EquiBench: Benchmarking Large Language Models' Reasoning about Program Semantics via Equivalence Checking

Anjiang Wei[1]* **Jiannan Cao**[2] **Ran Li**[1,3] **Hongyu Chen**[4] **Yuhui Zhang**[1]
Ziheng Wang[1] **Yuan Liu**[3] **Thiago S. F. X. Teixeira**[5]
Diyi Yang[1] **Ke Wang**[4] **Alex Aiken**[1]

[1]Stanford University [2]MIT [3]Google [4]Nanjing University [5]Intel

Abstract

As large language models (LLMs) become integral to code-related tasks, a central question emerges: Do LLMs truly understand program semantics? We introduce EquiBench, a new benchmark for evaluating LLMs through equivalence checking, i.e., determining whether two programs produce identical outputs for all possible inputs. Unlike prior code generation benchmarks, this task directly tests a model's ability to reason about program semantics. EquiBench consists of 2400 program pairs across four languages and six categories. These pairs are generated through program analysis, compiler scheduling, and superoptimization, ensuring high-confidence labels, nontrivial difficulty, and full automation. We evaluate 19 state-of-the-art LLMs and find that in the most challenging categories, the best accuracies are 63.8% and 76.2%, only modestly above the 50% random baseline. Further analysis reveals that models often rely on syntactic similarity rather than exhibiting robust reasoning about program semantics, highlighting current limitations. Our code and dataset are publicly available at https://github.com/Anjiang-Wei/equibench

1 Introduction

Large language models (LLMs) have rapidly become central to software engineering workflows, powering tools for code generation, program repair, test case generation, debugging, and beyond, significantly boosting developers' productivity (Jain et al., 2024; Yang et al., 2024a, 2023). This surge of capability has prompted a natural yet fundamental question: Do LLMs merely mimic code syntax they have seen during training, or do they genuinely understand what programs do?

Unlike natural language, code is executable. Two programs may differ syntactically yet be semantically equivalent, producing identical outputs for all inputs. Conversely, programs with only minor syntactic differences can behave quite differently at runtime. This gap between surface-level program features and actual execution behavior raises an important question: Does training on static code corpora equip LLMs with a grounded understanding of program semantics?

To rigorously assess whether LLMs truly understand code, we need benchmarks that demand reasoning about program semantics. However, widely used coding benchmarks such as HumanEval (Chen et al., 2021) and MBPP (Austin et al., 2021) primarily test a model's ability to generate short code snippets from natural language descriptions, offering limited insight into whether the model grasps the underlying semantics of the code it generates.

In this work, we introduce **equivalence checking** as a new task for evaluating LLMs' ability to reason about program semantics. Unlike tasks based on syntactic similarity, equivalence checking asks whether two programs are semantically equivalent, i.e., whether they produce identical outputs for all possible inputs, regardless of how differently they are written. Program equivalence problems test directly whether and how well models reason about code. Any question about program semantics can be formulated as an equivalence checking problem, and program equivalence problems can have any level of difficulty from trivially easy to extremely difficult. Program equivalence is undecidable in general: no algorithm can determine program equivalence for all cases while guaranteeing termination. This fundamental theoretical impossibility underscores the intrinsic difficulty of our task.

Designing a benchmark for equivalence checking requires both equivalent and inequivalent program pairs spanning diverse categories, which poses several challenges in terms of *label soundness*, *problem difficulty*, and *automation*. First, it is difficult to guarantee high-confidence labels, as

*Correspondence to: anjiang@cs.stanford.edu

Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing, pages 33868–33881
November 4-9, 2025

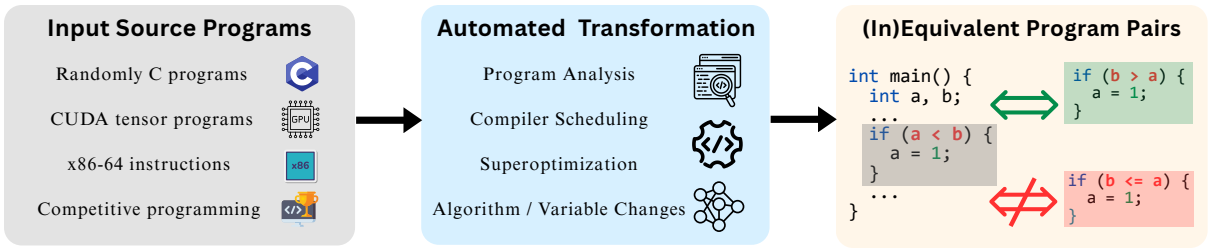

Figure 1: **Overview of EquiBench.** We construct (in)equivalent program pairs from diverse sources, including C and CUDA programs, x86-64 assembly, and competitive programming, using automated transformations based on program analysis, compiler scheduling, superoptimization, and changes in algorithms or variable names.

verifying equivalence by exhaustively executing all possible inputs is almost always computationally infeasible. Second, existing generation techniques rely on superficial syntactic edits (Badihi et al., 2021; Maveli et al., 2024), which are too simplistic to meaningfully challenge state-of-the-art LLMs and fail to probe their semantic reasoning limits. Third, to enable comprehensive evaluation, the benchmark must be large-scale and modular, necessitating a fully automated construction pipeline.

In this work, we introduce **EquiBench**, a dataset of 2400 program pairs for evaluating large language models on equivalence checking. Covering Python, C, CUDA, and x86-64 programs, it enables a systematic assessment of LLMs' ability to reason about program semantics.

As illustrated in Figure 1, EquiBench addresses these challenges by automatically constructing both equivalent and inequivalent program pairs from diverse input sources, including randomly generated C and CUDA code, assembly instructions, and competitive programming solutions. To ensure label soundness without exhaustive execution, we apply program transformation techniques grounded in program analysis and superoptimization. To increase problem difficulty beyond trivial edits, we incorporate structural transformations through compiler scheduling and algorithmic equivalences. The entire generation pipeline is fully automated, enabling scalable construction of a large and diverse benchmark. Finally, EquiBench is extensible to additional categories of equivalence checking problems, which we anticipate will be useful as LLMs improve.

Our experiments show that EquiBench is a challenging benchmark for LLMs. Among the 19 models evaluated, OpenAI o4-mini performs best overall, yet achieves only 60.8% in the CUDA category despite reaching the highest overall accuracy of 82.3%. In the two most difficult categories, the best accuracies are 63.8% and 76.2%, respectively, only modestly better than the random baseline of 50% for binary classification. In contrast, purely syntactic changes such as variable renaming are much easier, with accuracies as high as 96.5%. We further find, through difficulty analysis, that models often rely on superficial form features such as syntactic similarity rather than demonstrating robust semantic reasoning. Moreover, prompting strategies such as few-shot in-context learning and Chain-of-Thought (CoT) prompting barely improve LLM performance, underscoring the difficulty of reasoning about program semantics.

In summary, our contributions are as follows:

- **New Task and Dataset:** We introduce equivalence checking as a new task to assess LLMs' reasoning about program semantics. We present *EquiBench*, a benchmark for equivalence checking spanning four languages and six equivalence categories.
- **Automated Generation:** We develop a fully automated pipeline for constructing diverse (in)equivalent program pairs using techniques that ensure high-confidence labels and nontrivial difficulty. The pipeline covers transformations ranging from syntactic edits to structural modifications and algorithmic equivalence.
- **Evaluation and Analysis:** We evaluate 19 state-of-the-art models on EquiBench. In the two most challenging categories, the best accuracies are only 63.8% and 76.2%, highlighting fundamental limitations. Our analysis shows that models often rely on superficial form features rather than demonstrating robust reasoning about program semantics.

2 Related Work

LLM Reasoning Benchmarks Extensive research has evaluated LLMs' reasoning capabilities

across diverse tasks (Cobbe et al., 2021; Huang and Chang, 2022; Bubeck et al., 2023; Mirzadeh et al., 2024; Zhou et al., 2022; Ho et al., 2022; Wei et al., 2022; Chen et al., 2024; Clark et al., 2018; Zhang et al., 2024). In the context of code reasoning, i.e., predicting a program's execution behavior without running it, CRUXEval (Gu et al., 2024) focuses on input-output prediction, while CodeMind (Liu et al., 2024) extends evaluation to natural language specifications. Another line of work seeks to improve LLMs' code simulation abilities through prompting (La Malfa et al., 2024) or targeted training (Liu et al., 2023; Ni et al., 2024; Ding et al., 2024; Chen et al., 2025). Unlike prior work that tests LLMs on specific inputs, our benchmark evaluates their ability to reason over all inputs.

Equivalence Checking Equivalence checking underpins applications such as performance optimization (Shypula et al., 2023; Cummins et al., 2023, 2024), code transpilation (Lu et al., 2021; Yang et al., 2024b; Ibrahimzada et al., 2024; Pan et al., 2024), refactoring (Pailoor et al., 2024), and testing (Felsing et al., 2014; Tian et al., 2024). Due to its undecidable nature, no algorithm can decide program equivalence for all program pairs while always terminating. Existing techniques (Sharma et al., 2013; Dahiya and Bansal, 2017; Gupta et al., 2018; Mora et al., 2018; Churchill et al., 2019; Badihi et al., 2020) focus on specific domains, such as SQL query equivalence (Zhao et al., 2023; Ding et al., 2023; Singh and Bedathur, 2024). EQBENCH (Badihi et al., 2021) and SeqCoBench (Maveli et al., 2024) are the main datasets for equivalence checking, but have limitations. EQBENCH is too small (272 pairs) for LLM evaluation, while SeqCoBench relies only on statement-level syntactic changes (e.g., renaming variables). In contrast, our work introduces a broader set of equivalence categories, creating a more systematic and challenging benchmark.

3 Benchmark Construction

While we have so far discussed the standard notion of equivalence, namely that two programs produce the same output on any input, each benchmark category adopts a more precise definition tailored to its domain. All follow the principle of "producing the same output given the same input," but the exact criteria differ. For example, the CUDA category tolerates small discrepancies from floating-point rounding rather than requiring strict bit-level

```
char b[2];
static int c = 0;

int main() {
  char* p1 = &b[0];
  int*  p2 = &c;
  ...
  if (p1 == p2) {
     // dead code
     c = 1;
  }
  return 0;
}
```

```
char b[2];
static int c = 0;

int main() {
  char* p1 = &b[0];
  int*  p2 = &c;
  ...
  if (p1 == p2) {
   // code eliminated
  }
  return 0;
}
```

Figure 2: **An equivalent pair from the DCE category in EquiBench.** In the left program, `c = 1` is dead code that has no effect on the program state and is removed in the right program. Such pairs are generated using the Dead Code Elimination (DCE) pass in compilers.

equivalence. These definitions are grounded in real-world use cases and chosen to capture practical notions of equivalence in each setting. For each category, we provide the corresponding definition in the prompt when testing LLM reasoning. We describe how we generate (in)equivalent pairs across the six categories as follows:

- **DCE**: C program pairs generated via the compiler's dead code elimination (DCE) pass (Section 3.1).
- **CUDA**: CUDA program pairs created by applying different scheduling strategies using a tensor compiler (Section 3.2).
- **x86-64**: x86-64 assembly program pairs generated by a superoptimizer (Section 3.3).
- **OJ_A**, **OJ_V**, **OJ_VA**: Python program pairs from online judge submissions, featuring algorithmic differences (OJ_A), variable-renaming transformations (OJ_V), and combinations of both (OJ_VA) (Section 3.4).

3.1 Pairs from Program Analysis (DCE)

Dead code elimination (DCE), a compiler pass, removes useless program statements. After DCE, the remaining statements in the modified program naturally *correspond* to those in the original program.

Definition of Equivalence. Two programs are considered equivalent if, when executed on the same input, they *always* have identical *program states* at all corresponding points reachable by program execution. We expect language models to

```cuda
__global__ void GEMV(const float* A,
                     const float* x,
                     float* y,
                     int R,
                     int C) {

    // Calculate the row index
    // assigned to the thread
    int r = blockIdx.x * blockDim.x
            + threadIdx.x;

    // Return if out of bounds
    if (r >= R) return;
    float s = 0.0f;

    for (int c = 0; c < C; c++) {
        s += A[r * C + c] * x[c];
    }

    y[r] = s;
}
```

```cuda
__global__ void GEMV(const float* A, const float* x,
                     float* y, int R, int C) {
    __shared__ float tile[32]; // tiling with shared memory
    int r = blockIdx.x * blockDim.x + threadIdx.x;
    bool valid = (r < R);
    float s = 0.0f;
    for (int start = 0; start < C; start += 32) {
        for (int i = threadIdx.x; i < 32; i += blockDim.x) {
            int c = start + i;
            if (c < C) tile[i] = x[c]; // load x into tile
        }
        __syncthreads();
        if (valid) {
            for (int j = 0; j < min(32, C - start); j++) {
                s += A[r * C + (start + j)] * tile[j];
            }
        }
        __syncthreads();
    }
    if (valid) y[r] = s;
}
```

Figure 3: **An equivalent pair from the CUDA category in EquiBench.** Both programs perform matrix-vector multiplication ($y = Ax$). The right-hand program uses *shared memory tiling* to improve performance. Tensor compilers are utilized to explore different *scheduling strategies*, automating the generation.

identify differences between the two programs, align their states, and determine whether these states are consistently identical.

Example. Figure 2 illustrates an equivalent pair of C programs. In the left program, the condition (`p1 == p2`) compares the memory address of the first element of the array `b` with that of the static variable `c`. Since `b` and `c` reside in different memory locations, this condition can never be satisfied. As a result, the assignment `c = 1` is never executed in the left program and is removed in the right program.

Automation. This reasoning process is automated by compilers through *alias analysis*, which statically determines whether two pointers can reference the same memory location. Based on this analysis, the compiler's *Dead Code Elimination (DCE)* pass removes code that does not affect program semantics to improve performance.

Dataset Generation. We utilize CSmith (Yang et al., 2011) to create an initial pool of random C programs. Building on techniques from prior compiler testing research (Theodoridis et al., 2022), we implement an LLVM-based tool (Lattner and Adve, 2004) to classify code snippets as either dead or live. Live code is further confirmed by executing random inputs with observable side effects. Equivalent program pairs are generated by eliminating dead code, while inequivalent pairs are generated by removing live code.

3.2 Pairs from Compiler Scheduling (CUDA)

Definition of Equivalence. Two CUDA programs are considered equivalent if they produce the same mathematical output for any valid input, *disregarding floating-point rounding errors*. This definition *differs* from that in Section 3.1, as it does not require the internal program states to be identical during execution.

Example. Figure 3 shows an equivalent CUDA program pair. Both compute matrix-vector multiplication $y = Ax$, where A has dimensions (R, C) and x has size C. The right-hand program applies the *shared memory tiling* technique, loading `x` into shared memory `tile` (declared with `__shared__`). Synchronization primitives `__syncthreads()` are properly inserted to prevent synchronization issues.

Automation. The program transformation can be automated with tensor compilers, which provide a set of *schedules* to optimize loop-based programs. These schedules include loop tiling, loop fusion, loop reordering, loop unrolling, vectorization, and cache optimization. For any given schedule, the compiler can generate the transformed code. While different schedules can significantly impact program performance on the GPU, they do not affect the program's correctness (assuming no compiler

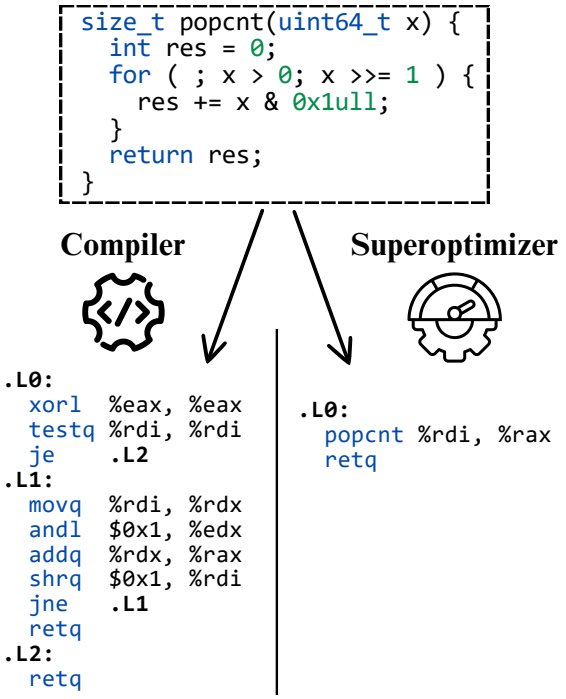

Figure 4: **An equivalent pair from the x86-64 category in EquiBench.** Both programs are compiled from the same C function shown above, the left using a compiler and the right using a *superoptimizer*. The function counts the number of set bits in the input `%rdi` register and stores the result in `%rax`. Their equivalence has been formally verified by the superoptimizer.

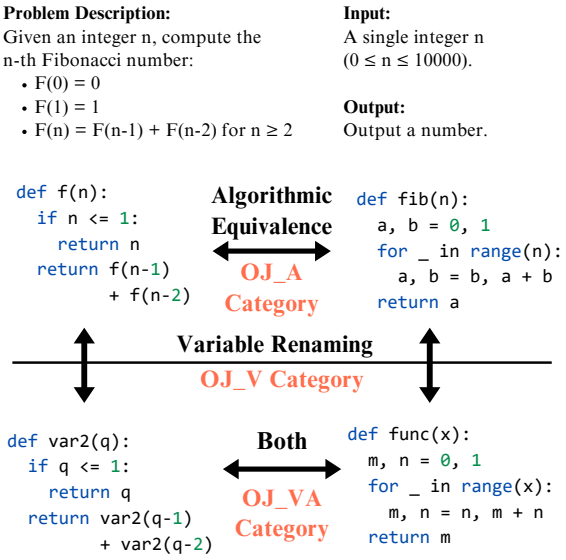

Figure 5: **Equivalent pairs from the OJ_A, OJ_V, OJ_VA categories in EquiBench.** OJ_A pairs demonstrate *algorithmic equivalence*, OJ_V pairs involve *variable renaming* transformations, and OJ_VA pairs combine *both* types of variations.

bugs), providing the foundation for automation.

Dataset Generation. We utilize TVM as the tensor compiler (Chen et al., 2018) and sample tensor program schedules from TenSet (Zheng et al., 2021) to generate equivalent CUDA program pairs. Inequivalent pairs are created by sampling code from different tensor programs.

3.3 Pairs from a Superoptimizer (x86-64)

Definition of Equivalence. Two x86-64 assembly programs are considered equivalent if, for any input provided in the specified input registers, both programs produce identical outputs in the specified output registers. Differences in other registers or memory are ignored for equivalence checking.

Example. Figure 4 shows an example of an equivalent program pair in x86-64 assembly. Both programs implement the same C function, which counts the number of bits set to 1 in the variable `x` (mapped to the `%rdi` register) and stores the result in `%rax`. The left-hand program, generated by GCC with O3 optimization, uses a loop to count each bit individually, while the right-hand program, produced by a superoptimizer, leverages the `popcnt` instruction, a hardware-supported operation for efficient bit counting. The superoptimizer verifies that both programs are semantically equivalent. Determining this equivalence requires a solid understanding of x86-64 assembly semantics and the ability to reason about all possible bit patterns.

Automation. A superoptimizer searches a space of programs to find one equivalent to the target. Test cases efficiently prune incorrect candidates, while formal verification guarantees the correctness of the optimized program. Superoptimizers apply aggressive and non-local transformations, making semantic equivalence reasoning more challenging. For example, in Figure 4, while a traditional compiler translates the loop in the source C program into a loop in assembly, a superoptimizer can find a more optimal instruction sequence by leveraging specialized hardware instructions. Such transformations are beyond the scope of traditional compilers.

Dataset Generation. We use Stoke (Schkufza et al., 2013) to generate program pairs. Assembly programs are sampled from prior work (Koenig et al., 2021), and Stoke applies transformations to produce candidate programs. If verification succeeds, the pair is labeled as equivalent; if the generated test cases fail, it is labeled as inequivalent.

Category	Language	# Pairs	Lines of Code		
			Min	Max	Avg.
DCE	C	400	98	880	541
CUDA	CUDA	400	46	1733	437
x86-64	x86-64	400	8	29	14
OJ_A	Python	400	3	3403	82
OJ_V	Python	400	2	4087	70
OJ_VA	Python	400	3	744	35

Table 1: **Statistics of the EquiBench dataset.**

3.4 Pairs from Programming Contests

Definition of Equivalence. Two programs are considered equivalent if they solve the same problem by producing the same output for any valid input, as defined by the problem description. Both programs, along with the problem description, are provided to determine equivalence.

Example. Given the problem description in Figure 5, all four programs are equivalent as they correctly compute the nth Fibonacci number. The **OJ_A** pairs demonstrate **algorithmic** equivalence—the left-hand program uses recursion, while the right-hand program employs a for-loop. The **OJ_V** pairs are generated through **variable renaming**, a purely syntactic transformation that obscures the program's semantics by removing meaningful variable names. The **OJ_VA** pairs combine **both** algorithmic differences and variable renaming.

Dataset Generation. We sample Python submissions using a publicly available dataset from Online Judge (OJ) (Puri et al., 2021). For OJ_A pairs, accepted submissions are treated as equivalent, while pairs consisting of an accepted submission and a wrong-answer submission are considered inequivalent. Variable renaming transformations are automated with an open-source tool (Flook, 2025).

4 Experimental Setup

Dataset. EquiBench consists of 2,400 program pairs across six equivalence categories, each with 200 equivalent and 200 inequivalent pairs. Table 1 summarizes the statistics of program lengths. Constructing the program pairs required substantial systems effort. For example, for the DCE category, we developed a 2,917-line LLVM-based tool, including 1,472 lines in C and C++, with alias analysis and path feasibility analysis to accurately classify live and dead code.

Prompts. The 0-shot evaluation is conducted using the prompt "You are here to judge if two programs are semantically equivalent. Here equivalence means {*definition*}. [Program 1]: {code1} [Program 2]: {code2} Please only output the answer of whether the two programs are equivalent or not. You should only output Yes or No." The definition of equivalence and the corresponding program pairs are provided for each category. Additionally, for the categories of OJ_A, OJ_V, and OJ_VA, the prompt also includes the problem description. The full prompts used in our experiments for each equivalence category are in Appendix A.3.

5 Results

5.1 Model Accuracy

Table 2 shows the accuracy results for 19 state-of-the-art large language models on EquiBench under zero-shot prompting. Our findings are as follows:

Reasoning models achieve the highest performance. As shown in Table 2, reasoning models such as OpenAI o3-mini, DeepSeek R1, and o1-mini significantly outperform all others in our evaluation. This further underscores the complexity of equivalence checking, where reasoning models exhibit a distinct advantage.

EquiBench is a challenging benchmark. Among the 19 models evaluated, OpenAI o4-mini achieves only 60.8% in the CUDA category despite being the top-performing model overall, with an accuracy of 82.3%. For the two most difficult categories, the highest accuracy across all models is 63.8% and 76.2%, respectively, only modestly above the random baseline of 50% accuracy for binary classification, highlighting the substantial room for improvement.

Scaling up models improves performance. Larger models generally achieve better performance. Figure 6 shows scaling trends for the Qwen2.5, Llama-3.1, and Mixtral families, where accuracy improves with model size. The x-axis is on a logarithmic scale, highlighting how models exhibit consistent gains as parameters increase.

5.2 Difficulty Analysis

We conduct a detailed difficulty analysis across equivalence categories and study how syntactic similarity influences model predictions.

Model	DCE	CUDA	x86-64	OJ_A	OJ_V	OJ_VA	Overall Accuracy
Random Baseline	*50.0*	*50.0*	*50.0*	*50.0*	*50.0*	*50.0*	*50.0*
Llama-3.2-3B-Instruct-Turbo	50.0	49.8	50.0	51.5	51.5	51.5	50.7
Llama-3.1-8B-Instruct-Turbo	41.8	49.8	50.5	57.5	75.5	56.8	55.3
Mistral-7B-Instruct-v0.3	51.0	57.2	73.8	50.7	50.5	50.2	55.6
Mixtral-8x7B-Instruct-v0.1	50.2	47.0	64.2	59.0	61.5	55.0	56.1
Mixtral-8x22B-Instruct-v0.1	46.8	49.0	62.7	63.5	76.0	62.7	60.1
Llama-3.1-70B-Instruct-Turbo	47.5	50.0	58.5	66.2	72.0	67.5	60.3
QwQ-32B-Preview	48.2	50.5	62.7	65.2	71.2	64.2	60.3
Qwen2.5-7B-Instruct-Turbo	50.5	49.2	58.0	62.0	80.8	63.0	60.6
gpt-4o-mini-2024-07-18	46.8	50.2	56.8	64.5	91.2	64.0	62.2
Qwen2.5-72B-Instruct-Turbo	42.8	56.0	64.8	72.0	76.5	70.8	63.8
Llama-3.1-405B-Instruct-Turbo	40.0	49.0	75.0	72.2	74.5	72.8	63.9
DeepSeek-V3	41.0	50.7	69.2	73.0	83.5	72.5	65.0
gpt-4o-2024-11-20	43.2	49.5	65.2	71.0	87.0	73.8	65.0
claude3.5-sonnet-2024-10-22	38.5	62.3	70.0	71.2	78.0	73.5	65.6
claude3.7-sonnet-2025-04-16	40.5	**63.8**	64.8	70.5	89.2	73.5	67.0
o1-mini-2024-09-12	55.8	50.7	74.2	80.0	89.8	78.8	71.5
DeepSeek-R1	52.2	61.0	78.2	79.8	91.5	78.0	73.5
o3-mini-2025-01-31	68.8	59.0	**84.5**	84.2	88.2	83.2	78.0
o4-mini-2025-04-16	**76.2**	60.8	83.0	**89.0**	**96.5**	**88.5**	**82.3**
Mean	49.0	53.4	66.7	68.6	78.1	68.5	64.0

Table 2: **Accuracy of 19 models on EquiBench under 0-shot prompting.** We report accuracy for each of the six equivalence categories along with the overall accuracy.

Difficulty by Transformation Type. Each category adopts a specific definition of equivalence (see Section 3), and the program transformations used in each category differ accordingly. We find that purely syntactic transformations are substantially easier for models, while structural and compiler-involved transformations are much harder. Specifically, **OJ_V** (variable renaming) achieves the highest mean accuracy of 78.1%, as it only requires surface-level reasoning. **OJ_A** (algorithmic equivalence) and **OJ_VA** (variable renaming combined with algorithmic differences) achieve similar accuracies of 68.6% and 68.5%, respectively. In contrast, **x86-64** (66.7%) and **CUDA** (53.4%) involve complex instruction-level or memory-level transformations, requiring deeper semantic reasoning. **DCE** (dead code elimination) is the most difficult category, with a mean accuracy of 49.0%, suggesting that models struggle with nuanced program analysis concepts.

Difficulty by Syntactic Similarity. To assess whether LLM predictions reflect understanding of program semantics rather than reliance on surface-level syntax, we analyze how syntactic similarity affects model behavior. Using Moss (Schleimer et al., 2003), a plagiarism detection tool, we observe the following:

- For program pairs with low syntactic similarity, models tend to predict "inequivalent," even when the programs are semantically equivalent. This suggests an overreliance on the superficial form of the code.
- For syntactically similar pairs, models are more likely to predict "equivalent," indicating a tendency to associate similarity in form with equivalence in program semantics.

We validate this trend through statistical testing: at significance level ($\alpha = 0.05$), model accuracy on equivalent pairs increases with syntactic similarity, while accuracy on inequivalent pairs decreases. This disconnect between syntactic form and execution behavior, as discussed in Section 1, suggests that models do not fully grasp program semantics.

Implications for Benchmark Design. These findings suggest that future benchmarks should emphasize *syntactically dissimilar yet equivalent program pairs* and *syntactically similar yet inequivalent program pairs* to create more challenging and diagnostic benchmarks for evaluating the deep semantic reasoning capabilities of LLMs.

5.3 Bias in Model Prediction

We evaluate the prediction bias of the models and observe a pronounced tendency to misclassify equivalent programs as inequivalent in the CUDA and x86-64 categories. Table 3 presents the results for four representative models, showing high accuracy for inequivalent pairs but significantly lower

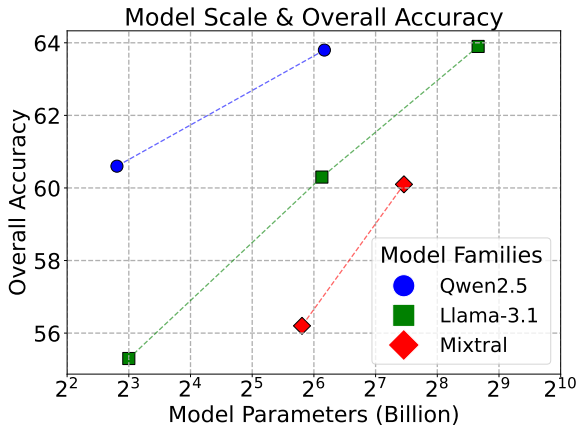

Figure 6: **Scaling Trend on EquiBench.** Models exhibit consistent gains as parameters increase.

accuracy for equivalent pairs, with full results for all models in Appendix A.2.

The bias in the CUDA category arises from extensive structural transformations, such as loop restructuring and shared memory optimizations, which make paired programs appear substantially different. In the x86-64 category, superoptimization applies non-local transformations to achieve optimal instruction sequences, introducing aggressive code restructuring that complicates equivalence reasoning and leads models to misclassify equivalent pairs as inequivalent frequently.

Model	CUDA		x86-64	
	Eq	Ineq	Eq	Ineq
Random Baseline	*50.0*	*50.0*	*50.0*	*50.0*
o3-mini	27.5	90.5	69.5	99.5
o1-mini	2.5	99.0	50.0	98.5
DeepSeek-R1	28.0	94.0	57.5	99.0
DeepSeek-V3	8.5	93.0	44.0	94.5

Table 3: Accuracies on equivalent and inequivalent pairs in the CUDA and x86-64 categories under 0-shot prompting, showing that **models perform significantly better on inequivalent pairs**. Random guessing serves as an unbiased baseline for comparison. More results are in Appendix A.2.

5.4 Prompting Strategies Analysis

We study few-shot in-context learning and Chain-of-Thought (CoT) prompting, evaluating four strategies: 0-shot, 4-shot, 0-shot with CoT, and 4-shot with CoT. For 4-shot, prompts include 2 equivalent and 2 inequivalent pairs. Table 4 shows the results.

Our key finding is that prompting strategies barely improve performance on EquiBench, highlighting the difficulty of understanding program semantics.

Model	0S	4S	0S-CoT	4S-CoT
o1-mini	71.5	71.5	**71.9**	**71.9**
gpt-4o	65.0	**66.5**	62.5	62.7
DeepSeek-V3	65.0	**66.9**	63.3	62.5
gpt-4o-mini	62.2	**63.5**	60.2	61.2

Table 4: **Accuracies of different prompting techniques.** We evaluate 0-shot and 4-shot in-context learning, both without and with Chain-of-Thought (CoT). Prompting strategies barely improve performance.

6 Discussion and Future Directions

Scope and Positioning Machine learning has been applied to many code-related tasks, such as clone detection (White et al., 2016), code search (Gao et al., 2024), and bug finding (Deng et al., 2023). EquiBench focuses on equivalence checking, which differs fundamentally by evaluating a model's understanding of program semantics. Unlike natural language, code is executable, and its correctness depends on execution results rather than form. For example, clone detection captures syntactic or structural similarity without considering behavior. In contrast, EquiBench tests whether two programs produce the same outputs for all inputs, offering an informative benchmark for reasoning about program behavior.

Developer Use Cases EquiBench evaluates whether LLMs truly understand program semantics, a capability that underpins downstream tasks such as program optimization, software refactoring, and transpilation. These tasks are central to practical scenarios where coding assistants must propose improvements or transformations without changing program behavior. For example, after a developer performs a refactoring, a coding assistant that performs well on EquiBench would be better positioned to judge whether the transformed code preserves the same functionality as the original.

Labeling Soundness To ensure high-assurance equivalence labels, EquiBench relies on transformations grounded in program analysis, compiler scheduling, and superoptimization, all of which offer strong soundness guarantees. In contrast, approaches such as random testing (Jiang and Su, 2009), similarity-based tools (Silva and Valente, 2017), and refactoring datasets lack formal guarantees and risk introducing incorrect labels.

Design and Extensibility EquiBench is designed with modularity in mind: each equivalence category corresponds to a distinct class of program transformations. Demonstrating strong performance in these settings would indicate that LLMs could support some components of compiler pipelines (e.g., dead code elimination (DCE) as a core compiler optimization, or CUDA program scheduling for high-performance ML systems). We focus on the six categories where large-scale, high-confidence labels can be generated automatically. That said, equivalence checking is a general task that is applicable to all programming languages. We view our benchmark as a first step, and its modular design allows future extension to additional categories and languages.

Evaluation of Reasoning Trace While our evaluation centers on binary classification, understanding the rationale behind model predictions is an important direction. Explanations may take the form of natural language or formal proofs, but verifying their correctness remains difficult. Natural language lacks reliable automated validation, since using LLMs as judges can produce unsound results. Building a proof-based evaluation framework using tools such as Lean is also highly nontrivial. We present a manual case analysis of reasoning trace correctness in Appendix A.1 and leave automated robust evaluation of reasoning as future work.

Effect of Fine-Tuning We tested whether supervised fine-tuning improves performance. Fine-tuning Qwen2.5-14B-Instruct with LoRA for 3 epochs on 1,200 labeled examples increased accuracy from 59.8% to 63.2%. The small gain suggests that binary labels alone provide limited learning signals for reasoning about program semantics. Prior work has explored training with *program execution traces* to better capture execution behavior. We conducted an additional experiment to evaluate the training approach from SemCoder (Ding et al., 2024). The base model (DeepSeek-Coder-6.7B) achieves 49.9% accuracy on our benchmark, and the fine-tuned model released by SemCoder reaches 54.9%. While this shows some benefit, the improvement remains modest. These results support our broader claim: EquiBench presents a difficult and meaningful challenge even for fine-tuned models, and deeper semantic understanding remains out of reach for current approaches.

Future Directions We believe EquiBench can inform future research on task-specific training methods, including: (1) distilling reasoning traces from stronger models, (2) scaling training with larger datasets generated through our pipeline, (3) developing agentic approaches where LLMs actively execute and compare programs using tools (e.g., a Python interpreter) to generate inputs that expose differences, (4) applying reinforcement learning with execution-based feedback, and (5) creating datasets with program analysis concepts (see Appendix A.1) for training LLMs.

7 Conclusion

EquiBench is a benchmark for evaluating whether large language models (LLMs) truly understand program semantics. We propose the task of equivalence checking, which asks whether two programs produce identical outputs for all possible inputs, as a direct way to test a model's ability to reason about program behavior. The dataset consists of 2400 program pairs across four languages and six categories, constructed through a fully automated pipeline that provides high-confidence labels and nontrivial difficulty. Our evaluation of 19 state-of-the-art LLMs shows that even the best-performing models achieve only modest accuracy in the most challenging categories. Further analysis shows that LLMs often rely on syntactic similarity instead of demonstrating robust reasoning about program semantics, underscoring the need for further advances in the semantic understanding of programs.

Limitations

While we make every effort to ensure that all program pairs in EquiBench are correctly labeled, we cannot guarantee absolute accuracy. The dataset is built through automated transformation pipelines that rely on external toolchains such as compilers, superoptimizers, and analysis frameworks. These components, although carefully chosen for their soundness guarantees, are not immune to subtle bugs or rare edge cases that may produce incorrect outputs. Furthermore, in the categories based on competitive programming submissions, some input programs may themselves be mislabeled due to incorrect online judge verdicts or subtle implementation flaws that escape the test cases. These sources of noise, though limited in scope, remain a potential source of labeling inaccuracy.

Acknowledgments

We thank Mingfei Guo, Lianmin Zheng, Allen Nie, Shiv Sundram, and Xiaohan Wang for their discussions. This work was partially supported by a Google Research Award and OpenAI's Researcher Access Program.

References

Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, and 1 others. 2021. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732*.

Sahar Badihi, Faridah Akinotcho, Yi Li, and Julia Rubin. 2020. Ardiff: scaling program equivalence checking via iterative abstraction and refinement of common code. In *Proceedings of the 28th ACM joint meeting on European software engineering conference and symposium on the foundations of software engineering*, pages 13–24.

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.

Sébastien Bubeck, Varun Chandrasekaran, Ronen Eldan, Johannes Gehrke, Eric Horvitz, Ece Kamar, Peter Lee, Yin Tat Lee, Yuanzhi Li, Scott Lundberg, and 1 others. 2023. Sparks of artificial general intelligence: Early experiments with gpt-4. *arXiv preprint arXiv:2303.12712*.

Liangyu Chen, Bo Li, Sheng Shen, Jingkang Yang, Chunyuan Li, Kurt Keutzer, Trevor Darrell, and Ziwei Liu. 2024. Large language models are visual reasoning coordinators. *Advances in Neural Information Processing Systems*, 36.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde De Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, and 1 others. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*.

Minyu Chen, Guoqiang Li, Ling-I Wu, and Ruibang Liu. 2025. Dce-llm: Dead code elimination with large language models. In *Proceedings of the 2025 Conference of the Nations of the Americas Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers)*, pages 9942–9955.

Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, and 1 others. 2018. Tvm: An automated end-to-end optimizing compiler for deep learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 578–594.

Berkeley Churchill, Oded Padon, Rahul Sharma, and Alex Aiken. 2019. Semantic program alignment for equivalence checking. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 1027–1040.

Peter Clark, Isaac Cowhey, Oren Etzioni, Tushar Khot, Ashish Sabharwal, Carissa Schoenick, and Oyvind Tafjord. 2018. Think you have solved question answering? try arc, the ai2 reasoning challenge. *arXiv preprint arXiv:1803.05457*.

Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, Mark Chen, Heewoo Jun, Lukasz Kaiser, Matthias Plappert, Jerry Tworek, Jacob Hilton, Reiichiro Nakano, and 1 others. 2021. Training verifiers to solve math word problems. *arXiv preprint arXiv:2110.14168*.

Chris Cummins, Volker Seeker, Dejan Grubisic, Mostafa Elhoushi, Youwei Liang, Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Kim Hazelwood, Gabriel Synnaeve, and 1 others. 2023. Large language models for compiler optimization. *arXiv preprint arXiv:2309.07062*.

Chris Cummins, Volker Seeker, Dejan Grubisic, Baptiste Roziere, Jonas Gehring, Gabriel Synnaeve, and Hugh Leather. 2024. Meta large language model compiler: Foundation models of compiler optimization. *arXiv preprint arXiv:2407.02524*.

Manjeet Dahiya and Sorav Bansal. 2017. Black-box equivalence checking across compiler optimizations. In *Asian Symposium on Programming Languages and Systems*, pages 127–147. Springer.

Yinlin Deng, Chunqiu Steven Xia, Haoran Peng, Chenyuan Yang, and Lingming Zhang. 2023. Large language models are zero-shot fuzzers: Fuzzing deep-learning libraries via large language models. In *Proceedings of the 32nd ACM SIGSOFT international symposium on software testing and analysis*, pages 423–435.

Haoran Ding, Zhaoguo Wang, Yicun Yang, Dexin Zhang, Zhenglin Xu, Haibo Chen, Ruzica Piskac, and Jinyang Li. 2023. Proving query equivalence using linear integer arithmetic. *Proceedings of the ACM on Management of Data*, 1(4):1–26.

Yangruibo Ding, Jinjun Peng, Marcus J Min, Gail Kaiser, Junfeng Yang, and Baishakhi Ray. 2024. Semcoder: Training code language models with comprehensive semantics reasoning. *arXiv preprint arXiv:2406.01006*.

Dennis Felsing, Sarah Grebing, Vladimir Klebanov, Philipp Rümmer, and Mattias Ulbrich. 2014. Automating regression verification. In *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*, pages 349–360.

Daniel Flook. 2025. Python variable renaming tool. https://github.com/dflook/python-minifier.

Zeyu Gao, Hao Wang, Yuanda Wang, and Chao Zhang. 2024. Virtual compiler is all you need for assembly code search. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 3040–3051.

Alex Gu, Baptiste Rozière, Hugh Leather, Armando Solar-Lezama, Gabriel Synnaeve, and Sida I Wang. 2024. Cruxeval: A benchmark for code reasoning, understanding and execution. *arXiv preprint arXiv:2401.03065*.

Shubhani Gupta, Aseem Saxena, Anmol Mahajan, and Sorav Bansal. 2018. Effective use of smt solvers for program equivalence checking through invariant-sketching and query-decomposition. In *International Conference on Theory and Applications of Satisfiability Testing*, pages 365–382. Springer.

Namgyu Ho, Laura Schmid, and Se-Young Yun. 2022. Large language models are reasoning teachers. *arXiv preprint arXiv:2212.10071*.

Jie Huang and Kevin Chen-Chuan Chang. 2022. Towards reasoning in large language models: A survey. *arXiv preprint arXiv:2212.10403*.

Ali Reza Ibrahimzada, Kaiyao Ke, Mrigank Pawagi, Muhammad Salman Abid, Rangeet Pan, Saurabh Sinha, and Reyhaneh Jabbarvand. 2024. Repository-level compositional code translation and validation. *arXiv preprint arXiv:2410.24117*.

Naman Jain, King Han, Alex Gu, Wen-Ding Li, Fanjia Yan, Tianjun Zhang, Sida Wang, Armando Solar-Lezama, Koushik Sen, and Ion Stoica. 2024. Livecodebench: Holistic and contamination free evaluation of large language models for code. *arXiv preprint arXiv:2403.07974*.

Lingxiao Jiang and Zhendong Su. 2009. Automatic mining of functionally equivalent code fragments via random testing. In *Proceedings of the eighteenth international symposium on Software testing and analysis*, pages 81–92.

Jason R Koenig, Oded Padon, and Alex Aiken. 2021. Adaptive restarts for stochastic synthesis. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, pages 696–709.

Emanuele La Malfa, Christoph Weinhuber, Orazio Torre, Fangru Lin, Samuele Marro, Anthony Cohn, Nigel Shadbolt, and Michael Wooldridge. 2024. Code simulation challenges for large language models. *arXiv preprint arXiv:2401.09074*.

Chris Lattner and Vikram Adve. 2004. Llvm: A compilation framework for lifelong program analysis & transformation. In *International symposium on code generation and optimization, 2004. CGO 2004.*, pages 75–86. IEEE.

Changshu Liu, Shizhuo Dylan Zhang, Ali Reza Ibrahimzada, and Reyhaneh Jabbarvand. 2024. Codemind: A framework to challenge large language models for code reasoning. *arXiv preprint arXiv:2402.09664*.

Chenxiao Liu, Shuai Lu, Weizhu Chen, Daxin Jiang, Alexey Svyatkovskiy, Shengyu Fu, Neel Sundaresan, and Nan Duan. 2023. Code execution with pre-trained language models. *arXiv preprint arXiv:2305.05383*.

Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin Clement, Dawn Drain, Daxin Jiang, Duyu Tang, Ge Li, Lidong Zhou, Linjun Shou, Long Zhou, Michele Tufano, MING GONG, Ming Zhou, Nan Duan, Neel Sundaresan, and 3 others. 2021. CodeXGLUE: A machine learning benchmark dataset for code understanding and generation. In *Thirty-fifth Conference on Neural Information Processing Systems Datasets and Benchmarks Track (Round 1)*.

Nickil Maveli, Antonio Vergari, and Shay B Cohen. 2024. What can large language models capture about code functional equivalence? *arXiv preprint arXiv:2408.11081*.

Iman Mirzadeh, Keivan Alizadeh, Hooman Shahrokhi, Oncel Tuzel, Samy Bengio, and Mehrdad Farajtabar. 2024. Gsm-symbolic: Understanding the limitations of mathematical reasoning in large language models. *arXiv preprint arXiv:2410.05229*.

Federico Mora, Yi Li, Julia Rubin, and Marsha Chechik. 2018. Client-specific equivalence checking. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, pages 441–451.

Ansong Ni, Miltiadis Allamanis, Arman Cohan, Yinlin Deng, Kensen Shi, Charles Sutton, and Pengcheng Yin. 2024. Next: Teaching large language models to reason about code execution. *arXiv preprint arXiv:2404.14662*.

Shankara Pailoor, Yuepeng Wang, and Işıl Dillig. 2024. Semantic code refactoring for abstract data types. *Proceedings of the ACM on Programming Languages*, 8(POPL):816–847.

Rangeet Pan, Ali Reza Ibrahimzada, Rahul Krishna, Divya Sankar, Lambert Pouguem Wassi, Michele Merler, Boris Sobolev, Raju Pavuluri, Saurabh Sinha, and Reyhaneh Jabbarvand. 2024. Lost in translation: A study of bugs introduced by large language models while translating code. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, pages 1–13.

Ruchir Puri, David S Kung, Geert Janssen, Wei Zhang, Giacomo Domeniconi, Vladimir Zolotov, Julian Dolby, Jie Chen, Mihir Choudhury, Lindsey Decker, and 1 others. 2021. Codenet: A large-scale ai for code dataset for learning a diversity of coding tasks. *arXiv preprint arXiv:2105.12655*.

Eric Schkufza, Rahul Sharma, and Alex Aiken. 2013. Stochastic superoptimization. *ACM SIGARCH Computer Architecture News*, 41(1):305–316.

Saul Schleimer, Daniel S Wilkerson, and Alex Aiken. 2003. Winnowing: local algorithms for document fingerprinting. In *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, pages 76–85.

Rahul Sharma, Eric Schkufza, Berkeley Churchill, and Alex Aiken. 2013. Data-driven equivalence checking. In *Proceedings of the 2013 ACM SIGPLAN international conference on Object oriented programming systems languages & applications*, pages 391–406.

Alexander Shypula, Aman Madaan, Yimeng Zeng, Uri Alon, Jacob Gardner, Milad Hashemi, Graham Neubig, Parthasarathy Ranganathan, Osbert Bastani, and Amir Yazdanbakhsh. 2023. Learning performance-improving code edits. *arXiv preprint arXiv:2302.07867*.

Danilo Silva and Marco Tulio Valente. 2017. Refdiff: Detecting refactorings in version histories. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*, pages 269–279. IEEE.

Rajat Singh and Srikanta Bedathur. 2024. Exploring the use of llms for sql equivalence checking. *arXiv preprint arXiv:2412.05561*.

Theodoros Theodoridis, Manuel Rigger, and Zhendong Su. 2022. Finding missed optimizations through the lens of dead code elimination. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 697–709.

Zhao Tian, Honglin Shu, Dong Wang, Xuejie Cao, Yasutaka Kamei, and Junjie Chen. 2024. Large language models for equivalent mutant detection: How far are we? In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 1733–1745.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, and 1 others. 2022. Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems*, 35:24824–24837.

Martin White, Michele Tufano, Christopher Vendome, and Denys Poshyvanyk. 2016. Deep learning code fragments for code clone detection. In *Proceedings of the 31st IEEE/ACM international conference on automated software engineering*, pages 87–98.

Chenyuan Yang, Yinlin Deng, Runyu Lu, Jiayi Yao, Jiawei Liu, Reyhaneh Jabbarvand, and Lingming Zhang. 2024a. Whitefox: White-box compiler fuzzing empowered by large language models. *Proceedings of the ACM on Programming Languages*, 8(OOPSLA2):709–735.

Chenyuan Yang, Zijie Zhao, and Lingming Zhang. 2023. Kernelgpt: Enhanced kernel fuzzing via large language models. *arXiv preprint arXiv:2401.00563*.

Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. 2011. Finding and understanding bugs in c compilers. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*, pages 283–294.

Zhen Yang, Fang Liu, Zhongxing Yu, Jacky Wai Keung, Jia Li, Shuo Liu, Yifan Hong, Xiaoxue Ma, Zhi Jin, and Ge Li. 2024b. Exploring and unleashing the power of large language models in automated code translation. *Proceedings of the ACM on Software Engineering*, 1(FSE):1585–1608.

Dylan Zhang, Curt Tigges, Zory Zhang, Stella Biderman, Maxim Raginsky, and Talia Ringer. 2024. Transformer-based models are not yet perfect at learning to emulate structural recursion. *arXiv preprint arXiv:2401.12947*.

Fuheng Zhao, Lawrence Lim, Ishtiyaque Ahmad, Divyakant Agrawal, and Amr El Abbadi. 2023. Llm-sql-solver: Can llms determine sql equivalence? *arXiv preprint arXiv:2312.10321*.

Lianmin Zheng, Ruochen Liu, Junru Shao, Tianqi Chen, Joseph E Gonzalez, Ion Stoica, and Ameer Haj Ali. 2021. Tenset: A large-scale program performance dataset for learned tensor compilers. In *Thirty-fifth Conference on Neural Information Processing Systems Datasets and Benchmarks Track (Round 1)*.

Denny Zhou, Nathanael Schärli, Le Hou, Jason Wei, Nathan Scales, Xuezhi Wang, Dale Schuurmans, Claire Cui, Olivier Bousquet, Quoc Le, and 1 others. 2022. Least-to-most prompting enables complex reasoning in large language models. *arXiv preprint arXiv:2205.10625*.

A Appendix

A.1 Case Studies

Models lack capabilities for sound equivalence checking. We find that simple changes that lead to semantic differences can confuse the models, causing them to produce incorrect predictions despite their correct predictions on the original program pairs. For example, o3-mini, which is one of the top-performing models in CUDA category, can correctly classify the pair shown in Figure 3 as equivalent. Next, we introduce synchronization bugs into the right-hand program, creating two inequivalent pairs with the original left-hand program: (1) removing the first `__syncthreads();` allows reads before all writes complete, causing race conditions; (2) removing the second `__syncthreads();` lets faster threads overwrite shared data while slower threads read it. Despite these semantic differences, o3-mini misclassifies both pairs as equivalent.

Proper hints enable models to correct misjudgments. After o3-mini misclassifies the modified pairs, a hint about removed synchronization primitives allows it to correctly identify both as inequivalent, with accurate explanations highlighting data races. This suggests that training models on dedicated program analysis datasets, beyond only raw source code, may be useful for improving their code reasoning capabilities.

A.2 Model Prediction Bias

We evaluate the prediction bias of the models and observe a pronounced tendency to misclassify equivalent programs as inequivalent in the CUDA and x86-64 categories. Figure A1 here shows the full results on models under 0-shot prompting.

A.3 Prompts

A.3.1 DCE Category

We show the prompts for the 0-shot setting.

You are here to judge if two C programs are semantically equivalent.
Here equivalence means that, when run on the same input, the two programs always have the same program state at all corresponding points reachable by program execution.
[Program 1]:

```
{program_1_code}
```

[Program 2]:

```
{program_2_code}
```

Please only output the answer of whether the two programs are equivalent or not. You should only output YES or NO.

A.3.2 CUDA Category

We show the prompts for the 0-shot setting.

You are here to judge if two CUDA programs are semantically equivalent.
Here equivalence means that, when run on the same valid input, the two programs always compute the same mathematical output (neglecting floating point rounding errors).
[Program 1]:

```
{program_1_code}
```

[Program 2]:

```
{program_2_code}
```

Please only output the answer of whether the two programs are equivalent or not. You should only output YES or NO.

A.3.3 x86-64 Category

We show the prompts for the 0-shot setting.

You are here to judge if two x86-64 programs are semantically equivalent.
Here equivalence means that, given any input bits in the register `{def_in}`, the two programs always have the same bits in register `{live_out}`. Differences in other registers do not matter for equivalence checking.

[Program 1]:

```
{program_1_code}
```

[Program 2]:

```
{program_2_code}
```

Please only output the answer of whether the two programs are equivalent or not. You should only output YES or NO.

Model	CUDA		x86-64	
	Eq	**Ineq**	**Eq**	**Ineq**
Random Baseline	*50.0*	*50.0*	*50.0*	*50.0*
deepseek-ai/DeepSeek-V3	8.5	93.0	44.0	94.5
deepseek-ai/DeepSeek-R1	28.0	94.0	57.5	99.0
meta-llama/Llama-3.1-405B-Instruct-Turbo	6.0	92.0	68.5	81.5
meta-llama/Llama-3.1-8B-Instruct-Turbo	2.0	97.5	1.0	100.0
meta-llama/Llama-3.1-70B-Instruct-Turbo	7.0	93.0	27.5	89.5
meta-llama/Llama-3.2-3B-Instruct-Turbo	0.0	99.5	0.0	100.0
anthropic/claude-3-5-sonnet-20241022	62.5	62.0	49.5	90.5
Qwen/Qwen2.5-7B-Instruct-Turbo	18.5	80.0	17.5	98.5
Qwen/Qwen2.5-72B-Instruct-Turbo	14.5	97.5	36.0	93.5
Qwen/QwQ-32B-Preview	35.0	66.0	39.0	86.5
mistralai/Mixtral-8x7B-Instruct-v0.1	18.0	76.0	50.5	78.0
mistralai/Mixtral-8x22B-Instruct-v0.1	10.5	87.5	32.5	93.0
mistralai/Mistral-7B-Instruct-v0.3	52.5	62.0	87.0	60.5
openai/gpt-4o-mini-2024-07-18	0.5	100.0	16.5	97.0
openai/gpt-4o-2024-11-20	0.0	99.0	68.5	62.0
openai/o3-mini-2025-01-31	27.5	90.5	69.5	99.5
openai/o1-mini-2024-09-12	2.5	99.0	50.0	98.5

Figure A1: Model prediction bias.

A.3.4 OJ_A, OJ_V, OJ_VA Category

We show the prompts for the 0-shot setting.

You are here to judge if two Python programs are semantically equivalent.
You will be given [Problem Description], [Program 1] and [Program 2].
Here equivalence means that, given any valid input under the problem description, the two programs will always give the same output.

[Problem Description]:

```
{problem_html}
```

[Program 1]:

```
{program_1_code}
```

[Program 2]:

```
{program_2_code}
```

Please only output the answer of whether the two programs are equivalent or not. You should only output YES or NO.