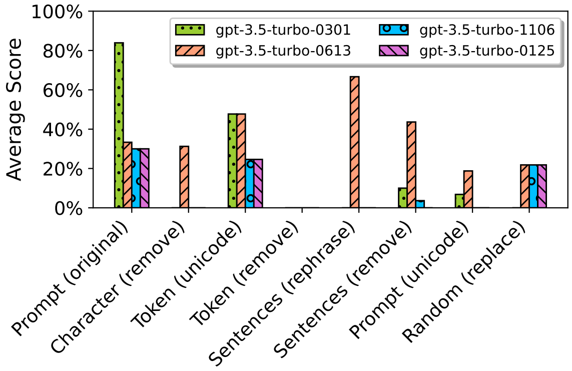


Figure 8: Average correctness score of the ChatGPT model checkpoints on the user study problems for the perturbation techniques.

  - Inspect the generated code by running
  - Use given test cases
  - Use manually created test cases
  - Use ChatGPT-generated test cases
  - Validate the solution using ChatGPT
  - Compare to the manually written code

- Fixing the generated code
  - Fix the code manually
  - Fix the code using ChatGPT

- Verdict about the correctness
  - Correct solution from ChatGPT
  - Incorrect solution from ChatGPT

G Research Participant Agreement

G.1 Voluntary Participation

You are being asked to participate in a research study. Your participation in this research study is voluntary. You may choose to voluntarily discontinue participation in the study at any time without penalty, even after starting the survey. This document contains important information about this study and what to expect if you decide to participate. Please consider the information carefully. Feel free to ask questions before deciding whether to participate.

Through this study, we will understand how well we can solve CS1 and CS2-level programming tasks using AI tools such as ChatGPT. The survey consists of three CS introductory assignment problems for each student. For each problem, you have to solve it using ChatGPT and then answer the follow-up questions. We estimate that the whole process will take around 45-60 minutes. You are free to take the survey anywhere you choose. You will be emailed the survey to complete, and you will need to provide your email address in the survey.

By signing up you are agreeing that you took CS1 and CS2. You will proceed with the study once the verification of your historical enrollment in the CS1 and CS2 courses is confirmed with the moderator of the CS undergraduate listserv (Martin Marquez, Director of Academic and Support Services, CS). Education records used by this research project are education records as defined and protected by the Family Educational Rights and Privacy Act (FERPA). FERPA is a federal law that protects the privacy of student education records. Your consent gives the researcher permission to access the records identified above for research purposes.

G.2 Risks for the Participants

1. **Social risk:** A minor risk is the potential of loss of confidentiality because the form asks for your email address. Google Forms automatically collects email addresses for the survey, so the email address will be attached to the survey responses.

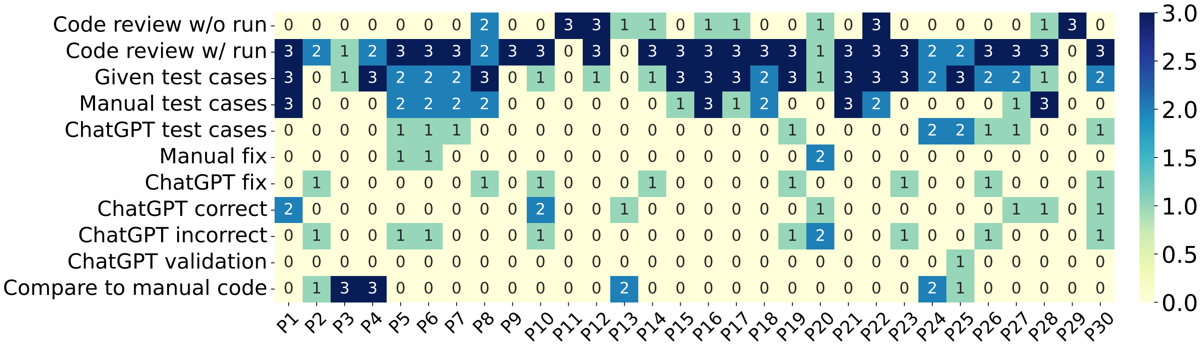
2. **Economic risk:** An economic risk may be that you complete the vast majority of the survey, but we cannot reward any cash, and so you lose some leisure time with no cash reward.

3. **Psychological risk:** A psychological risk may be that you may get fatigued while solving the given problems.

However, the risks here are largely minimal. The analysis considers the survey responses as a whole and does not investigate one specific survey response. That said, your email address will be removed before the analysis of the surveys after you collect your reward (details below).

G.3 Incentive

You will receive a $20 Amazon e-gift card for completing the survey in full. To receive your $20 award, please contact the Anonymized author. He will then check that you have completed the survey in full using your email and arrange the payment. You must collect your reward within one month of completing the survey. For any compensation you receive, we are required to obtain identifiable information such as your name and address for financial compliance purposes. However, your name will



	P1	P2	P3	P4	P5	P6	P7	P8	P9	P10	P11	P12	P13	P14	P15	P16	P17	P18	P19	P20	P21	P22	P23	P24	P25	P26	P27	P28	P29	P30
Code review w/o run	0	0	0	0	0	0	0	2	0	0	3	3	1	1	0	1	1	0	0	1	0	3	0	0	0	0	0	1	3	0
Code review w/ run	3	2	1	2	3	3	3	2	3	3	0	3	0	3	3	3	3	3	3	1	3	3	3	2	2	3	3	3	0	3
Given test cases	3	0	1	3	2	2	2	3	0	1	0	1	0	1	3	3	3	2	3	1	3	3	3	2	3	2	2	1	0	2
Manual test cases	3	0	0	0	2	2	2	2	0	0	0	0	0	0	1	3	1	2	0	0	3	2	0	0	0	0	1	3	0	0
ChatGPT test cases	0	0	0	0	1	1	1	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	2	2	1	1	0	0	1
Manual fix	0	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	2	0	0	0	0	0	0	0	0	0	0
ChatGPT fix	0	1	0	0	0	0	0	1	0	1	0	0	0	1	0	0	0	0	1	0	0	0	1	0	0	1	0	0	0	1
ChatGPT correct	2	0	0	0	0	0	0	0	0	2	0	0	1	0	0	0	0	0	0	1	0	0	0	0	0	0	1	1	0	1
ChatGPT incorrect	0	1	0	0	1	1	0	0	0	1	0	0	0	0	0	0	0	0	1	2	0	0	1	0	0	1	0	0	0	1
ChatGPT validation	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0
Compare to manual code	0	1	3	3	0	0	0	0	0	0	0	0	2	0	0	0	0	0	0	0	0	0	0	2	1	0	0	0	0	0

Figure 9: The vertical axis lists the most frequent validation strategies, while the horizontal axis represents participants. Each cell's value, capped at 3, indicates the number of times a specific code was applied to a participant's response across three problems. The color gradient ranges from bright yellow (indicating 0 occurrences) to dark blue (indicating 3 occurrences).

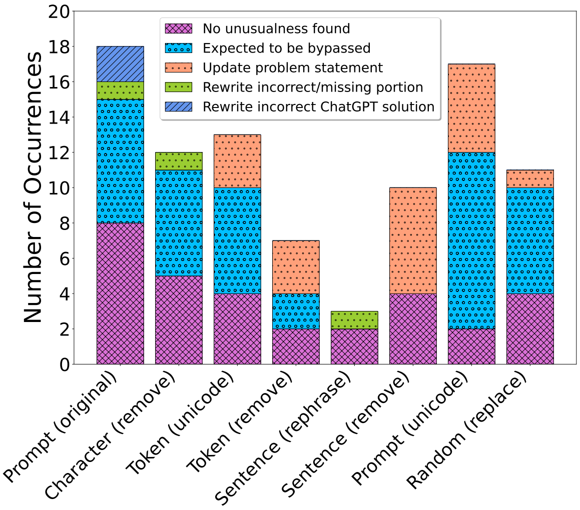


Figure 10: Number of occurrences of handling strategies for each perturbation technique.

not be used in any report or analysis of the survey results. Identifiable research data will be stored on a password-secured local lab computer accessible only to the research project members.

G.4 Confidentiality of Data

Your information may be used for future research or shared with another researcher for future research studies without additional consent. In addition, your email addresses will be deleted from the response spreadsheets, which will be stored on a password-secured local server computer accessible only by the research team members. The form containing the list of student emails that signed up to participate will be deleted once all surveys are complete. Once the entire research project is complete and the conference paper is published, anyone can view the results of the survey by referring to the conference website. The conference at which this paper will be accepted cannot be guaranteed at this moment.

The information that you provide in the study will be handled confidentially. However, there may be circumstances where this information must be released or shared as required by law. The Institutional Review Board may review the research records for monitoring purposes.

For questions, concerns, or complaints about the study, you may contact the Anonymized author. By completing the entire survey, you are allowing your responses to be used for research purposes.

G.5 Instructions to the Participants

1. Create a free ChatGPT (3.5) account if you don't have any.

2. Each problem comes with a problem statement (shared via email). Create a separate chat window in ChatGPT to solve each problem.

3. After solving each problem, you have to answer the corresponding survey questions.

4. You also have to give the shareable link of the chat from ChatGPT for each problem. (ChatGPT Shared Links FAQ)

5. Don't delete the chats until you receive an email from us about the deletion step.