LLMs are Brittle to Simple Code Transformations: Introducing CETBench – A Benchmark for Code-Equivalence Checking

Anonymous ACL submission

Abstract

LLMs have been extensively used for the task of automated code generation. In this work, we examine the applicability of LLMs for the related but relatively unexplored task of code-equivalence checking, i.e., given two programs, whether they are functionally equivalent or not. This is an important problem since benchmarking code equivalence can play a critical role in evaluating LLM capabilities for tasks such as code re-writing and code translation. Towards this end, we present CETBench- Code Equivalence with Transformations Benchmark, constructed via a repository of programs, where two programs in the repository may be solving the same or different tasks. Each instance in our dataset is obtained by taking a pair of programs in the repository and applying a random series of pre-defined code transformations, resulting in (non-)equivalent pairs. Our analysis on this dataset reveals a surprising finding that very simple code transformations in the underlying pair of programs can result in a significant drop in performance of SOTA LLMs for the task of code-equivalence checking. We find these challenges are exacerbated in the cross-lingual setting when comparing programs from different languages. To remedy this, we present a simple fine-tuning-based approach to boost LLM performance on the transformed pairs of programs. Our approach for dataset generation is generic, supporting cross-lingual equivalence checking, generation of program pairs with varying difficulty levels, and the application of diverse transformations. In our experiments, we perform ablations over the difficulty level of original programs, as well as the kind of transformations used in generating pairs for equivalence checking. Our analysis presents deep insights into the working of LLMs for the task of code-equivalence, and points to the fact that they may still be far from what could be termed as a semantic understanding of the underlying code.

1 Introduction

The field of AI-assisted code generation and understanding has witnessed a rapid rise in adoption and capability over the past few years, with models increasingly integrated into real-world software development workflows (Jin et al., 2025). The reasons behind this trend include (i) the emergence of powerful, general-purpose large language models and (ii) the development of standardised benchmarks like HumanEval (Chen et al., 2021) and MBPP (Austin et al., 2021), which evaluate text-to-code generation. In this work, our focus is on the related task of checking *code-equivalence*: given two pieces of code, $p1$ and $p2$, how effective are LLMs in determining whether the two pieces of code are equivalent or not. While there are many definitions of equivalence in the literature, we focus on the most general one, which is *functional equivalence*, i.e., two programs are said to be equivalent if for any given input, they produce exactly the same output. Doing this task successfully can help with multiple auxiliary tasks, such as verifying the correctness of code rewrites, code translation, etc. While there has been a good amount of work in the classical literature on this task, using AST and program analysis (Baxter et al., 1998; Gabel et al., 2008), this is very tedious, does not scale, and often has multiple challenges. Of late, there have been some attempts to examine the capabilities of LLMs for the task of code-equivalence checking, with some of the approaches being parallel to our work (Maveli et al., 2025; Wei et al., 2025). We highlight the differences from these works, as well as contrast with other related work in the related work section.

In this work, we provide a general-purpose dataset to evaluate the robustness of LLMs for the undecidable task of code equivalence. The dataset is generated using the following steps: (a) we start with a repository of programs written to solve a

variety of programming tasks. (b) We construct a set of pairs of programs by randomly picking programs which (i) solve the same task (either correctly or incorrectly) (ii) solve different tasks. (c) each pair in the set is taken through a series of (randomly picked) pre-defined transformations, such as *variable renaming*, *swap of if-else blocks*, etc., to generate the corresponding perturbed pair. We note that some of the transformations are semantic preserving (sp), and others are not (np). These steps result in the dataset of perturbed pairs, and are referred to as CETBench. We perform a detailed analysis of relative performance among different pair categories, before and after introducing the perturbations. Our analysis shows that for a variety of SOTA LLMs, including both open source and closed source, the performance drops significantly on the perturbed dataset. Of specific interest is the drop in performance when the starting pair represents a single program. We present additional insights on LLM performance, we vary the difficulty level (MikeMirzayanov, 2018; Ebtekar, 2019) of the original program repository. Additional ablations present the variation in performance across a variety of LLMs, with varying kinds of transformations applied. Furthermore, we extend our analysis to the cross-lingual setting, revealing that these challenges are exacerbated when comparing code from different programming languages.

Finally, we present a simple fine-tuning-based approach to boost LLM performance on the unperturbed as well as perturbed datasets. Best results are obtained when the fine-tuning is done using a combination of pairs from the perturbed as well as unperturbed datasets. We also demonstrate that the knowledge gained from this fine-tuning successfully transfers to unseen, more difficult tasks in a zero-shot setting. Overall, our experiments present deep insights into the working of LLMs for the task of code-equivalence checking for a variety of categories. Further, our findings clearly point to the fact that LLMs are still quite far from semantic understanding of underlying programs, and likely rely on statistical patterns for the task of code equivalence.

The contributions of this paper can be summarised as follows: (a) We present a novel dataset of perturbed program pairs generated via a series of code transformations applied randomly to pairs of programs in a repository. (b) We present a detailed analysis demonstrating a significant drop in LLM performance over the perturbed program pairs. We also present additional ablations across varying program difficulty, cross-lingual settings, and the types of transformations applied. (c) We propose a fine-tuning approach that not only boosts performance on the perturbed dataset (proving LLMs can learn with explicit supervision) but also demonstrates strong generalizability, as the learned skills successfully transfer to more difficult and cross-lingual tasks. *We plan to release our dataset and code publicly upon acceptance.*

2 Related Work

Determining code equivalence—whether two programs yield identical outputs for all possible inputs—is a foundational challenge in computer science, relevant to program verification (Chakraborty et al., 2023), refactoring (Liu et al., 2025), translation (Roziere et al., 2020), and optimization (Rosas et al., 2024). While undecidable in the general case (Goldblatt and Jackson, 2012), several heuristic and formal methods have been developed.

Traditional Approaches: Static program analysis techniques (e.g., AST comparison (Baxter et al., 1998), data-flow analysis (Gabel et al., 2008)) struggle with semantics-preserving syntactic variations. Formal methods like symbolic execution (Kim et al., 2011) and SMT solving (Stolee et al., 2014) offer stronger guarantees but often lack scalability. Test-case-based execution is more practical in real-world settings; however, it has fundamental limitations in establishing true functional equivalence. This limitation arises because test suites inherently cover only a finite subset of potentially infinite program input domains and thus passing these tests merely indicates identical behaviour for those specific inputs rather than universal functional equivalence. Additionally, developing comprehensive test suites remains a significant practical challenge for complex, real-world software.

LLM-as-a-Judge for Code Evaluation: The *LLM-as-a-Judge* paradigm (Gu et al., 2024)—where LLMs are used to assess task outputs—has gained traction for its scalability and flexibility in both NLP (Zheng et al., 2023; Dong et al., 2024) and code-related tasks (Tong and Zhang, 2024; Zhuo, 2024; He et al., 2025). However, functional equivalence checking demands deeper semantic reasoning, making LLM reliability in this setting an open question. It requires more than understanding surface-level syntax or problem

intent; it demands robust semantic reasoning to judge whether two programs, even if structurally dissimilar, will produce identical behaviour across all possible inputs.

Benchmarks for Code Equivalence: Prior datasets like BigCloneBench (Svajlenko et al., 2014) suffer from false positives and class imbalance (Krinke and Ragkhitwetsagul, 2022). GPT-CloneBench (Alam et al., 2023) relies heavily on LLM-generated code, limiting its diversity. EqBench (Badihi et al., 2021) is limited in scale and diversity, with only 147 equivalent and 125 non-equivalent C/Java pairs, making it less suitable for robust LLM evaluation.

Recent Parallel Work: Two recent benchmarks are closely related to our work: (More detailed experimental comprision in Appendix C)

SeqCoBench: Maveli et al. (2025) apply over 20 single-step transformations to Python programs from MBPP (Austin et al., 2021), focusing on subtle functional changes. In contrast, CETBench applies composed sequences of up to seven transformations, often across distinct but algorithmically similar programs. Additionally, our work evaluates CETBench on frontier models such as GPT-4o, extending beyond the models used in SeqCoBench.

EquiBench: Wei et al. (2025) present a multi-language benchmark. While three of EquiBench's six categories (OJ_A, OJ_V, OJ_VA) are Python-based and sourced from online judge submissions like CETBench, key differences remain:

• **Transformation Depth:** CETBench applies composed sequences of up to seven diverse transformations. In contrast, EquiBench's OJ_A and OJ_V use single transformation types (algorithmic difference or variable renaming), while OJ_VA combines only two. Our compositional complexity provides a more rigorous test of LLM robustness.

• **Nature of Negative Pairs:** CETBench generates negative pairs via subtle, semantic-breaking changes between similar or identical programs, unlike EquiBench's use of syntactically distinct wrong-answer submissions, which are often easier to classify.

• **Prompting Context:** EquiBench includes problem descriptions in prompts; CETBench omits them to focus solely on LLMs' standalone code reasoning abilities.

• **Cross-Lingual:** CETBench is the first benchmark to include comparing two programs written in two different languages.

3 CETBench: Constructing a Perturbation-Based Benchmark

To evaluate the ability of LLMs to determine program equivalence, we construct a perturbation-based benchmark that systematically generates *both* semantically equivalent and non-equivalent pairs of code. Our design mirrors practical scenarios in code translation (Roziere et al., 2020), repair (Le Goues et al., 2019), and synthesis (Devlin et al., 2017), where seemingly minor changes in the code can either leave the intended behaviour unchanged or silently change it.

3.1 Source Corpora

Our benchmark is constructed by extending the CodeContests dataset[1], originally introduced by DeepMind to train AlphaCode (Li et al., 2022). It consists of approximately 10000 competitive programming problems curated from platforms such as Codeforces[2]. Each problem includes complex problem descriptions, test cases (input–output pairs) for validation, correct/incorrect human solutions in multiple programming languages, and metadata including difficulty ratings, time/memory constraints, and competition source tags. For this benchmark implementation, we use Python, Java and C++ language solutions from the CodeContests dataset and perturb only the Python language solution. However, our perturbation-based methodology is general and can be applied to other programming languages and datasets, as long as they include correct solutions to the problems.

For each problem, we group the solutions that solve it into a *cluster*. We denote a cluster of solutions for a given problem i as P_i. Such a cluster P_i may contain multiple solutions, including those that are correct, incorrect, syntactically different but functionally equivalent, or even identical. The entire collection of these clusters, across all N problems, is denoted $\mathcal{P} = \{P_1, \ldots, P_N\}$.

3.2 Perturbation Pipeline

Our pipeline begins by sampling an initial pair of human-written programs, (s_1, s_2), from $\mathcal{P}$ using the SAMPLEPAIR function (see Algorithm 1). This function is designed to select pairs that reflect diverse initial relationships—such as being identical,

[1] https://github.com/google-deepmind/code_contests

[2] https://codeforces.com/

functionally equivalent (but syntactically different), one correct and one incorrect solution to the same problem, or correct solutions to entirely different problems (these initial relationships are further detailed in the dataset categorization Section 3.4).

Once an initial pair of programs is selected, the algorithm applies a sequence of k perturbations ($1 \leq k \leq K_{\max}$). Before applying any perturbations, the algorithm randomly decides whether the full sequence should preserve the original semantic relationship between the program pair or introduce a semantic break. To preserve semantics, all k perturbations must be semantic-preserving. To break semantics, at least one of the k perturbations must be semantic-non-preserving, with the remainder (if any) drawn from the semantic-preserving variants. At each step, one of the two programs (s_1 or s_2) is randomly selected for modification. A perturbation type is then randomly selected from the set of available types described in the types of perturbation section 3.3, and the corresponding variant—semantic-preserving or semantic-non-preserving—is applied based on the overall goal for the sequence. Algorithm 1 formalizes the procedure for generating perturbed code pairs via controlled semantic modifications.

3.3 Types of Perturbations

We design seven distinct perturbation types to generate controlled code variations, each with *semantic-preserving* and *semantic-non-preserving* variants that simulate both benign refactorings and subtle logic-breaking edits. The semantic-preserving variants are theoretically behaviour-preserving transformations, empirically validated by executing the perturbed code against all test cases from the original problem; instances that fail validation are discarded. This perturbation set thus enables structured control over code semantics, facilitating fine-grained evaluation of LLMs' ability to distinguish syntactic from semantic changes. Table 1 summarizes all perturbation types, with detailed explanations and examples in Appendix A.1.‘

3.4 Dataset Categorization:

Each resulting pair from Algorithm 1 assigned a structured label $\langle \textit{prefix}, k, \textit{flag} \rangle$. The *prefix* encodes the initial relationship of the pair before any perturbations were applied. The value k is the total number of perturbations that were actually applied. The *flag* indicates the net semantic effect of the perturbations. The following are the four possible

Algorithm 1 Perturbed Program Pair Generation

Require: A set of solution clusters $\mathcal{P} = \{P_1, \ldots, P_N\}$ with $P_i = \{s_{i1}, \ldots, s_{i|P_i|}\}$; perturbation budget $K_{\max}$, `max_tries`

$\{s_1, s_2\} \leftarrow$ SAMPLEPAIR$(\mathcal{P})$ ▷ With probability p_{same} both from one P_i, else from different clusters
$k \sim \text{Uniform}(1, K_{\max})$ ▷ Number of perturbations to apply.
preserve $\sim \text{Bernoulli}(0.5)$ ▷ Decide if the k-step perturbation sequence should be net semantic-preserving.
if not preserve **then**
  $k_{\text{neg}} \sim \text{Uniform}(1, k)$ ▷ Number of semantic-non-preserving perturbations.
else
  $k_{\text{neg}} \leftarrow 0$
end if
for $j = 1$ to $k - k_{\text{neg}}$ **do**
  $\hat{s} \leftarrow$ RANDOMCHOICE$(\{s_1, s_2\})$
  Apply to $\hat{s}$: the *semantic-preserving version* of a randomly chosen perturbation type. Retry up to `max_tries` on failure.
end for
$\hat{s} \leftarrow$ RANDOMCHOICE$(\{s_1, s_2\})$
for $j = 1$ to k_{neg} **do**
  Apply to $\hat{s}$: the *semantic-non-preserving version* of a randomly chosen perturbation type. Retry up to `max_tries` on failure.
end for
return Perturbed solution pair

values of the *prefix*:

`id`	identical correct submissions
`fe`	functionally equivalent but syntactically different solutions
`ne`	one correct, one incorrect submission
`di`	correct submissions of *different* problems

Example: A pair starting from correct solutions to *different problems* (`di`), then perturbed twice ($k = 2$), with one of those perturbations being a semantic-non-preserving ($k_{\text{neg}} = 1$), would be labeled $\langle di, 2, \text{np} \rangle$.

Our benchmark results in program pairs classified into *eight categories* by their structured labels (reflecting four initial relationship types combined with two overall semantic impact flags). Table 8 in Appendix A summarises the distribution of original and perturbed program pairs across the eight structured categories. For the benchmark test set, we

Table 1: Summary of Perturbation Types. Each type has a semantic-preserving (SP) variant that maintains program logic and a semantic-non-preserving (NP) variant that intentionally breaks it.

Perturbation Type	Semantic-Preserving Variant	Semantic-Non-Preserving Variant
If-Else Swapping	Negates the `if` condition and swaps the `if`/`else` blocks.	Negates the `if` condition but does not swap the blocks.
For-While Swapping	Correctly converts a `for` loop to an equivalent `while` loop.	Converts to a `while` loop with an intentional error (e.g., faulty iterator update).
If Condition Flipping	Rewrites the `if` condition to a logically equivalent form (e.g., $a < b$ to `not` $(a \geq b)$).	Inverts the relational operator in the condition (e.g., $a < b$ to $a \geq b$).
Variable Renaming	Consistently renames a variable across its entire scope.	Renames a variable inconsistently, breaking its semantic link in one or more places.
Boolean Inversion	Inverts a boolean variable's declaration and all its subsequent uses.	Inverts the variable's declaration but leaves its subsequent uses unchanged.
Statement Reordering	Swaps two statements that are data-independent.	Swaps two data-dependent statements, altering the execution logic.
Expression Reformatting	Refactors a sub-expression into a temporary variable without changing the final result.	Incorrectly refactors an expression, altering the final computational result.

chose K_{max} = 5. Hence, k ranges from 1 through 5. For more discussion on the choice of sample count, please refer to D.1. CETBenchenables a comprehensive and fine-grained assessment of large language models under controlled semantic variation, facilitating systematic analysis of model robustness and failure modes across different solution types and perturbation impacts.

4 Experiments

4.1 Methodology

To observe the effect of perturbation on LLM's performance in equivalence checking, we curate two parallel sets of code pairs. One set contains the categorised perturbed code pairs generated using our dataset generation method, while the other contains the corresponding original pairs prior to perturbation. For benchmarking, we use problems with a difficulty level of 900 from the CodeContests dataset. We prompt various LLMs to check whether the codes in a given pair are semantically equivalent for both sets. Detailed description of the prompts is present in the Appendix A. The LLMs used in the experiments are Qwen: Qwen2.5-Coder-7B-Instruct, Mistral: Mistral-7B-Instruct-v0.3, DSC: Deepseek-coder-6.7b-instruct, GPT-4o: GPT-4o-2024-08-06. We report accuracy and F1 scores as our evaluation metric.

4.2 Main Results

Table 2 shows the effect of perturbations on various models for the task of code similarity detection. There are four values corresponding to each category (id, fe, ne, and di). Within each row, the left column corresponds to non-semantic-preserving perturbations, and the right column to semantic-preserving perturbations. Overall accuracy and F1 scores are computed by aggregating results across all four categories, for both the original (top row) and perturbed (bottom row) code pairs. We observe a significant decrease in overall performance across 4 different LLMs. We observe that for the *id* category - i.e. pairs of identical code, all models have a significant drop in performance after perturbation, while having nearly 100% accuracy on the corresponding unperturbed set. Moreover, a similar trend is observed for *fe* category - i.e. functionally equivalent original pairs. This suggests that models severely struggle to tell similar-looking code as inequivalent on adding simple perturbations. This drop in performance is very surprising given how easy it is for humans to spot semantically non-preserving differences in a pair of otherwise identical code. We observe proprietary model GPT-4o being able to gain an advantage in identifying inequivalent pairs especially for ne category, by being able to spot cases of semantic inequivalence. However, it still remains vulnerable in the positive class. Both these effects tend to counterbalance each other, keeping the weighted accuracy nearly stable, though there is a significant decrease in F1 for the positive class.

4.3 Ablation Analysis

We conduct a comprehensive analysis to understand how various models respond to three key factors: the presence of Chain-of-Thought (CoT) reasoning, different difficulty levels of the problem, and the type of perturbation applied.

4.3.1 Effect of Chain of Thought Prompt

From table 3, we observe that asking the model to first generate a CoT of reasoning before concluding to the final answers helps weaker models like Mistral, Qwen significantly. We observe significant boost in their performance in both original pairs as well as the perturbed pairs as compared to prompting without chain of thought. However, we observe a drop in the performance of GPT-4o in perturbed

Model	id		fe		ne		di		Avg	W-Avg	Mac-F1	Mic-F1	PosF1	NegF1
	np	sp	np	sp	np	sp	np	sp						
Qwen	100.0	99.0	80.7	78.7	44.0	42.9	100.0	100.0	80.6	80.6	81.2	81.2	72.5	66.8
	30.5	98.0	39.7	76.9	61.4	47.6	99.5	100.0	69.2	69.2	78.7	78.7	69.2	78.7
Mistral	100.0	100.0	90.3	90.3	25.2	24.7	86.5	89.5	76.3	76.3	82.6	65.1	71.7	58.4
	0.2	98.0	16.5	86.5	33.0	27.8	90.0	89.0	56.6	68.3	64.4	49.9	44.8	54.9
DSC	100.0	100.0	100.0	99.9	0.1	0.3	2.5	2.0	50.6	50.6	51.1	32.5	63.1	1.9
	0.0	99.5	0.0	99.9	1.2	0.3	2.5	2.5	23.6	50.6	27.8	20.2	37.6	2.8
GPT-4o	99.5	100.0	72.5	73.0	67.7	66.7	100.0	100.0	84.9	84.9	91.1	76.8	75.5	78.1
	81.0	97.0	77.4	71.8	91.8	72.2	100.0	100.0	86.4	85.7	92.3	77.3	66.7	88.1

Table 2: Results comparing performance of various models before and after perturbations. First 10 columns show accuracy values. W-Avg is average accuracy after re-weighing examples to have equal weight on positively and negatively labelled examples. Macro F1, Micro F1, and F1 scores of each individual labels are also reported.

pairs. This signifies the brittleness of reasoning for equivalence on the generated perturbations.

4.3.2 Error Analysis

On human evaluation of the Chain of thought generations, it is observed that for original pairs of programs, LLMs struggle to correctly identify that two different algorithms are solving the same problem and hence are equivalent. For many of the $\langle fe, k, np \rangle$ samples where the model correctly generates the final answer as no, its reasoning is flawed - incorrectly stating that the difference in the algorithm leads to inequivalent behaviour. The model fails to focus on the perturbation and the error caused by it in the intended algorithm. Apart from this, a significant number of correct final generations have missing reasoning details despite explicit instructions in the prompt to add them. For detailed results and sample generation examples, please refer to the Appendix F.

4.3.3 Increasing Problem Difficulty Level

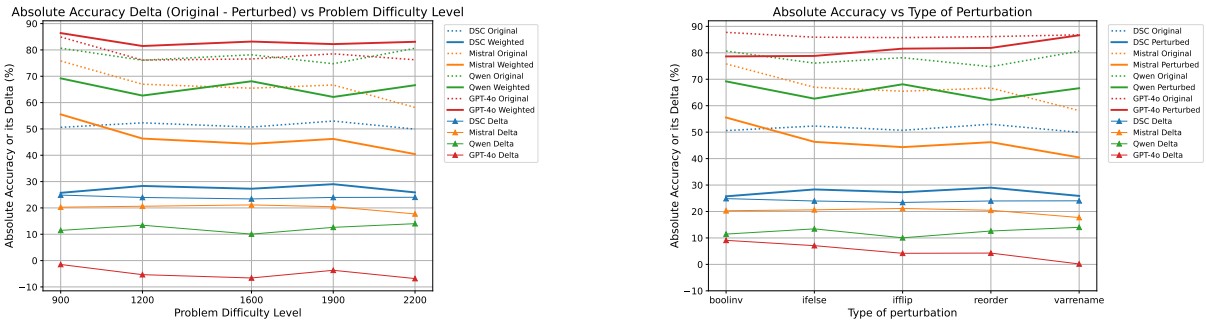
We also investigate how LLM model performance varies with increasing problem difficulty, using levels 900, 1200, 1600, 1900, and 2200, which represent a progression from simple, beginner-level problems to complex tasks requiring advanced algorithmic reasoning. These levels correspond to Codeforces problem ratings, a widely used metric in competitive programming. As shown in Fig 1a, there is a drop in accuracy—both before and after-perturbation across difficulty levels. The drop, however, remains constant across difficulty levels, which suggests that whether the problem is basic or complex, introducing perturbations impacts the model's reasoning in a similar way. For GPT-4o, we observe fluctuations in performance. Further, we note that while the drop for GPT-4o in terms of accuracy is negative, it is primarily due to high performance on ne and di categories. For closer analysis, we refer to the corresponding graph in Appendix B, plotting weighted Acc vs difficulty level, where we see that the drop is always positive across difficulty levels, pointing to nuanced performance behaviour of highly complex models.

4.3.4 Effect of Individual Perturbations

We also analysed how different types of perturbations impact the model by comparing the performance drop (delta) before and after applying each perturbation. We chose 5 perturbation types for which dataset generation was the quickest. Fig 1b shows that logic-altering perturbations like boolean inversion, if condition flipping, and if-else swapping have a slightly higher delta, indicating that the model struggles to understand the transformed logic. In contrast, surface-level changes such as variable renaming and statement reordering are comparatively easier for the models to interpret, resulting in comparatively slightly lower delta values. On observing GPT-4o's delta plot, we observe that it struggles on various perturbations except the Variable Renaming type of perturbation. We hypothesised that the presence of a relatively easy-to-spot perturbation is helping the model to label inequivalent pairs correctly. To test this hypothesis, we generated a separate test set consisting of all the types of perturbations except Variable Renaming. In table 4, the results of GPT-4o on this set show a significant decrease in the $\langle fe, k, \text{np} \rangle$ category and a higher decrease in the $\langle id, k, \text{np} \rangle$ category, unlike in table 2. And we now observe a drop in GPT-4o's average accuracy for a combination of different types of perturbations, similar to individual perturbations, pinning down its weakness.

Model	id		fe		ne		di		Avg	W-Avg	Mac-F1	Mic-F1	PosF1	NegF1
	np	sp	np	sp	np	sp	np	sp						
Qwen	100.0	98.5	56.8	57.7	54.5	57.5	99.0	100.0	79.6	79.6	87.3	69.0	67.6	70.4
	24.1	85.7	43.3	58.9	63.5	59.6	99.5	99.5	66.7	68.6	77.2	58.0	77.2	43.9
Mistral	100.0	98.9	56.8	57.7	63.7	64.5	96.0	93.0	78.8	78.8	86.9	68.4	65.5	71.5
	1.57	94.9	39.6	59.6	59.0	60.2	93.0	89.5	62.1	67.2	71.5	55.9	42.8	68.9
DSC	98.9	98.7	87.2	87.2	23.8	24.0	89.0	86.0	74.3	74.3	81.1	62.4	69.4	55.5
	16.7	92.3	18.8	85.2	30.4	26.0	92.0	89.5	56.3	48.5	65.8	67.1	43.8	53.2
GPT-4o	100.0	100.0	76.2	76.3	60.5	61.1	100.0	100.0	84.3	84.3	90.5	75.7	75.3	76.2
	65.5	99.0	47.6	74.6	74.0	66.4	100.0	100.0	78.3	81.1	86.6	68.0	56.6	79.5

Table 3: Results comparing various models using COT prompt



(a) Accuracy vs Problem Difficulty

(b) Accuracy vs Perturbation Type

Figure 1: Change in Accuracy with varying Difficulty Level and varying Perturbation Types for different Models

4.4 Finetuning

We fine-tuned multiple large language models (LLMs) using three distinct data configurations: the original clean dataset, a perturbed version of the dataset, and a balanced 50:50 mixture of both. To assess performance, we evaluated each model against two test sets: one comprising unseen, unperturbed programs and another comprising perturbed programs. The models trained on the combined 50:50 dataset consistently achieved the best results (shown in table 5) for each LLM. These optimal models not only demonstrated improved performance on both the unperturbed and perturbed test sets but also reached a level of performance comparable to GPT-4o. Further experimental details are available in Appendix A.

4.5 Cross-lingual Equivalence

We extend our benchmark to evaluate cross-lingual program equivalence. For this, we created a test set of Java-Python code pairs of the same size as our monolingual sets. In this cross-lingual setting, we apply perturbations only to the Python programs. We then compare the performance of LLMs on perturbed pairs and their corresponding original versions (unperturbed pairs). As shown in Table 6, the LLMs find it more difficult to verify equivalence for unperturbed cross-lingual pairs than for their monolingual counterparts. Furthermore, perturbing the Python program leads to an additional drop in performance, mirroring the trend observed in the monolingual setting. Similar results were observed for C++ and Python pairs, with detailed results available in Appendix B.

4.6 Transferability of Finetuned Models

To determine if the skills learned during fine-tuning generalize, we evaluated models in a zero-shot transfer setting. Models fine-tuned on 900-level Python-Python pairs were tested without further training on: (1) Python programs of higher difficulty levels, and (2) cross-lingual code pairs. Table 7a shows that finetuned models exhibit stable improvements across difficulty levels, indicating effective knowledge transfer. Similarly, Table 7b demonstrates consistent accuracy gains across Python-Python, Python-Java, and Python-C++ pairs, where the perturbation is always applied to a single Python program. Notably, models with initially poor performance improved the most, highlighting the benefits of our fine-tuning in challenging or noisy settings.

Model	id		fe		ne		di		Avg	W-Avg	Mac-F1	Mic-F1	PosF1	NegF1
	np	sp	np	sp	np	sp	np	sp						
GPT-4o	100.0	100.0	81.0	78.0	67.0	67.0	100.0	100.0	86.6	86.6	92.2	89.0	87.0	90.1
	75.5	94.0	65.5	80.5	90.5	71.0	100.0	100.0	84.7	85.6	91.2	83.1	74.1	92.0

Table 4: Results on GPT-4o on combinations of all other perturbations except variable-renaming

Model	id		fe		ne		di		Avg	W-Avg	Mac-F1	Mic-F1	PosF1	NegF1
	np	sp	np	sp	np	sp	np	sp						
Qwen	98.0	97.0	81.0	81.5	46.3	45.1	99.5	99.0	80.9	80.9	87.6	71.5	73.1	69.8
	82.7	96.5	82.2	90.1	85.3	39.4	99.5	97.5	84.2	87.3	90.1	75.8	66.9	84.8
Mistral	100.0	99.5	63.6	63.7	61.5	59.3	99.0	98.5	80.6	80.6	88.0	65.7	87.7	64.7
	77.0	87.5	76.7	67.6	87.5	55.6	100.0	98.0	81.2	80.0	88.9	71.2	58.5	83.9
DSC	100.0	100.0	84.1	83.9	42.6	41.0	99.5	100.0	81.3	81.3	87.5	71.2	73.6	68.7
	89.0	91.41	84.2	88.9	87.4	39.0	100.0	98.5	84.8	86.5	90.4	76.2	67.0	85.4

Table 5: Effect of finetuning

Model	fe		ne		di		Avg	W-Avg	Mac-F1	Mic-F1	PosF1	NegF1
	np	sp	np	sp	np	sp						
Qwen	87.5	87.8	33.4	33.9	100.0	99.5	73.7	77.2	81.2	68.3	72.1	64.4
	46.7	82.8	65.7	41.3	100.0	100.0	72.8	75.0	82.0	63.0	52.8	73.2
Mistral	46.7	82.8	63.7	64.5	96.0	93.0	78.8	78.8	86.9	68.4	65.5	71.5
	39.6	59.6	59.0	60.2	93.0	89.5	62.1	67.8	71.5	55.9	64.0	68.9
DSC	99.7	99.3	0.9	0.8	3.5	2.5	34.5	50.7	35.8	33.1	63.3	2.9
	2.2	99.5	2.6	1.0	10.5	2.5	19.7	51.6	22.5	21.8	38.0	5.6
GPT-4o	68.0	64.7	74.8	76.6	100.0	100.0	80.7	76.9	88.6	74.6	71.1	78.1
	79.7	64.8	93.4	79.5	100.0	100.0	86.2	75.9	92.1	75.3	62.4	88.2

Table 6: Results comparing various models for pairs with one Java code and one Python code

Model	900		1200		1600		1900		2200	
	B	A	B	A	B	A	B	A	B	A
Qwen	80.6	80.9	76.1	74.9	78.1	77.5	74.8	76.8	80.6	80.4
	69.2	84.2	62.7	80.3	68.1	81.4	62.1	81.6	66.6	75.6
Mistral	76.3	80.6	67.0	84.3	65.5	79.4	66.7	77.5	58.2	77.3
	56.6	81.2	46.3	77.4	44.3	75.5	46.2	72.7	40.4	72.5
DSC	50.6	81.3	52.3	82.9	50.7	86.1	53.0	78.1	49.9	81.4
	23.6	84.8	28.3	84.1	27.3	82.7	29.0	81.6	25.9	79.6

(a) Transfer across problem difficulty levels

Model	Py-Py		Py-Java		Py-C++	
	B	A	B	A	B	A
Qwen	81.4	82.7	73.7	75.6	73.3	67.1
	74.2	83.6	72.8	77.0	74.3	78.6
Mistral	77.2	81.3	59.8	72.5	58.9	72.0
	57.4	79.7	51.5	79.6	48.2	77.2
DSC	50.6	81.8	34.5	75.3	34.7	75.5
	26.8	83.2	19.7	80.9	19.0	81.3

(b) Transfer across languages

Table 7: Testing finetuned model across difficulty levels and cross-lingual setting. **B**: Before, **A**: After Finetuning

5 Conclusion

We have introduced CETBench, a comprehensive benchmark for evaluating LLMs on the task of code-equivalence checking via controlled program transformations. Our experiments show that even simple perturbations applied to program code can significantly degrade the performance of state-of-the-art LLMs, highlighting their limitations in understanding functional semantics. CETBench's methodology is generalizable, supporting extension to multiple programming languages and transformation types, and enables deeper analysis of LLM capabilities. Fine-tuning with perturbed examples improves performance, suggesting room for enhancement through targeted supervision. These findings underscore that current LLMs rely heavily on surface patterns rather than true semantic understanding of code.

Limitations

While our analysis presents deep insights into the performance of LLMs on the task of code equivalence under various perturbations, several limitations remain. First, our evaluation is primarily restricted to a single programming language (Python), highlighting the need to explore other languages to assess the generality of observed trends. Second, although we have performed experiments on a diverse set of models, the findings may not hold true for other models or future versions of the current models.

The dataset uses a binary Yes/No labelling scheme for functional equivalence, which oversimplifies nuanced cases where equivalence may be partial, conditional, or input-dependent. Furthermore, while the transformation set covers a wide range of syntactic and semantic changes, it is not exhaustive. Real-world coding variations—such as advanced refactoring patterns, concurrency-related modifications, or differences in library usage—are not fully captured.

Finally, although fine-tuning leads to performance gains within the distribution of the dataset, it remains unclear how well these models generalise to unseen perturbation types or real-world scenarios that lack the synthetic structure of our benchmarks.

References

Ajmain I Alam, Palash R Roy, Farouq Al-Omari, Chanchal K Roy, Banani Roy, and Kevin A Schneider. 2023. Gptclonebench: A comprehensive benchmark of semantic clones and cross-language clones using gpt-3 model and semanticclonebench. In *2023 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 1–13. IEEE.

Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie J. Cai, Michael Terry, Quoc V. Le, and Charles Sutton. 2021. Program synthesis with large language models. *ArXiv*, abs/2108.07732.

Sahar Badihi, Yi Li, and Julia Rubin. 2021. Eqbench: A dataset of equivalent and non-equivalent program pairs. In *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*, pages 610–614. IEEE.

Ira D Baxter, Andrew Yahin, Leonardo Moura, Marcelo Sant'Anna, and Lorraine Bier. 1998. Clone detection using abstract syntax trees. In *Proceedings. International Conference on Software Maintenance (Cat. No. 98CB36272)*, pages 368–377. IEEE.

Saikat Chakraborty, Shuvendu Lahiri, Sarah Fakhoury, Akash Lal, Madanlal Musuvathi, Aseem Rastogi, Aditya Senthilnathan, Rahul Sharma, and Nikhil Swamy. 2023. Ranking llm-generated loop invariants for program verification. In *Findings of the Association for Computational Linguistics: EMNLP 2023*, pages 9164–9175.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Pondé, Jared Kaplan, Harrison Edwards, Yura Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, and 34 others. 2021. Evaluating large language models trained on code. *ArXiv*, abs/2107.03374.

Jacob Devlin, Jonathan Uesato, Surya Bhupatiraju, Rishabh Singh, Abdel-rahman Mohamed, and Pushmeet Kohli. 2017. Robustfill: Neural program learning under noisy i/o. In *International conference on machine learning*, pages 990–998. PMLR.

Yijiang Dong, Tiancheng Hu, and Nigel Collier. 2024. Can llm be a personalized judge? In *Findings of the Association for Computational Linguistics: EMNLP 2024*, pages 10126–10141.

Aram Ebtekar. 2019. How to interpret contest ratings. Codeforces Blog. Accessed: 2025-05-18.

Mark Gabel, Lingxiao Jiang, and Zhendong Su. 2008. Scalable detection of semantic clones. In *Proceedings of the 30th international conference on Software engineering*, pages 321–330.

Robert Goldblatt and Marcel Jackson. 2012. Well-structured program equivalence is highly undecidable. *ACM Transactions on Computational Logic (TOCL)*, 13(3):1–8.

Jiawei Gu, Xuhui Jiang, Zhichao Shi, Hexiang Tan, Xuehao Zhai, Chengjin Xu, Wei Li, Yinghan Shen, Shengjie Ma, Honghao Liu, and 1 others. 2024. A survey on llm-as-a-judge. *arXiv preprint arXiv:2411.15594*.

Junda He, Jieke Shi, Terry Yue Zhuo, Christoph Treude, Jiamou Sun, Zhenchang Xing, Xiaoning Du, and David Lo. 2025. From code to courtroom: Llms as the new software judges. *arXiv preprint arXiv:2503.02246*.

Haolin Jin, Linghan Huang, Haipeng Cai, Jun Yan, Bo Li, and Huaming Chen. 2025. From llms to llm-based agents for software engineering: A survey of current, challenges and future. *Preprint*, arXiv:2408.02479.

Heejung Kim, Yungbum Jung, Sunghun Kim, and Kwankeun Yi. 2011. Mecc: memory comparison-based clone detector. In *Proceedings of the 33rd International Conference on Software Engineering*, pages 301–310.

Jens Krinke and Chaiyong Ragkhitwetsagul. 2022. Bigclonebench considered harmful for machine learning. In *2022 IEEE 16th International Workshop on Software Clones (IWSC)*, pages 1–7. IEEE.

Claire Le Goues, Michael Pradel, and Abhik Roychoudhury. 2019. Automated program repair. *Communications of the ACM*, 62(12):56–65.

Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom, Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, Thomas Hubert, Peter Choy, Cyprien de, Masson d'Autume, Igor Babuschkin, Xinyun Chen, Po-Sen Huang, Johannes Welbl, and 11 others. 2022. Competition-level code generation with alphacode. *Science*, 378:1092 – 1097.

Bo Liu, Yanjie Jiang, Yuxia Zhang, Nan Niu, Guangjie Li, and Hui Liu. 2025. Exploring the potential of general purpose llms in automated software refactoring: an empirical study. *Automated Software Engineering*, 32(1):26.

Nickil Maveli, Antonio Vergari, and Shay B Cohen. 2025. What can large language models capture about code functional equivalence? In *Findings of the Association for Computational Linguistics: NAACL 2025*, pages 6865–6903, Albuquerque, New Mexico.

MikeMirzayanov. 2018. Codeforces: Problem difficulties. Codeforces Blog. Accessed: 2025-05-18.

Miguel Romero Rosas, Miguel Torres Sanchez, and Rudolf Eigenmann. 2024. Should ai optimize your code? a comparative study of current large language models versus classical optimizing compilers. *arXiv preprint arXiv:2406.12146*.

Baptiste Roziere, Marie-Anne Lachaux, Lowik Chanussot, and Guillaume Lample. 2020. Unsupervised translation of programming languages. *Advances in neural information processing systems*, 33:20601–20611.

Kathryn T Stolee, Sebastian Elbaum, and Daniel Dobos. 2014. Solving the search for source code. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 23(3):1–45.

Jeffrey Svajlenko, Judith F Islam, Iman Keivanloo, Chanchal K Roy, and Mohammad Mamun Mia. 2014. Towards a big data curated benchmark of inter-project code clones. In *2014 IEEE International Conference on Software Maintenance and Evolution*, pages 476–480. IEEE.

Weixi Tong and Tianyi Zhang. 2024. Codejudge: Evaluating code generation with large language models. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, pages 20032–20051.

Anjiang Wei, Jiannan Cao, Ran Li, Hongyu Chen, Yuhui Zhang, Ziheng Wang, Yaofeng Sun, Yuan Liu, Thiago SFX Teixeira, Diyi Yang, and 1 others. 2025. Equibench: Benchmarking code reasoning capabilities of large language models via equivalence checking. *arXiv preprint arXiv:2502.12466*.

Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zi Lin, Zhuohan Li, Dacheng Li, Eric P Xing, and 1 others. 2023. Judging llm-as-a-judge with mt-bench and chatbot arena. In *Proceedings of the 37th International Conference on Neural Information Processing Systems*, pages 46595–46623.

Terry Yue Zhuo. 2024. Ice-score: Instructing large language models to evaluate code. In *Findings of the Association for Computational Linguistics: EACL 2024*, pages 2232–2242.

A Experimental Details

To keep our results reproducible, we use these hyperparameters of LLMs in our experiments: We set sampling to False, temperature to 0. Hence, we report the final results using a single run of the inference experiments.

For finetuning, we used LoRA finetuning using A100 GPU. We finetuned for nearly 48 GPU hours and selected the model with the least validation loss. The training and validation datasets were generated by the same algorithm 1. We initially split the 900-level problems in the CodeContests dataset into non-overlapping sets of 8, 8 and 40 problems, respectively, for test, validation and train sets. We then use the correct and incorrect submissions to these sets of problems to get the solution clusters for sampling. We used 2000 examples of each of the 8 categories and their original pairs for training. And we used 200 examples of each of the categories in the validation set. We experimented with learning rates from $1e-3$ to $1e-6$, and experimented with different combinations of [q_proj, k_proj, v_proj, o_proj] layers for LoRA.

We made use of the **trl** [3] library for SFTTrainer for finetuning and **ast** [4] and **astor** [5] libraries for dataset perturbation. We would be publicly releasing the code, which includes the setting of seeds for any randomness used, along with setting the final seeds and hyperparameters used for reproducing the experiments.

The zero-shot prompt used in our experiments is shared in the following page. We also experimented with various few-shot prompting methods, but the performance of models degraded. We hypothesize that this is caused by a larger context that involves multiple long pieces of code pairs. In Table 21, we present the results of few shot prompt with 4 examples each of positive and negative labels given along with the instruction of zero-shot prompt. It is observed that models get bias towards positive label.

[3] https://github.com/huggingface/trl, https://huggingface.co/docs/trl/en/sft_trainer

[4] https://docs.python.org/3/library/ast.html

[5] https://astor.readthedocs.io/en/latest/

Original Category	Sample Count	Perturbed Category	Sample Count
id	400	$\langle id, k, \text{np}\rangle$	200
		$\langle id, k, \text{sp}\rangle$	200
fe	2000	$\langle fe, k, \text{np}\rangle$	1000
		$\langle fe, k, \text{sp}\rangle$	1000
ne	2000	$\langle ne, k, \text{np}\rangle$	1000
		$\langle ne, k, \text{sp}\rangle$	1000
di	400	$\langle di, k, \text{np}\rangle$	200
		$\langle di, k, \text{sp}\rangle$	200

Table 8: Sample Counts of original and perturbed program-pairs

A.1 Detailed explanation of types of perturbation

Given below are the implementation details and examples of the various types of perturbations:

1. If-Else Swapping: *Semantic-preserving:* The `if` condition is logically negated, and the `if/else` blocks are interchanged, maintaining the original logic. *Non-preserving:* The `if` condition is logically negated, but blocks are *not* swapped.

Listing 1: If-else swapping examples

```
# Original
if a < b:
    print("A")
else:
    print("B")

# --- Semantic-preserving ---
if not (a >= b):
    print("B")
else:
    print("A")

# --- Semantic-non-preserving ---
if not (a < b):
    print("A")
else:
    print("B")
```

2. For-While Swapping *Semantic-preserving:* Convert `for` loop to equivalent `while`. *Non-preserving:* Converted to a `while` loop with an intentional error, such as incorrect iterator initialization a flawed loop termination condition, or an improper iterator update.

Listing 2: For-while swapping examples

```
# Original
for i in range(10):
    print(i)

# --- Semantic-preserving ---
i = 0
while i < 10:
    print(i)
```

Basic Prompt

Given two programs, you must check if they solve the same problem - that is, for any input, they must produce identical outputs.
Your task is to answer only with one word: Yes or No.
No explanation, no extra words, no formatting.
Program 1: {prog1}
Program 2: {prog2}

COT Prompt

You are given two programs. Your task is to analyze whether they solve the same problem — meaning, for any given input, both programs must always produce identical outputs.

Start by reasoning step-by-step through the logic and structure of both programs. You must write your reasoning steps.

After your analysis, you must end your response with exactly one of the following:
FINAL_ANSWER_IS_YES
FINAL_ANSWER_IS_NO

Only use this exact format for the final answer, and do not write anything after it.

Program 1: {prog1}
Program 2: {prog2}

```python
n, m = map(int, input().split())
a = sorted(list(map(int, input().split())))
res = a[n-1] - a[0]
for i in range(m - n + 1):
    temp = a[i+n-1] - a[i]
    if temp < res: res = temp
print(res)
```

(a) Original Code

```python
shfxn, hkdgf = map(int, input().split())
a = sorted(list(map(int, input().split())))
res = a[shfxn - 1] - a[0]
for i in range(hkdgf - shfxn + 1):
    temp = a[i + shfxn - 1] - a[i]
    if temp < res:
        res = temp
print(res)
```

(b) Perturbed Pair program -1

```python
firig, ozsoi = map(int, input().split())
res = a[firig - 1] - a[0]
a = sorted(list(map(int, input().split())))
for i in range(ozsoi - firig + 1):
    temp = a[i + firig - 1] - a[i]
    if temp < res:
        res = temp
print(res)
```

(c) Perturbed Pair program -2

Figure 2: Example of perturbation on programs.

```
    i += 1

# --- Semantic-non-preserving ---
i = 1
while i < 10:
    print(i)
    i += 1
```

3. If Condition Flipping: *Semantic-preserving:* The condition of an `if` statement is transformed to an equivalent form (e.g., $a < b$ to $not(a >= b)$), leaving the body unchanged. *Non-preserving:* The `if` condition's relational operator is inverted (e.g., $a < b$ to $a >= b$) or the condition negated.

Listing 3: If flipping examples

```
# Original
if a < b:
    print("bigger")

# --- Semantic-preserving ---
if not (a >= b):
    print("bigger")

# --- Semantic-non-preserving ---
if a >= b:
    print("bigger")
```

4. Variable Renaming: *Semantic-preserving:* One or more randomly selected variables are consistently renamed to new, unique identifiers (e.g., random five-letter strings) across all their occurrences within their scope. *Non-preserving:* Similar to above, but for at least one selected variable, one of its occurrences is renamed to a different identifier than its other occurrences, effectively breaking its semantic link.

Listing 4: Variable renaming examples

```
# Original
x = 10
y = 15
if x > y:
    print("bigger")

# --- Semantic-preserving ---
abcde = 10
hogav = 15
if abcde > hogav:
    print("bigger")

# --- Semantic-non-preserving ---
abcde = 10
hogav = 15
if noagd > hogav:  # inconsistent
    renaming
    print("bigger")
```

5. Boolean Variable Inversion: *Semantic-preserving:* The initialization of a selected boolean variable is inverted (e.g., `True` to `False`). All subsequent usages of this variable in conditions are then modified to use its negation (e.g., `if(Flag)` becomes `if(not Flag)`). *Non-preserving:* The initialization of a boolean variable is inverted, but its subsequent usages are not altered to reflect this change.

Listing 5: Boolean Variable Inversion examples

```
# Original
Flag = True
if Flag:
    print("Go")

# --- Semantic-preserving ---
Flag = False
if not Flag:
    print("Go")

# --- Semantic-non-preserving ---
Flag = False
if Flag:
    print("Go")
```

6. Statement Reordering: *Semantic-preserving:* Two data-independent statements (whose relative execution order doesn't affect program outcome) are swapped. *Non-preserving:* Two data-dependent statements are swapped, potentially causing errors like using variables before definition or with stale values.

Listing 6: Statement Swap examples

```
# Original
a = 5
b = 0
print(2 * a)

# --- Semantic-preserving ---
a = 5
print(2 * a)
b = 0

# --- Semantic-non-preserving ---
print(2 * a)
b = 0
a = 5
```

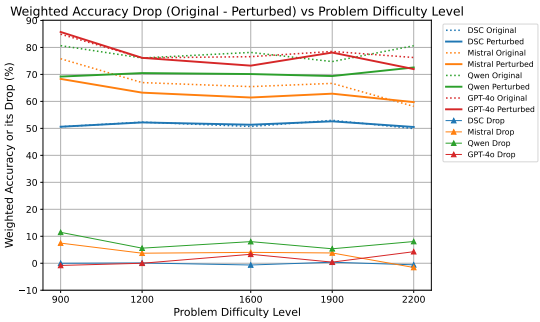
7. Expression Reformatting: *Semantic-preserving:* A sub-expression is computed and stored in a new temporary variable, which then replaces the sub-expression in the main expression, maintaining the original computation. *Non-preserving:* A temporary variable may be used, but the main expression is reconstructed incorrectly (e.g., wrong operands/operators, re-introducing original terms), altering the result.

Listing 7: Expression reformatting examples

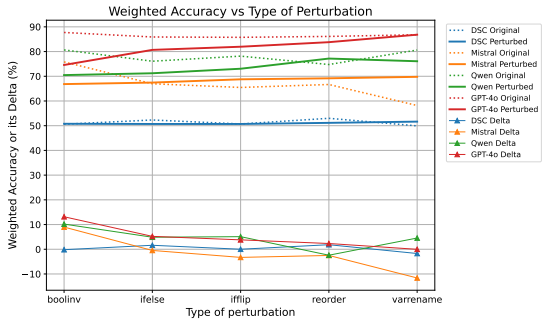
```
# Original
ans = a + b - c

# --- Semantic-preserving ---
temp = a + b
ans = temp - c

# --- Semantic-non-preserving ---
temp = a + b
ans = temp + b - c
```

B Extended Analysis

As we perturb the dataset to generate equal halves of the four original categories into sp and np, the previously positive samples from id and fe categories now have half of the samples as positive and the remaining half as negative. Due to this, in the final perturbed dataset, we have 75% negative samples and the remaining 25% as positive samples. We want the model to be equally good on both positives and negatives. Hence, we evaluate the performance on weighted accuracy, which takes the average of the average accuracy on positive samples and the same on negative samples. The figures 3a and 3b, contain the plots for this metric for the ablations of sections of 'Increasing Problem Difficulty Level' 4.3.3 and 4.3 'Effect of Individual Perturbations'. Similarly, figures 4a and 4b contain these plots for F1 metric. Tables 12 through 15 contain detailed results of analysis experiments. Table 17 contains the results of Python-C++ pairs of cross-lingual programs.

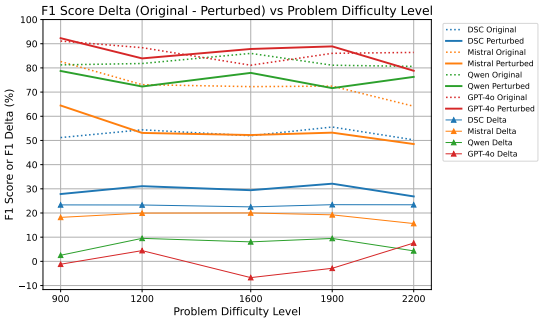


(a) Weighted Accuracy vs Difficulty

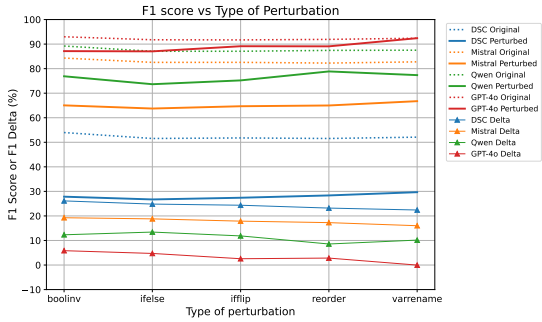


(b) Weighted Accuracy vs Perturbation Type

Figure 3: Change in Weighted accuracy with varying Difficulty Level and varying Perturbation Types for different Models



(a) F1 vs Problem Difficulty



(b) F1 score vs Perturbation Type

Figure 4: Change in Macro-F1 with varying Difficulty Level and varying Perturbation Types for different Models

Model	**id**		**fe**		**ne**		**di**		**Avg**	**W-Avg**	**Mac-F1**	**Mic-F1**	**PosF1**	**NegF1**
	np	**sp**	**np**	**sp**	**np**	**sp**	**np**	**sp**						
900	100.0	99.0	80.7	78.7	44.0	42.9	100.0	100.0	80.6	80.6	81.2	81.2	72.5	66.8
	30.5	98.0	39.7	76.9	61.4	47.6	99.5	100.0	69.2	69.2	78.7	78.7	69.2	78.7
1200	100.0	100.0	81.4	80.9	24.2	22.2	100.0	100.0	76.1	76.1	81.8	62.5	68.1	57.0
	26.0	95.0	32.5	77.3	42.1	28.4	100.0	100.0	62.6	70.5	72.3	53.2	45.0	61.5
1600	100.0	100.0	57.4	60.3	53.7	53.7	100.0	100.0	78.1	78.1	86.0	66.1	64.1	68.2
	21.0	92.5	53.4	55.9	64.9	57.1	100.0	100.0	68.1	70.1	77.9	59.0	44.4	73.6
1900	100.0	100.0	76.9	73.2	24.7	23.3	100.0	100.0	74.8	74.8	81.1	60.6	65.3	55.9
	30.5	98.0	39.7	76.9	61.4	47.6	99.5	100.0	62.1	69.4	71.6	52.8	43.1	62.4
2200	100.0	100.0	77.5	78.6	45.9	44.5	99.2	99.3	80.6	80.6	87.4	70.3	71.8	68.8
	19.0	96.0	39.1	73.0	59.6	51.1	96.5	98.6	66.6	72.6	76.3	59.3	48.4	70.2

Table 9: Effect of varying difficulty level on Qwen model.

Model	id		fe		ne		di		Avg	W-Avg	Mac-F1	Mic-F1	PosF1	NegF1
	np	sp	np	sp	np	sp	np	sp						
900	100.0	100.0	90.3	90.3	25.2	24.7	86.5	89.5	76.3	76.3	82.6	65.1	71.7	58.4
	0.2	98.0	16.5	86.5	33.0	27.8	90.0	89.0	56.6	68.3	64.4	49.9	44.8	54.9
1200	100.0	100.0	95.5	96.5	10.7	8.6	61.0	63.5	66.9	66.9	73.1	53.5	68.1	38.8
	1.5	99.0	4.5	95.1	16.2	12.9	74.0	67.5	46.3	63.2	53.1	39.4	42.2	36.5
1600	98.9	98.4	96.1	96.0	10.8	11.1	58.5	54.0	65.5	65.5	72.3	52.6	67.5	37.7
	3.0	97.0	5.4	94.3	16.1	14.4	63.5	61.0	44.3	61.4	52.2	38.3	41.5	35.0
1900	99.5	99.5	96.5	94.8	8.0	7.6	64.5	63.0	66.7	66.7	72.5	52.5	67.5	37.5
	2.0	99.5	7.4	92.8	13.4	12.6	70.5	71.5	46.2	62.8	53.2	39.0	41.6	36.4
2200	99.5	100.0	99.5	99.3	13.0	13.0	19.7	21.9	58.2	58.2	64.1	46.4	66.7	26.2
	0.5	99.5	3.8	97.1	18.3	17.3	39.8	47.3	40.4	59.7	48.5	36.2	41.5	30.8

Table 10: Effect of varying difficulty level on Mistral model.

Model	id		fe		ne		di		Avg	W-Avg	Mac-F1	Mic-F1	PosF1	NegF1
	np	sp	np	sp	np	sp	np	sp						
900	100.0	100.0	100.0	99.9	0.1	0.3	2.5	2.0	50.6	50.6	51.1	32.5	63.1	1.9
	0.0	99.5	0.0	99.9	1.2	0.3	2.5	2.5	26.3	50.6	27.8	20.2	37.6	2.8
1200	100.0	99.5	99.8	99.7	0.1	0.4	9.0	10.0	52.3	52.3	54.4	34.8	63.9	5.6
	1.0	100.0	1.2	99.9	1.0	0.6	12.5	10.5	28.3	52.2	31.1	22.1	38.2	5.9
1600	98.9	98.4	99.6	99.7	0.7	0.3	3.0	5.0	50.7	50.7	52.0	33.1	63.2	3.0
	0.5	99.5	0.8	99.5	1.5	0.5	10.5	5.5	27.3	51.4	29.4	21.2	37.9	4.5
1900	99.6	100.0	99.6	99.7	0.4	0.7	11.0	13.0	53.0	53.0	55.5	35.7	64.1	7.3
	0.0	100.0	1.0	99.6	0.8	0.7	15.5	14.5	29.0	52.6	32.1	22.6	38.3	7.0
2200	98.5	99.5	100.0	100.0	0.0	0.0	1.1	0.0	49.9	49.9	50.2	31.8	63.1	0.4
	1.0	99.5	0.5	100.0	0.6	0.1	2.8	2.7	25.9	50.5	26.8	19.7	37.7	1.8

Table 11: Effect of varying difficulty level on Deepseek-Coder model.

Model	id		fe		ne		di		Avg	W-Avg	Mac-F1	Mic-F1	PosF1	NegF1
	np	sp	np	sp	np	sp	np	sp						
900	99.5	100.0	72.5	73.0	67.7	66.7	100.0	100.0	84.9	84.9	91.1	76.8	75.5	78.1
	81.0	97.0	77.4	71.8	91.8	72.2	100.0	100.0	86.4	85.7	92.3	77.3	66.66	88.1
1200	100.0	99.5	40.2	40.3	65.2	63.9	100.0	100.0	76.6	76.6	81.1	59.1	44.9	73.3
	78.5	92.5	86.2	38.4	87.7	68.6	100.0	100.0	81.4	76.1	88.4	66.0	47.7	84.4
1600	100.0	99.5	17.4	18.9	87.9	88.9	100.0	100.0	76.6	76.6	81.1	59.1	44.9	73.3
	76.5	88.0	96.4	18.8	97.1	90.8	100.0	100.0	83.2	73.2	87.8	64.2	40.7	87.6
1900	99.5	100.0	47.8	45.1	66.6	68.7	100.0	100.0	78.5	78.5	86.0	66.1	60.7	71.5
	71.0	98.0	81.6	41.8	92.5	72.8	100.0	100.0	82.2	78.1	88.9	68.2	51.1	85.3
2200	100.0	100.0	10.0	9.4	96.7	94.0	100.0	100.0	76.2	76.2	78.8	56.2	38.5	74.0
	72.0	91.0	97.4	8.5	99.4	96.4	100.0	100.0	83.1	72.0	86.4	60.8	33.5	88.0

Table 12: Effect of varying difficulty level on GPT-4o model.

Model	id		fe		ne		di		Avg	W-Avg	Mac-F1	Mic-F1	PosF1	NegF1
	np	sp	np	sp	np	sp	np	sp						
boolinv	100.0	99.5	77.3	76.3	51.3	56.9	99.5	100.0	82.6	82.6	89.2	73.4	73.6	73.1
	30.5	81.0	28.9	75.2	58.0	59.7	100.0	100.0	66.7	70.5	76.9	59.3	48.3	70.4
ifelse	99.5	100.0	80.2	78.0	48.1	44.2	100.0	100.0	80.4	80.4	87.1	69.9	71.7	68.1
	23.2	92.5	24.9	78.4	46.1	47.6	100.0	100.0	64.1	71.2	73.6	55.8	46.8	64.9
ifflip	99.5	100.0	80.5	79.8	42.8	41.8	100.0	100.0	80.5	80.5	87.1	70.0	72.0	68.0
	30.3	97.0	25.2	78.5	48.9	45.8	100.0	100.0	65.7	73.0	75.2	56.5	47.5	65.6
reorder	100.0	100.0	82.2	83.2	42.6	41.5	100.0	100.0	81.2	81.2	87.4	70.9	73.2	68.6
	25.5	100.0	45.0	83.2	63.3	42.7	100.0	100.0	70.0	77.2	78.9	62.8	53.3	72.3
varrename	100.0	100.0	82.5	83.3	42.4	42.1	100.0	100.0	81.3	81.3	87.5	71.0	73.3	68.7
	20.0	100.0	37.4	81.9	63.6	46.3	100.0	100.0	68.6	76.1	77.3	61.7	52.2	71.2

Table 13: Effect of different perturbations on Qwen model.

Model	id		fe		ne		di		Avg	W-Avg	Mac-F1	Mic-F1	PosF1	NegF1
	np	sp	np	sp	np	sp	np	sp						
boolinv	99.5	99.0	81.9	82.5	36.7	38.7	84.5	88.0	76.4	76.4	84.3	66.6	70.4	62.7
	1.5	96.0	20.1	80.0	37.6	37.3	88.5	89.0	56.2	66.8	65.0	50.2	43.8	56.6
ifelse	99.4	100.0	88.1	88.3	25.6	27.7	90.0	89.5	76.0	76.0	82.6	65.0	71.7	58.3
	0.5	94.0	14.9	86.9	27.6	31.1	93.5	94.5	55.9	67.4	63.7	48.7	44.3	53.1
ifflip	99.5	100.0	87.9	89.0	27.7	25.1	88.5	91.5	76.1	76.1	82.6	64.6	70.8	58.3
	30.3	97.0	25.2	78.5	48.9	45.8	100.0	100.0	56.8	68.8	64.7	48.8	44.6	53.0
reorder	100.0	100.0	88.8	87.8	24.8	26.7	89.0	89.5	75.8	75.8	82.3	64.1	70.6	57.6
	25.5	100.0	45.0	83.2	63.3	42.7	100.0	100.0	57.4	69.2	65.0	51.0	45.6	56.4
varrename	100.0	100.0	88.6	90.0	24.1	28.6	92.0	89.0	76.5	76.5	82.8	65.0	71.3	58.7
	0.4	100.0	22.0	84.5	37.2	35.1	94.5	91.0	58.5	69.8	66.7	51.9	45.9	57.9

Table 14: Effect of different perturbations on Mistral model.

Model	id		fe		ne		di		Avg	W-Avg	Mac-F1	Mic-F1	PosF1	NegF1
	np	sp	np	sp	np	sp	np	sp						
boolinv	100.0	99.5	99.7	99.3	1.9	1.3	5.0	9.0	52.0	52.0	54.0	34.9	63.7	6.1
	0.0	99.5	0.2	99.6	1.1	0.8	5.5	4.5	26.4	50.8	27.8	20.4	37.7	3.0
ifelse	99.5	100.0	99.7	99.8	0.6	0.5	3.0	2.5	50.7	50.7	51.5	32.8	63.2	2.3
	1.0	100.0	0.2	99.7	0.6	0.3	3.5	1.5	25.8	50.5	26.7	19.6	37.5	1.6
ifflip	99.5	100.0	99.8	99.9	0.6	0.4	2.5	4.0	50.8	50.8	51.8	32.9	63.3	2.5
	0.0	99.5	0.1	99.9	0.7	0.4	4.5	4.5	26.2	50.7	27.4	20.0	37.7	2.4
reorder	100.0	100.0	99.9	99.9	0.5	0.4	3.5	2.0	50.8	50.8	51.5	32.8	63.4	2.2
	0.0	100.0	0.1	99.9	2.2	0.4	7.5	3.0	26.7	51.1	28.3	20.9	37.9	3.8
varrename	100.0	100.0	99.8	100.0	0.8	0.4	4.0	3.5	51.1	51.1	52.1	33.2	63.5	2.9
	0.5	100.0	0.7	99.9	1.1	0.9	11.0	6.0	27.5	51.6	29.7	21.4	38.0	4.7

Table 15: Effect of different perturbations on Deepseek-Coder model.

Model	id		fe		ne		di		Avg	W-Avg	Mac-F1	Mic-F1	PosF1	NegF1
	np	sp	np	sp	np	sp	np	sp						
boolinv	100.0	100.0	72.4	71.8	80.4	77.2	100.0	100.0	87.7	87.7	93.0	80.9	78.9	82.9
	79.0	61.0	52.5	71.9	86.2	78.5	100.0	100.0	78.6	74.6	87.1	70.7	57.8	83.6
ifelse	100.0	100.0	75.9	75.6	66.1	69.6	100.0	100.0	85.9	85.9	91.7	78.2	77.2	79.3
	60.1	95.0	49.3	73.8	79.5	72.8	100.0	100.0	78.8	80.7	87.0	69.8	58.1	81.5
ifflip	100.0	100.0	73.8	72.6	70.1	70.1	99.5	100.0	85.8	85.8	91.7	78.0	76.6	79.5
	60.1	95.0	49.3	73.8	79.5	72.8	100.0	100.0	78.8	80.7	87.0	69.8	58.1	81.5
reorder	100.0	100.0	75.3	76.8	69.1	67.8	100.0	100.0	86.1	86.1	91.9	78.6	77.6	79.6
	71.0	98.0	81.6	41.8	92.5	72.8	100.0	100.0	81.8	83.8	89.1	74.5	63.8	85.3
varrename	100.0	100.0	75.6	77.3	72.0	69.5	100.0	100.0	86.8	86.8	92.4	79.6	78.5	80.8
	79.0	100.0	75.5	74.4	93.0	71.7	100.0	99.5	86.6	86.8	92.4	78.0	67.8	88.2

Table 16: Effect of different perturbations on GPT-4o model.

Model	fe		ne		di		Avg		W-Avg	Mac-F1	Mic-F1	PosF1	NegF1
	np	sp	np	sp	np	sp	np	sp					
Qwen	81.6	82.6	37.4	38.0	100.0	100.0	73.3		74.8	81.6	67.6	70.1	65.0
	52.5	74.9	70.9	47.8	100.0	100.0	74.3		72.2	83.7	51.6	75.6	63.6
Mistral	96.3	96.7	11.1	13.0	66.5	70.0	58.9		68.1	66.9	55.8	68.8	42.9
	0.8	95.6	17.9	14.5	76.5	77.0	48.2		64.6	57.0	41.9	43.2	40.6
DSC	99.9	99.6	0.9	0.6	3.0	4.0	34.7		50.9	36.0	33.2	63.4	3.0
	0.9	99.7	2.8	1.0	0.8	1.5	19.0		51.1	21.1	21.1	37.9	4.2
GPT-4o	62.0	59.0	80.5	79.2	100.0	100.0	80.0		74.4	88.0	73.1	68.2	78.0
	82.6	57.8	95.6	80.5	100.0	100.0	86.1		73.2	91.8	73.7	59.2	88.2

Table 17: Results comparing various models for C++ and Python pairs of code

B.1 Disclosing the information about the perturbations to models

We tried to prompt the model to check for equivalence, along with providing information on possible kinds of perturbations applied to the programs. We added the examples of both np and sp types of all 7 types of perturbations. The examples used are the same as Appendix A. Comparing the results in table 18 with table 2, we observe that adding this information hurts the performance of the GPT-4o model. It did not help the Qwen and the Deepseek-Coder models. It improves the performance of Mistral for categories with negative labels, but hurts its performance for categories with positive labels. Similarly, by comparing the results of table 19 to table 3, we observe that adding the information of perturbation to the Chain-of-Thought prompt hurts the performance of the Mistral and the Deepseek-Coder models. It doesn't improve GPT-4o's performance. It improves Qwen's performance for some of the negative categories, but the decrease in the positive categories and the di category hurts overall performance. This shows that overall, the models struggle on the perturbed set even after having additional information about the perturbations disclosed to them.

C Comparision with existing benchmarks

On the Equibench (Wei et al., 2025) dataset's split of python programs, we perform the experiment of asking LLMs to check programs for equivalence using our zero-shot prompt. Their dataset contains three categories: OJA, OJV and OJVA. The positive labelled pairs in OJA category are similar to our fe category before perturbation. Similarly, the negative labelled pairs in OJA category are similar to ne before perturbation. The positive and negative categories of OJV are similar to $\langle id, k, sp\rangle$ and $\langle id, k, np\rangle$ respectively. And the positive and negative categories of OJVA are similar to $\langle fe, k, sp\rangle$ and $\langle fe, k, np\rangle$ respectively. As observed on comparing results of Table 20 with Table2, the final perturbed pairs are more challenging in our dataset. Moreover, our perturbations show decrease in performance as compared to the original pairs, which isn't present for the positive labelled pairs of OJA to OJVA. For (Maveli et al., 2025), the final perturbed test set of code pairs isn't available.

Prompt with Perturbation Information

Given two programs, you must check if they solve the same problem - that is, for any input, they must produce identical outputs.
Information regarding the programs: One or both of the programs may be modified from their original form using 7 different categories of perturbations.
Each of the 7 categories has 2 subtypes of perturbations: semantic preserving or not preserving. Given below are the examples for each of them:

1. If-else swapping examples:
# Original
if $a < b$:
print("A")
else:
print("B")

# Semantic-preserving
if not ($a >= b$):
print("B")
else:
print("A")

# Semantic-non-preserving
if not ($a < b$):
print("A")

else:
print("B")

2. ...
...
...

The original pair of programs before perturbation may or may not be solving the same problem. Given all this information, now check if the final given pair solves the same problem or not.
Your task is to answer only with one word: Yes or No.
No explanation, no extra words, no formatting.

Program 1: {prog1}
Program 2: {prog2}

Model	id		fe		ne		di		Avg	W-Avg	Mac-F1	Mic-F1	PosF1	NegF1
	np	sp	np	sp	np	sp	np	sp						
Qwen	100.0	99.0	78.8	78.4	44.0	41.5	100.0	100.0	80.2	80.2	86.9	69.6	71.4	67.7
	13.5	99.0	38.1	74.4	60.0	47.8	100.0	100.0	66.6	73.3	75.4	59.4	48.8	69.9
Mistral	100.0	99.0	68.4	70.4	50.7	47.9	90.0	93.5	77.5	77.5	85.8	67.4	67.8	67.0
	15.5	86.5	46.5	58.8	65.6	60.7	95.5	98.5	66.0	68.1	76.1	58.6	44.4	72.7
DSC	100.0	98.5	99.8	99.6	0.4	0.3	0.5	0.5	49.9	50.0	50.3	31.9	63.1	0.8
	0.0	99.5	0.2	99.8	0.1	0.3	0.0	0.5	25.0	49.9	25.2	18.9	37.5	0.4
GPT-4o	100.0	100.0	79.9	79.3	54.3	56.0	100.0	100.0	83.7	83.7	89.9	74.9	75.3	74.5
	58.0	98.0	62.4	73.8	82.5	61.4	100.0	100.0	79.5	81.6	87.6	70.7	59.2	82.2

Table 18: Results of performance after disclosing information about perturbation to the prompts.

Model	id		fe		ne		di		Avg	W-Avg	Mac-F1	Mic-F1	PosF1	NegF1
	np	sp	np	sp	np	sp	np	sp						
Qwen	33.5	79.0	56.6	37.8	68.7	68.1	85.8	91.5	65.1	62.9	77.0	54.0	34.4	73.7
Mistral	5.5	90.5	14.7	85.4	28.7	24.9	76.0	76.0	50.2	63.0	60.0	44.8	42.0	47.6
DSC	25.5	44.0	14.3	36.9	18.9	17.5	33.5	43.5	29.3	33.0	44.1	25.0	19.1	30.9
GPT-4o	66.0	96.5	50.3	74.8	72.2	63.1	100.0	100.0	77.9	80.4	86.4	67.6	56.1	79.1

Table 19: Results on perturbed set after disclosing information about perturbation to the COT prompt

Model	OJA		OJV		OJVA	
	neg	pos	neg	pos	neg	pos
Qwen	39.0	83.5	36.0	99.5	41.0	83.5
Mistral	17.5	91.5	16.0	99.0	19.0	90.5
DSC	99.5	1.0	99.5	2.0	99.5	2.0

Table 20: Results on EquiBench dataset

Model	id		fe		ne		di		Avg	W-Avg	Mac-F1	Mic-F1	PosF1	NegF1
	np	sp	np	sp	np	sp	np	sp						
Qwen	100.0	98.5	100.0	100.0	0.0	0.1	0.0	0.5	49.9	49.9	50.0	31.7	63.1	0.2
	8.5	98.5	6.7	100.0	8.0	0.1	6.0	0.5	28.5	52.1	31.8	23.8	38.6	9.1
Mistral	99.5	99.0	99.4	99.3	0.3	1.0	0.0	0.5	49.9	49.9	50.3	32.0	63.0	1.1
	0.0	99.5	0.2	99.3	0.5	1.1	1.0	1.0	25.3	50.0	25.9	19.4	37.4	1.3
DSC	99.5	98.0	99.7	99.9	0.1	0.4	0.0	0.5	49.8	49.8	50.1	31.7	62.9	0.5
	0.5	98.0	0.3	100.0	0.1	0.0	0.0	0.5	24.9	49.6	25.2	18.9	37.4	0.3

Table 21: Few shot prompt

D Variance bounds

To analyse the variance in performance of models, we performed the following two experiments:

D.1 Variance on different original pairs, for the same set of programming problems

For this experiment, we generated four parallel sets of original and perturbed datasets. Here, we sampled the original pairs from the solution set of the same programming problems. We calculated the standard deviation in accuracies of the Qwen model on sets of original pairs, and as observed in table 22, we see negligible variance.

D.2 Variance on different perturbed pairs for the same original pairs

Here, we generated four parallel perturbed datasets by applying steps 2-17 of algorithm 1 four times on a sampled pair. We calculated the standard deviation in the accuracies of Qwen. We previously observed higher variance for pairs perturbed from fe and ne categories. Due to this, we increased their number of samples in the test set from 400 to 2000 in each category. The table 23 shows that the final variance is negligible compared to the drop in performance from the original pairs.

E Error bounds

For calculating the error bounds for each type of perturbation, we calculated the percentage of erroneous samples generated over a total number of perturbation attempts. The results for the same are in Table 24. The low values of error percent denote the efficiency of our perturbation techniques.

F Error Analysis

The sample programs of fe category are shown in Figures 5 and 6. The original pair of programs correctly solve the same problem: '670-A Holidays' of Codeforces[6]. The sample generation in Figure 7 (i.e. the Chain-of-Thought generation of Mistral on being given programs of Figure 5 (a) and 6 (a)) demonstrates how the model fails to identify that two programs are solving the same problem with different algorithms in spite of being able to identify each of the algorithms correctly. Similarly, Figure 8 shows how it still fails when semantic-preserving perturbations are applied to the pair of programs and Figure 9 shows how it fails when semantic non-preserving perturbations are applied. This time, it correctly identifies how the original (intended) programs are equivalent, but fails to identify that non-semantic preserving perturbations have made them not equivalent. Tables 28, 25, 26 and 27 show the percentage of times the final answer in Chain-of-Thought prompts changes or remains unchanged for a given pair of codes when applying respective (np or sp) perturbation, with the highlighted numbers depicting the number of times the model's answer is affected adversally due to the perturbation. Also, percentage values like 18.5 % of 'No' remaining unchanged for $\langle fe, k, np \rangle$ in table 28 show that 18.5 % fraction of accuracy of $\langle fe, k, np \rangle$ category on perturbed dataset could be due to the incorrect interpretation of the original pairs itself. Table 27 also contains the percentage of generations of Chain-of-Thought in which Deepseek-Coder gave a direct answer without providing any reason for it. This percentage was 0 for other models.

G Licence Details and Ethics

Please refer table 29 for details of Licences. The data used from CodeContests dataset contains code submissions of various users on codeforces platform. The userid or name of the owner of the submissions are not present in the dataset.

[6] https://codeforces.com/problemset/problem/670/A

P1:

```python
a = int(input())
z =a//7
x = a%7
if x ==0:print(z*2,z*2)
elif x<3:print(z*2,(z*2)+x)
elif x>5:print((z*2)+1,(z*2)+2)
else:print(z*2,(z*2)+2)
```

(a) Original Pair program -1

P1:

```python
a = int(input())
z = a // 7
x = a % 7
if not (x != 0):
    print(z * 2, z * 2)
elif x < 3:
    print(z * 2, z * 2 + x)
elif x <= 5:
    print(z * 2, z * 2 + 2)
else:
    print(z * 2 + 1, z * 2 + 2)
```

(b) SP Perturbed Pair program -1

P1:

```python
a = int(input())
z = a // 7
x = a % 7
if x == 0:
    print(z * 2, z * 2)
elif x >= 3:
    print(z * 2, z * 2 + x)
elif x <= 5:
    print(z * 2, z * 2 + 2)
else:
    print(z * 2 + 1, z * 2 + 2)
```

(c) NP Perturbed Pair program -1

Figure 5: Sample pairs (Program -1).

P2:

```python
def p1():
        n = int(input())
        for i in range(0,n,1234567):
                for j in range(0,n-i,123456):
                        if (n-i-j)%1234 == 0:
                                return 'YES'
        return 'NO'
def p2():
        n,m = [int(i) for i in input().split()]
        l = [int(i) for i in input().split()]
        for i in range(n):
                if m-i-1 < 0 :
                        return l[m-1]
                m = m-i-1
def p3():
        n = int(input())
        b = n//7*2+min(2,n%7)
        a = n//7*2
        if n == n//7*7+6:
                a = a+1
        print(a,b)
p3()
```

(a) Original Pair program -2

P2:

```python
def p1():
        n = int(input())
        for i in range(0,n,1234567):
                for j in range(0,n-i,123456):
                        if (n-i-j)%1234 == 0:
                                return 'YES'
        return 'NO'
def p2():
        n,m = [int(i) for i in input().split()]
        l = [int(i) for i in input().split()]
        for i in range(n):
                if m-i-1 < 0 :
                        return l[m-1]
                m = m-i-1
def p3():
        n = int(input())
        b = n//7*2+min(2,n%7)
        a = n//7*2
        if n == n//7*7+6:
                a = a+1
        print(a,b)
p3()
```

(b) SP Perturbed Pair program -2

P2:

```python
def p1():
        n = int(input())
        for i in range(0,n,1234567):
                for j in range(0,n-i,123456):
                        if (n-i-j)%1234 == 0:
                                return 'YES'
        return 'NO'
def p2():
        n,m = [int(i) for i in input().split()]
        l = [int(i) for i in input().split()]
        for i in range(n):
                if m-i-1 < 0 :
                        return l[m-1]
                m = m-i-1
def p3():
        n = int(input())
        b = n//7*2+min(2,n%7)
        a = n//7*2
        if n == n//7*7+6:
                a = a+1
        print(a,b)
p3()
```

(c) NP Perturbed Pair program -2

Figure 6: Sample pairs (Program -2).

LLM Output:

```
<s> Reasoning:

Program 1:
- The program takes an integer input 'a'.
- It calculates the quotient 'z' of 'a' divided by 7 and the remainder 'x' of 'a' divided by 7.
- Depending on the value of 'x', it prints two numbers.
- If 'x' is 0, it prints the square of 'z'.
- If 'x' is between 1 and 2 (inclusive), it prints the square of 'z' plus 'x'.
- If 'x' is between 4 and 6 (inclusive), it prints the square of 'z' plus 1 or 2, depending on the value of 'x'.
- If 'x' is 3 or 5, it prints the square of 'z' plus 2.

Program 2:
- The program defines three functions: p1, p2, and p3.
- Function p1 takes an integer input 'n'.
- It iterates through all possible values of 'i' from 0 to 'n' with a step of 1234567.
- For each 'i', it iterates through all possible values of 'j' from 0 to 'n-i' with a step of 123456.
- If the difference between 'n' and the sum of 'i' and 'j' is a multiple of 1234, it returns 'YES'.
- If no such 'i' and 'j' are found, it returns 'NO'.
- Function p2 takes two integers 'n' and 'm' as input.
- It iterates through all possible values of 'i' from 0 to 'n'.
- If 'm-i-1' is less than 0, it returns the value of 'l[m-1]'.
- If 'm-i-1' is not less than 0, it sets 'm' to 'm-i-1'.
- Function p3 takes an integer input 'n'.
- It calculates 'b' as the product of the quotient of 'n' divided by 7 plus 2, and the minimum of 2 and the remainder of 'n' divided by 7.
- It calculates 'a' as the product of the quotient of 'n' divided by 7 plus 2.
- If 'n' is equal to the product of the quotient of 'n' divided by 7 plus 7 and 6, it increments 'a' by 1.
- It prints 'a' and 'b'.
- The last line calls function p3 and prints the result.

Final_Answer_Is_No

The two programs do not solve the same problem. Program 1 calculates two numbers based on the remainder of an input number divided by 7, while program 2 checks if a number can be expressed as the sum of two numbers with specific conditions on the step sizes of the iterations. Program 2 also defines and uses additional functions that are not used in program 1.</s>
```

Figure 7: Sample generation of Mistral on original pair

Model	id	fe	ne	di
Unperturbed	0	4.77e-3	4.92e-2	0
NP Perturbed	4.02e-2	9.32e-2	5.66e-2	3.54e-3
SP Perturbed	6.86e-3	5.02e-2	6.78e-2	0

Table 22: Variance on changing original pairs

Model	id	fe	ne	di
NP Perturbed	8.92e-4	8.33e-4	4.35e-4	9.17e-5
SP Perturbed	3.22e-4	1.22e-4	1.21e-3	4.92e-4

Table 23: Variance on changing perturbed pairs

Perturbation type	error % in sp	error % in np
If-else swapping	3.86	11.68
For-while swapping	4.64	5.529
If condition flipping	0.12	4.37
Variable renaming	5.35	9.86
Boolean variable inversion	2.10	0.55
Statement reordering	1.93	8.99
Expression reformatting	2.88	3.33

Table 24: Error bounds in dataset generation

```
LLM Output:

<s> Reasoning:

Program 1:
- The program takes an integer input 'a'.
- It calculates the quotient 'z' of 'a' divided by 7 and the remainder 'x' of 'a' divided by 7.
- If the remainder is not 0, it checks if the remainder is less than 3, less than or equal to 5, or greater than 5. Depending on the range, it calculates a different output.
- If the remainder is 0, it always calculates the same output, which is twice the quotient.

Program 2:
- The program defines three functions: p1, p2, and p3.
- Function p1 takes an integer input 'n' and checks if there exist two numbers 'i' and 'j' such that 'i' is a multiple of 1234567, 'j' is a multiple of 123456, and 'n' can be expressed as the sum of 'i', 'j', and a multiple
of 1234. If such numbers exist, it returns 'YES'; otherwise, it returns 'NO'.
- Function p2 takes two integers 'n' and 'm' as input, as well as a list 'l'. It iterates through the list 'l' and returns the 'm'-th element of the list if 'm' is less than the length of the list; otherwise, it returns an
element from the list.
- Function p3 takes an integer input 'n' and calculates the output 'b' as the product of the quotient of 'n' divided by 7 and 2, plus the minimum of 2 and the remainder of 'n' divided by 7. It calculates the output 'a' as
the product of the quotient of 'n' divided by 7 and 2. If 'n' is a multiple of 49 (i.e., 'n' is a multiple of 7*7 plus 6), it increments 'a' by 1. It then prints 'a' and 'b'.
- The last line of the code calls function p3 and prints its output.

Since the logic and structure of both programs are different, they do not solve the same problem.

FINAL_ANSWER_IS_NO</s>
```

Figure 8: Sample generation of Mistral on SP perturbed pair

```
LLM Output:

<s>
FINAL_ANSWER_IS_YES

Explanation:

Both programs take an integer n as input, which represents the number of names to be entered.

Program 1:
1. It initializes an empty dictionary named 'names' to store the count of each unique first letter of the names.
2. It then enters a loop that runs n times. In each iteration, it takes a name as input, extracts the first letter of the name, and increments the count of that first letter in the 'names' dictionary if it already exists,
or initializes it to 1 if it doesn't.
3. After processing all the names, it initializes a counter 'count' to 0.
4. It then enters another loop that iterates over each key in the 'names' dictionary.
5. For each key, it checks if the count of that key is greater than or equal to 2. If so, it calculates the number of combinations of pairs of identical first letters in the names, depending on whether the count is even or
odd.
6. It then adds the calculated number of combinations to the 'count' variable.
7. Finally, it prints the value of the 'count' variable.

Program 2:
1. It initializes an empty dictionary named 'd' to store the count of each unique first letter of the names.
2. It then initializes the alphabet list 'alpha' with the lowercase letters from 'a' to 'z'.
3. It enters a loop that runs n times. In each iteration, it takes a name as input, extracts the first letter of the name, and increments the count of that first letter in the 'd' dictionary if it already exists, or
initializes it to 1 if it doesn't.
4. After processing all the names, it initializes a counter 'count' to 0.
5. It then enters another loop that iterates over each key in the 'alpha' list.
6. For each key, it calculates a list 'li' containing the number of pairs of identical first letters in the names that can be formed with the current key, considering both even and odd counts.
7. It then iterates over each element in the 'li' list and checks if the element is greater than or equal to 2. If so, it calculates the number of combinations of pairs of identical first letters in the names, depending on
the value of the element.
8. It then adds the calculated number of combinations to the 'count' variable.
9. Finally, it prints the value of the 'count' variable.

Both programs perform the same operations on the same input data, so they will always produce identical outputs for the same input. Therefore, the final answer is YES.</s>
```

Figure 9: Sample generation of Mistral on NP perturbed pair

Qwen's Generation	id		fe		ne		di	
	np	**sp**	**np**	**sp**	**np**	**sp**	**np**	**sp**
Original: Yes, Perturbed: Yes	**73.4**	82.1	**36.5**	36.6	15.4	16.8	0.0	0.0
Original: Yes, Perturbed: No	26.0	**14.3**	31.1	**25.7**	29.2	24.4	0.5	1.5
Original: No, Perturbed: Yes	0.4	3.6	18.8	23.1	**20.4**	**23.0**	**0.5**	**0.5**
Original: No, Perturbed: No	0.0	0.0	13.6	14.6	35.0	35.8	98.9	98.0

Table 25: Analysis of percentage of answers changed or unchanged by Qwen on the program pair before and after perturbation.

Mistral's Generation	id		fe		ne		di	
	np	**sp**	**np**	**sp**	**np**	**sp**	**np**	**sp**
Original: Yes, Perturbed: Yes	**98.5**	94.5	**38.7**	42.4	19.2	18.7	0.1	0.0
Original: Yes, Perturbed: No	1.5	**4.5**	18.6	**16.3**	16.3	15.6	3.0	2.6
Original: No, Perturbed: Yes	0.5	1.5	19.6	19.1	**20.5**	**20.7**	**6.0**	**9.9**
Original: No, Perturbed: No	0.0	0.5	22.9	22.2	43.9	45.0	90.0	87.4

Table 26: Analysis of percentage of answers changed or unchanged by Mistral on the program pair before and after perturbation.

DSC's Generation	id		fe		ne		di	
	np	**sp**	**np**	**sp**	**np**	**sp**	**np**	**sp**
Original: Yes, Perturbed: Yes	**83.0**	92.0	**73.3**	78.5	59.9	63.5	2.5	3.5
Original: Yes, Perturbed: No	16.0	**6.5**	14.0	**8.9**	16.3	12.5	8.0	10.5
Original: No, Perturbed: Yes	0.5	1.5	7.5	6.7	**9.6**	**10.5**	**5.5**	**6.5**
Original: No, Perturbed: No	0.5	0.0	5.1	5.8	14.2	13.5	83.9	79.5
Missing Reason	44.5	40.5	7.3	6.9	5.9	5.1	0.5	0.5

Table 27: Analysis of percentage of answers changed or unchanged by Deepseek-Coder on the program pair before and after perturbation.

GPT-4o's Generation	id		fe		ne		di	
	np	**sp**	**np**	**sp**	**np**	**sp**	**np**	**sp**
Original: Yes, Perturbed: Yes	**34.5**	99.0	**47.4**	66.4	21.3	27.9	0.0	0.0
Original: Yes, Perturbed: No	65.5	**1.0**	29.0	**1.0**	18.2	11.0	0.0	0.0
Original: No, Perturbed: Yes	0.0	0.0	5.1	8.2	**4.7**	**5.7**	0.0	0.0
Original: No, Perturbed: No	0.0	0.0	18.5	15.5	55.8	55.4	100.0	100.0

Table 28: Analysis of percentage of answers changed or unchanged by GPT-4o on the program pair before and after perturbation.

Model / Tool / Dataset	License
HuggingFace	Apache License, Version 2.0
DeepSeek-Coder	Code: MIT License; Model: DeepSeek License Agreement
Qwen	Tongyi Qianwen License Agreement
GPT-4o	Proprietary (OpenAI Terms of Use)
Mistral	Apache License, Version 2.0
CodeContests Dataset	CC BY 4.0 (Creative Commons Attribution 4.0)

Table 29: Licenses of the different models, tools, and datasets used in our experiments.