Assessing, Exploiting, and Mitigating Syntactic Robustness Failures in LLM-Based Code Generation

Laboni Sarker
Computer Science
University of California Santa Barbara
Santa Barbara, California, USA
labonisarker@cs.ucsb.edu

Mara Downing
Computer Science
University of California Santa Barbara
Santa Barbara, California, USA
maradowning@ucsb.edu

Achintya Desai
Computer Science
University of California, Santa Barbara
Santa Barbara, California, USA
achintya@ucsb.edu

Tevfik Bultan
Computer Science
University of California, Santa Barbara
Santa Barbara, California, USA
bultan@ucsb.edu

Abstract

Rapid advances in the field of Large Language Models (LLMs) have made LLM-based code generation an important area for investigation. An LLM-based code generator takes a prompt as input and produces code that implements the requirements specified in the prompt. Many software requirements include mathematical formulas that specify the expected behavior of the code to be generated. Given a code generation prompt that contains a mathematical formula, a reasonable expectation is that, if the formula is syntactically modified without changing its semantics, the generated code for the modified prompt should be semantically equivalent. We formalize this concept as syntactic robustness and investigate the syntactic robustness of LLMs as code generators. Our experimental assessment demonstrates that LLMs are not syntactically robust for code generation prompts with formulas, especially for the ones that require mathematical reasoning. We investigate attack strategies that can further deteriorate the syntactic robustness of LLMs. Finally, to mitigate syntactic robustness failures in LLMs, we propose a pre-processing step that uses reductions to transform formulas in prompts to a simplified form. Our experimental results demonstrate that the syntactic robustness of LLM-based code generation improves significantly using our approach, improving syntactic robustness of LLMs from 54.05% to 74.42%.

CCS Concepts

• **Software and its engineering → Software verification and validation**.

Keywords

syntactic robustness, LLMs, code generation, exploitation, preprocessing, formula reduction.

FORGE '26, Rio de Janeiro, Brazil

ACM ISBN 979-8-4007-2477-0/26/04
https://doi.org/10.1145/3793655.3793727

ACM Reference Format:
Laboni Sarker, Mara Downing, Achintya Desai, and Tevfik Bultan. 2026. Assessing, Exploiting, and Mitigating Syntactic Robustness Failures in LLM-Based Code Generation. In *2026 IEEE/ACM Third International Conference on AI Foundation Models and Software Engineering (FORGE '26), April 12–13, 2026, Rio de Janeiro, Brazil.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3793655.3793727

1 Introduction

Large language models (LLMs) are becoming popular in the software engineering community [33, 45], especially for code generation tasks [51, 53, 74, 97], and have shown strong performance on code generation benchmarks [6, 21, 46]. Recently, LLMs have also been used for code generation in the scientific computing domain [10, 31, 55, 58, 59, 78, 88] and for generating code that efficiently handles complex mathematical computations [24, 58, 59, 78, 95]. Furthermore, LLMs have shown promise in handling formal specifications [69, 91], which rely on mathematical formulas in the specification of software engineering tasks. As its adoption grows, there is a critical need for systematic evaluation of the correctness and reliability of LLM-based code generation [18, 33, 50, 74, 97, 104, 106, 114].

Reliability in machine learning systems is often assessed through robustness, which has been extensively studied for classification tasks [15, 17, 56, 89, 92, 98, 103]. Robustness in classification models involves defining a neighborhood around an input with a known classification and ensuring the same classification is obtained within that neighborhood [35]. For generative models, however, this definition must be adapted, as obtaining and verifying objectively accurate outputs is more complex—many different outputs may be objectively correct. This highlights the need for well-defined robustness metrics to analyze LLM-based code generators, which are a subdomain of generative models.

Several prior works [18, 33, 50, 74, 97, 104, 106, 114] highlight the importance of studying robustness for code generation by LLMs,

This material is based on research supported by NSF under grants 2124039 and 2008660 and DARPA under the agreement number N66001-22-2-4037. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the U.S. Government.

and some do investigate robustness [79, 97, 106]. However, the impact of syntactic transformations of the prompts together with the increasing syntactic distance (Section 4) has not been investigated. Additionally, black-box attacks specifically designed to expose failures in the robustness of LLM-based code generators in handling syntactic variations of semantically equivalent prompts (Section 5) have not been explored.

In this paper, we define a formal robustness measure for LLM-based code generators called syntactic robustness (Section 3) and analyze the performance of LLMs with respect to this measure (Section 8). Intuitively, we define syntactic robustness as the degree to which the semantically equivalent prompts elicit semantically equivalent responses from the LLM code generator.

We define a set of syntactic transformations (Section 4) that generate syntactically different but semantically equivalent variations of mathematical formulas to assess the syntactic robustness of LLM-based code generators. We present attack strategies (Section 5) to exploit the syntactic robustness failures of the existing LLMs while minimizing the modifications to the formulas in the prompts. Furthermore, we present a pre-processing step (Section 6) to mitigate syntactic robustness failures and defend against such attacks, and demonstrate its effectiveness. Our assessment reveals that state-of-the-art LLMs lack syntactic robustness and that prompts requiring mathematical reasoning exhibit lower robustness compared to those involving direct translation of mathematical formulas into code. Our experimental evaluation also shows that our attacks can effectively exploit the syntactic robustness failures of the LLMs. Finally, our experiments demonstrate that our pre-processing effectively mitigates syntactic robustness failures, significantly improving syntactic robustness of LLMs.

Contributions. Our contributions in this paper are as follows:

(1) A formal definition of syntactic robustness (Section 3).
(2) A set of prompts with mathematical formulas, designed to benchmark the robustness of LLM-based code generators (provided as artifact [7]).
(3) A set of rules to mutate the mathematical formulas, that along with the original prompts, can be used for syntactic robustness evaluation (Section 4).
(4) A set of black-box attacks for exploiting syntactic robustness failures of LLMs (Section 5).
(5) A prompt pre-processing technique that helps in mitigating syntactic robustness failures of LLMs (Section 6).
(6) Empirical assessment of syntactic robustness, along with novel attack and mitigation techniques, on state-of-the-art LLM foundational models and frameworks (Section 8).

Organization. We discuss motivating examples of code generation prompts in Section 2, then we provide a formal definition of syntactic robustness and related concepts in Section 3. We discuss our formula transformation procedure in Section 4 and detail our attacks in Section 5. We explain our pre-processing procedure in Section 6, and then explicate implementation design details in Section 7. We present our experimental evaluation in Section 8, discuss the related work in Section 9, and conclude the paper in Section 10.

2 Motivation

Mathematical formulas and constraints are common in code generation prompts in a variety of domains [10, 24, 55, 91, 95]. Given many equivalent ways one can write a mathematical formula or constraint, ensuring syntactic robustness is crucial for code generation in such contexts. Thus, it is critical to evaluate the robustness of LLM-based code generators for prompts with mathematical formulas. Note that, the equivalence of mathematical formulas can be defined clearly and unambiguously, allowing for an automated evaluation of robustness [44, 72, 93]. This is in contrast to prior works that mutate the text of the prompt [74, 97], which can alter the meaning due to the ambiguity of natural language.

Prompt A *"Implement a C program which takes 'a','b' as inputs*
where method func(a: array<int>, b: array<int>)
returns (result: bool)
requires a != null && b != null
ensures result ==> exists i, j :: 0 <= i < a.Length &&
0 <= j < b.Length && ***Eq***
ensures !result ==> forall i, j :: 0 <= i < a.Length &&
0 <= j < b.Length ==> a[i] != b[j]."

Expressions for placeholder *Eq* in Prompt A
Eq: a[i] == b[j]
Eq: (a[i]) * 7 - (b[j]) * 7 == (b[j]) * 7 - (b[j]) * 7

Syntactically transformed prompts with mutated formulas can also be interpreted as adversarial samples for assessing the robustness of LLM-based code generators under adversarial attacks [27, 110, 118]. Our evaluation demonstrates that LLMs are vulnerable to adversarial prompt-based attacks, motivating the need for assessing and improving the syntactic robustness of LLM-based code generators.

Gpt-4o Response Snippet for *Eq*: *a[i] == b[j]*

```
...
     for (int j = 0; j < length; j++) {
         if (a[i] == b[j]) {
                 return true; }}}
    return false; }
```

Gpt-4o Response Snippet for *Eq*: *(a[i]) * 7 - (b[j]) * 7 == (b[j]) * 7 - (b[j]) * 7*

```
...
    if ((a[i]*7 - b[j]*7) == (b[j]*7 - b[j]*7)){
             return true; }}}
... if (a[i] == b[j]) { return false; } ...
   return true; }
```

Figure 1: Code segments from the programs generated by Gpt-4o for Prompt A

Example prompts. We motivate our work through a series of example code generation prompts involving mathematical formulas from different datasets. Prompt A is from the clover dataset [91], which contains formal specifications, Prompt B is generated from a HackerRank problem [4], and Prompt C and D are from the MaTT dataset [25].

Prompt A and B demonstrate the syntactic robustness challenges in LLM-based code generation. Prompt A shows a code generation query to generate a program that returns *True* if two input arrays

share any common element and *False* otherwise. On the other hand, Prompt B asks to generate a program that can find all b-digit numbers that are also the b-th power of a natural number. The placeholder ***Eq*** replacements for both Prompt A and B are shown at the bottom of each prompt and the corresponding outputs are presented in Figures 1 and 2, respectively.

Prompt B *"The number of digits present in a number is represented by 'd'. 'a' represents a natural number. Write a C code which will return all possible 'x' given a value 'b' which will satisfy the following pre and post-condition. Pre-condition: 1 <= b && b <= 19 Post condition: d=b and **Eq**. Print only the value of all possible values of x in comma separated form."*

Expressions for placeholder *Eq* in Prompt B
Eq: x = a^b
Eq: x+b = a^b+b

Figures 1 and 2 show code segments generated by GPT-4o highlighting the differences for semantically equivalent prompts. Despite the semantic equivalence of the mathematical formulas in the prompts, the generated codes reflect different interpretations. Both examples demonstrate the case where Gpt-4o struggles to handle syntactic mutations on a simple equation. Note that, in both prompts, the natural language texts are kept the same—the only distinction is the semantically equivalent but syntactically different equations. However, the generated codes from Gpt-4o are not semantically equivalent. In our experiments (Section 8), we show that this problem exists across multiple LLMs for a variety of prompts.

Gpt-4o Response Snippet for *Eq*: x=a^b

```
while (1) {
      long long x = pow(a, b);
```

Gpt-4o Response Snippet for *Eq*: x+b=a^b+b

```
for (int a = 1; ; a++) {
       long long aPowerB = pow(a, b);
       long long x = aPowerB + b;
```

Figure 2: Code segments from the programs generated by Gpt-4o for Prompt B

Prompt C *"Newton's Law of Gravitation states that two bodies with masses a and b attract each other with a force F=9.8 * a * b / d^2 where d is the distance between the bodies and 9.8 is the gravitational constant. If one of the bodies is fixed, implement a C program to find the work needed to move the other from d=c to d=e where 'a', 'b','c','e' are inputs to the program. Assume none of 'a', 'b', 'c', 'd' or 'e' are 0. Print only the work needed up to 2 digit precision after decimal (do not print anything else)."*

In code generation tasks, semantically equivalent formulas can often appear in syntactically different forms, particularly when derived from complex specifications. As the examples above demonstrate, enhancing the reliability of LLMs requires investigating and improving their syntactic robustness before deploying them.

Reasoning and Translation Prompts. We define two types of prompts for our analysis: reasoning and translation prompts. Consider the Prompt C below which asks for a program to find out the work done when a body is moved from one fixed position to another. The expected output from this prompt is to get the amount of work done which is $9.8 \times a \times b \times (1/c - 1/e)$ obtained by computing the integral $\int_c^e (9.8 \times a \times b/d^2)$ with respect to d.

We create a variation of Prompt C (we call Prompt D) by replacing the second sentence of the prompt as *"Implement a C program to find the attraction force where 'a', 'b' and 'd' are inputs to the program."* Prompt D contains the same equation as Prompt C but asks to calculate the attraction force given the equation, where the expected output is computed by evaluating the expression $9.8 \times a \times b/d^2$, which is provided in the prompt.

Although both prompts involve the same mathematical formula, Prompt C requires mathematical reasoning to perform integration, while Prompt D requires translation of the input formula to an expression in the target programming language to be evaluated during program execution. We name these two classes of prompts as *reasoning* and *translation* prompts, respectively. Our experiments indicate that syntactic robustness of LLM-based code generators differ for reasoning and translation prompts.

3 Syntactic Robustness

In this section, we formalize the concept of syntactic robustness. We start by defining key concepts including LLM-based code generators, prompts, and generated programs and their equivalence, before introducing our formal definition of syntactic robustness.

LLM-based Code Generators. We define an *LLM-based code generator* as follows:

Definition 1. *An* **LLM-based code generator** G *takes a prompt* $P \in \mathcal{P}$ *as input and generates code* $C \in \mathcal{C}$*, denoted as* $G : \mathcal{P} \rightarrow \mathcal{C}$*, where* $\mathcal{P}$ *is the set of prompts and* $\mathcal{C}$ *is the set of programs.*

In this paper, we focus on prompts that contain both English text and mathematical formulas:

Definition 2. *A* **prompt** $P \in \mathcal{P}$ *consists of an English text* T *and a mathematical formula* F *denoted as a tuple* $P\langle T, F\rangle$.

When an LLM-based code generator generates code C for prompt $P\langle T, F\rangle$, we denote it as: $G(P\langle T, F\rangle) = C$.

Semantic Equivalence of Formulas. Two syntactically different formulas F_1 and F_2 can be semantically equivalent. We denote the semantic equivalence of two formulas F_1 and F_2 as:

$$[[F_1]] \equiv [[F_2]]$$

which means that given the same valuation, formulas F_1 and F_2 evaluate to the same value. For example, consider the two formulas F_1 and F_2:

$$F_1 : a \times x + b = 0 \qquad F_2 : a \times x + a + b = a$$

Note that although F_1 and F_2 are syntactically different formulas they are semantically equivalent (i.e., $F_1 \neq F_2$ and $[[F_1]] \equiv [[F_2]]$).

Programs as Functions. We focus on programs that can be modeled as functions. We assume that, for a given input, each program execution terminates and produces the same output. Formally:

Definition 3. *A* **program** *C is a total function from the domain of inputs to the domain of outputs, $C : I \rightarrow O$, where $C(i) = o$ denotes that on input $i \in I$, the output of C is $o \in O$.*

Program Equivalence. We define equivalence of programs based on their input-output behavior:

Definition 4. *Given two programs $C_1 : I_1 \rightarrow O_1$ and $C_2 : I_2 \rightarrow O_2$ where $I_1 = I_2$,*

- *C_1 and C_2 are* **equivalent**, *denoted as $[[C_1]] \equiv [[C_2]]$, if and only if, $\forall i \in I, C_1(i) = C_2(i)$.*
- *C_1 and C_2 are* **non-equivalent**, *denoted as $[[C_1]] \not\equiv [[C_2]]$, if and only if, $\exists i \in I, C_1(i) \neq C_2(i)$.*

Note that different implementations of the same functionality are considered equivalent according to this definition as long as the input-output behavior is the same.

Syntactic Robustness. We can now begin to define syntactic robustness for LLM-based code generators:

Definition 5. *An LLM-based code generator G is* **syntactically robust**, *if and only if, given any two prompts $P\langle T, F_1\rangle, P\langle T, F_2\rangle \in \mathcal{P}$ where $[[F_1]] \equiv [[F_2]]$, $[[G(P\langle T, F_1\rangle)]] \equiv [[G(P\langle T, F_2\rangle)]]$.*

i.e., an LLM-based code generator is syntactically robust if it generates equivalent code for semantically equivalent but syntactically different prompts.

There are two issues with the above definition that we will address. First, according to the definition, if an LLM-based code generator generates the same code for all prompts, it would be syntactically robust. i.e., an LLM-based code generator that for all prompts generates the same trivial code:

```
int main()  printf("0\n"); return 0;
```

would be syntactically robust. To address this problem, we introduce the concept of a reference code for each prompt as follows:

Definition 6. *Given a prompt $P\langle T, F\rangle$ we call $R(P\langle T, F\rangle) = C_F^R$ the* **reference code** *for the prompt $P\langle T, F\rangle$, where C_F^R is a correct implementation of the requirements specified in the prompt $P\langle T, F\rangle$.*

Note that C_F^R can be written manually or can be generated by a code generator and validated by different means (such as manual inspection, testing, or verification).

Second, Definition 5 of syntactic robustness requires the LLM-based code generator to generate semantically equivalent programs for all semantically equivalent prompts. Even if the code generator generates semantically different code for only one syntactically different prompt while generating semantically equivalent code for all other prompts, it is not syntactically robust according to the above definition. So, we extend our definition below to measure the *syntactic robustness degree*, where Definition 5 corresponds to the highest degree of syntactic robustness. Syntactic robustness degree for an LLM-based code generator for a given prompt, its reference code, and its syntactic variations as follows:

Definition 7. *Given an LLM-based code generator G, a prompt $P\langle T, F\rangle$, a reference code $R(P\langle T, F\rangle) = C_F^R$ for prompt P, and a set of formulas $\mathcal{F}_F$ containing syntactic variations of F where for each $F' \in \mathcal{F}_F$, $[[F']] \equiv [[F]]$, let $\mathcal{F}_F^{eq} \subseteq \mathcal{F}_F$ denote the set of formulas such that for each $F' \in \mathcal{F}_F^{eq}$, $[[G(P\langle T, F'\rangle)]] \equiv [[C_F^R]]$. Then, the* **syntactic robustness degree** *of G with respect to P and $\mathcal{F}_{\mathcal{F}}$ is defined as:*

$$|\mathcal{F}_F^{eq}|/|\mathcal{F}_F|$$

$$
\begin{aligned}
&F \rightarrow E = E \mid E\\
&E \rightarrow N \mid S \mid V \mid U \mid E + E \mid E - E \mid E \times E \mid E/E \mid E\hat{}E \mid (E)\\
&U \rightarrow sin(E) \mid cos(E) \mid tan(E) \mid log(E) \mid ln(E) \mid V(E) \mid V'(V)\\
&N \rightarrow -N \mid [0-9]^+ \mid [0-9]^+.\,[0-9]^+\\
&S \rightarrow a \mid b \mid c \mid d \mid e\\
&V \rightarrow x \mid y \mid z
\end{aligned}
$$

Figure 3: Context-free grammar for mathematical formulas.

where $|\mathcal{F}_F^{eq}|$ denotes the number of formulas in $\mathcal{F}_F^{eq}$ and $|\mathcal{F}_F|$ denotes the number of formulas in $\mathcal{F}_F$.

We report the syntactic robustness degree as a percentage where 100% corresponds to the case where $\mathcal{F}_F^{eq} = \mathcal{F}_F$. Note that the definition of syntactic robustness given in Definition 5 corresponds to the syntactic robustness degree of 100% for all prompts and their corresponding semantically equivalent syntactic variations.

4 Syntactic Mutations of Formulas

In this paper, we consider prompts that include mathematical equations and expressions which can be univariate or multivariate.

Grammar for formulas. The context-free grammar shown in Figure 3 captures the mathematical formulas we use, where F denotes the start symbol which can be extended to either an equation ($E = E$) or an expression (E), U denotes unary functions, N denotes number literals, S denotes coefficients, and V represents the variables or functions. Note that, $V_1'(V_2)$ denotes the first derivative of function V_1 with respect to V_2.

Now, we discuss how we generate the set of formulas $\mathcal{F}_F$ that are syntactic variations of a given formula F.

Definition 8. *A* **syntactic transformation** *ST is a function that maps a formula to another formula that is semantically equivalent, i.e., $ST : \mathcal{F} \rightarrow \mathcal{F}$ such that for any formula $F \in \mathcal{F}$, $[[F]] \equiv [[ST(F)]]$, where $\mathcal{F}$ denotes the set of formulas.*

We define 18 mutation types for equations and 12 mutation types for expressions that modify syntax while preserving the semantics of the mathematical formula. These mutations include commutative addition and multiplication, adding zero, multiplication by one, and variable renaming. Table 1 presents a representative subset of the mutation rules for both expressions and equations, while the complete set is provided in the artifact [7].

Given a formula F, we generate the set of syntactic mutations of F, called $\mathcal{F}_F$ (as described in Definition 7) by repeatedly applying mutations ($M(F), M(M(F))$, $M(M(M(F)))$,...) to F, i.e., each $F' \in \mathcal{F}_F$ and by definitions of the mutations we introduced, $[[F']] \equiv [[F]]$.

Definition 9. *Given a formula F and its syntactic mutant F', the* **syntactic distance** *of F and F' is n when $F' = M^n(F)$.*

I.e., the syntactic distance between F and its mutant F' is the number of mutations needed to mutate F to F'.

5 Attacks Targeting Syntactic Robustness

We consider a black-box adversary aiming to reduce the syntactic robustness of an LLM in code generation tasks. The adversary can only apply semantics-preserving syntactic transformations to the

Table 1: A representative subset of mutation rules for equations and expressions (where $I \to N \mid S$ and $0 \notin N$).

Equation		Expression	
M_1	$\frac{E_1=E_2}{E_1/I=E_2/I}$	$M_{1,2}$	$\frac{E}{E\times I/I}$
M_2	$\frac{E_1=E_2}{E_1\times I=E_2\times I}$		
M_3	$\frac{E_1=E_2}{E_1+I=E_2+I}$	$M_{3,4}$	$\frac{E}{E+I-I}$
M_4	$\frac{E_1=E_2}{E_1-I=E_2-I}$		

Algorithm 1 PROFILING($\mathcal{M}, \mathcal{P}$)

▷ Decides effectiveness of each mutation for attack.
▷ Calls function THRESHOLD to choose a threshold to get top K mutations.

Input: $\mathcal{M}$: mutation set and $\mathcal{P}$: set of prompts to mutate.
Output: $\mathcal{W}$: set of weights describing effect of each mutation type on syntactic robustness, $\mathcal{M}_{Top-K}$: A set of Top-K mutations.

for $M \in \mathcal{M}$ **do**
  $robust\_score \leftarrow \Sigma_{P\in\mathcal{P}}(|\mathcal{F}_F^{eq}|/|\mathcal{F}_F| - |\mathcal{F}_{M(F)}^{eq}|/|\mathcal{F}_{M(F)}|)$
  $\mathcal{W}[M] \leftarrow robust\_score/|\mathcal{M}|$
$t_k \leftarrow$ THRESHOLD($\mathcal{W}, k$)
for $M \in \mathcal{M}$ **do**
  if $\mathcal{W}[M] > t_k$ **then**
    $\mathcal{M}_{Top-K} \leftarrow \mathcal{M}_{Top-K} \cup \{M\}$
return $\mathcal{W}, \mathcal{M}_{Top-K}$

input prompts and has no access to the model architecture or parameters. The goal is to induce incorrect or non-compiling code without changing the intended semantics. We assume the model is pretrained and do not consider attacks on training data or deployment-stage injection. This threat model enables systematic analysis of LLM failure modes via adversarial robustness testing [82, 117] and is applicable to automated code generation pipelines, where inputs are fed directly to LLM-based code generators [111] with minimal human oversight.

We use Algorithm 1 to profile the impact of different mutations on syntactic robustness degree of LLMs. This algorithm assigns a score ($\mathcal{W}[M]$) for each mutation (M) based on the decrease in syntactic robustness from the original prompt ($P\langle T, F\rangle$) to a modified prompt ($P\langle T, M(F)\rangle$). Based on this profiling information, we also compute a threshold (t_k) which we use to choose a subset of the mutations that have the most impact on robustness ($\mathcal{M}_{Top-K}$).

Algorithm 2 outlines our three attack strategies. Each strategy takes as input a formula F from a prompt and a syntactic distance D, and produces a mutated formula $F' = M^D(F)$. For these strategies, we take care to ensure that a formula at distance D cannot be created by a series of mutations less than D. We construct the mutation space using a dynamic programming approach: We iteratively generate all unique formulas at distance 1, then distance 2 (excluding those already generated at smaller distances), and so on up to distance D. We additionally introduce another attack strategy **NaiveUniform** which does not include this distance assurance—a mutation of distance D with the **NaiveUniform** attack could potentially be created with a lower distance.

NaiveUniform chooses randomly from mutation set, but can choose mutations that lead to a lower distance mutated formula.

Algorithm 2 ATTACK(F, D, $\mathcal{M}$)

▷ Attacks by mutating a formula F to distance D.
▷ Calls function RANDOM which chooses mutations randomly based on the given weights.

Input: F: formula, D: Distance (number of mutations to apply), $\mathcal{M}$: set of mutations
Output: F': mutated formula.

$F' \leftarrow F$
for D **do**
  if Uniform **then**
    $M \leftarrow$ RANDOM($\mathcal{M}$, $\left[\frac{1}{|\mathcal{M}|} \cdots \frac{1}{|\mathcal{M}|}\right]$)
  else if Weighted **then**
    $M \leftarrow$ RANDOM($\mathcal{M}, \mathcal{W}$)
  else if Top-K **then**
    $M \leftarrow$ RANDOM($\mathcal{M}_{Top-K}$, $\left[\frac{1}{k} \cdots \frac{1}{k}\right]$)
  $F' \leftarrow M(F')$
return F'

The **Uniform** strategy in Algorithm 2 selects a mutation uniformly at random from the set of all mutations. The **Weighted** strategy chooses mutations probabilistically with the weights calculated in PROFILING based on each mutation's impact. Lastly, **Top-K** one selects a mutation randomly from top-k most impactful mutation candidates, $\mathcal{M}_{Top-K}$, which are identified by PROFILING. For these three attack strategies in Algorithm 2, when we declare a formula F' to have distance D from F, we assure that D is the shortest distance of mutations that can produce F'.

Table 2: A representative subset of reduction rules for equations (where $I \to N \mid S$ and $0 \notin N$)

R_1	$\frac{E_1=E_2}{E_1-E_2=0}$ Shift to L.H.S.
R_2	$\frac{E+I-I}{E}$; $\frac{E_1+I+E_2-I=0}{E_1+E_2=0}$; $\frac{E_1+I-E_2-I=0}{E_1-E_2=0}$ Remove redundant addition
R_3	$\frac{E-I+I}{E}$; $\frac{E_1-I+E_2+I=0}{E_1+E_2=0}$; $\frac{E_1-I-E_2+I=0}{E_1-E_2=0}$ Remove redundant subtraction
R_4	$\frac{E\times I=0}{E=0}$; $\frac{E_1\times I+E_2\times I=0}{E_1+E_2=0}$; $\frac{E_1\times I-E_2\times I=0}{E_1-E_2=0}$ Remove redundant multiplication
R_5	$\frac{E/I=0}{E=0}$; $\frac{E_1/I+E_2/I=0}{E_1+E_2=0}$; $\frac{E_1/I-E_2/I=0}{E_1-E_2=0}$ Remove redundant division

6 Prompt Pre-processing with Formula Reduction

To improve syntactic robustness, we add a preprocessing step where, instead of feeding a prompt with a mathematical formula to the LLM-based code generator as-is, we generate a reduced version of that formula. Then, we feed the reduced prompt with the same English text and the reduced mathematical formula to the LLM-based code generator to generate the target code.

We focus on syntactic transformations that reduce the size of a formula, based on the assumption that shorter formulas are simpler and more likely to be handled correctly by LLM-based code generators. Accordingly, prompts with reduced formulas are expected to

improve robustness against adversarial samples. We refer to these semantics-preserving syntactic transformations as *reductions*.

Table 3: A representative subset of reduction rules for expressions, given $I \rightarrow N \mid S$, where $0 \notin N$.

R_1	$\frac{E+I-I}{E}$; $\frac{E_1+I+E_2-I}{E_1+E_2}$; $\frac{E_1+I-E_2-I}{E_1-E_2}$ Remove redundant addition followed by subtraction
R_2	$\frac{E-I+I}{E}$; $\frac{E_1-I+E_2+I}{E_1+E_2}$; $\frac{E_1-I-E_2+I}{E_1-E_2}$ Remove redundant subtraction followed by addition
R_3	$\frac{E\times I/I}{E}$ Remove redundant multiplication/division

We define the size of a formula $|F|$ to be the number of terminal symbols in the formula as described in grammar shown in Figure 3.

Reductions. We formally define reductions as a type of syntactic transformations as follows:

Definition 10. *A* **reduction** *R is a syntactic transformation where for each formula $F \in \mathcal{F}$, $|R(F)| \leq |F|$.*

i.e., a reduction modifies the syntax of the formula without changing its semantics, while the size of the modified formula is either the same or less than the size of the original formula.

We define 9 types of reductions for equations and a representative subset of these reduction rules are shown in Table 2 (rest are presented in the artifact [7]). Reduction rule R_1 positions all nonzero elements of the equation on one side of the equality operator. Reductions R_2 and R_3 show a series of possible applications for removing redundant additions and subtractions—specifically, the two final rules show removal of these redundant operations when a different additive or subtractive term intercedes the redundant operations. Reductions R_4 and R_5 show a series of possible applications for removing redundant divisions and multiplications—this is to show that this removal may not necessarily be from one singular E comprising the full side of the formula but may also be from each individual additive or subtractive term on that side. For expressions, we introduce 7 reduction rules; we detail 3 in Table 3 and present the rest in the artifact [7]. Reductions R_1 and R_2 are similar to equation reductions R_2 and R_3, to remove redundant additions and subtractions. Reduction R_3 removes redundant multiplications and divisions. The other reduction rules include canceling the addition of extra 0's, multiplication by 1's, and other similar simple reductions. (the full set of rules is in the artifact [7]). Besides equation reduction R_1 (which is applied once at the beginning), all reductions are applied in a loop until the formula cannot be simplified further.

7 Implementation and Experimental Design

Figure 4 illustrates the workflow for assessing LLM-based code generators. We next describe our implementation and design.

Prompts: The prompts used in this paper are inspired from diverse scientific computing applications, including solving differential equations for molecular dynamics, as well as linear, polynomial, trigonometric, and logarithmic equations fundamental to techniques such as the Fourier Transform and Fast Fourier Transform (FFT) [41], shape function interpolation [119], and exponential growth or decay modeling. We also include prompts from established benchmarks [25, 43] and contract-based programming [91]. Table 4 summarizes all prompt categories. A *solve* prompt includes an equation and asks to generate code that takes the equation's coefficients as input and outputs the variable value that satisfies it. The *evaluate* prompts instruct the LLM to generate a program that evaluates a mathematical expression given the values of all variables and coefficients. The *differential* prompts involve manipulating differential formulas. The MATH [43] and Mathematical Topics Tree (MaTT) [25] datasets are designed to evaluate automated mathematical reasoning systems: MATH consists of challenging mathematics competition problems, while MaTT spans key topics across both pure and applied mathematics. The Clover [91] dataset, in contrast, focuses on verifiable code generation using LLMs and contains prompts with formal specifications.

Table 4: Summary of the code generation prompts

Prompt Types	Total	Classification 1		Classification 2	
		Expression	Equation	Reasoning	Translation
Solve	7	0	7	7	0
Evaluate	7	7	0	0	7
Differential	6	1	5	6	0
MaTT [25]	6	5	1	2	4
MATH [43]	7	2	5	6	1
Clover [91]	19	0	19	15	4
Total	52	15	37	36	16

Reasoning vs. Translation Prompts: We categorize prompts into reasoning and translation types (Section 2). Reasoning prompts require the LLM to engage in mathematical manipulation and reasoning to generate the appropriate code, as they involve complex tasks that need to be understood and processed based on the mathematical formula. In contrast, translation prompts allow the LLM to directly utilize the mathematical formulas that are in the prompts as part of the generated code, as these formulas can be directly translated into program expressions. Out of the 52 prompts, we have 36 reasoning and 16 translation prompts.

Programming language: We chose C as the target language for our code generation prompts. C and C++ are extensively used for mathematical computing tasks in scientific computing due to their efficiency and performance in handling complex computations [36, 47]. Recent studies have utilized these languages in LLM-based code generation for tasks such as differential equations, numerical methods, and advanced computer science problems [19, 55, 59, 78].

Mutated formula generation: To measure the impact of syntactic transformations of a formula on the generated code, we generate up to 20 mutations per syntactic distance (1–5). At distance 1, the number is often fewer than 20 due to limited applicable mutation operators. Distance 0 corresponds to the original prompt, yielding a single instance. Across 52 prompts, this results in approximately 4,000 mutated prompts in total.

Non-determinism of LLMs: LLMs are inherently non-deterministic. To address this, we experimented with various temperature, top_p, and seed configurations (where applicable) and selected the most deterministic settings, detailed in our artifact [7]. Additionally, we executed the LLM-based code generator five times per prompt to mitigate variability.

Post-processing of the LLM responses: Given a code generation prompt, the LLM responses typically contain the generated code and some English text explaining the generated code. Our

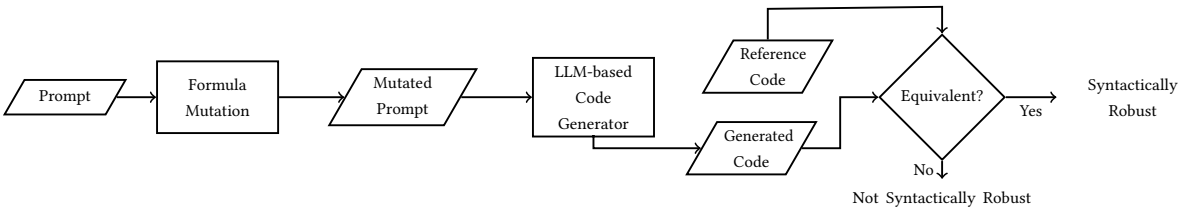

Figure 4: Assessment of Syntactic robustness degree of an LLM (this workflow is applied in a loop for multiple mutations).

implementation takes the generated response from an LLM, eliminates the non-code text, and converts it to a C file. The C files are then compiled into binaries using GCC [2], which is automated in our pipeline. Any C code with syntactic errors is considered non-equivalent with respect to our reference code.

Differential Testing and Code Equivalence: We implement reference solutions for each prompt and employ differential testing to evaluate the correctness and equivalence of the generated code. This involves generating 100 random inputs and comparing the outputs of the generated code with the reference solution. Numerical inputs are sampled from prompt-specific ranges. Our prompts also require LLM-generated outputs to follow a fixed format with either 6-digit or 2-digit precision, depending on the prompt type. For floating-point comparisons, outputs are considered correct if they fall within a predefined epsilon threshold relative to the expected result. This threshold mitigates rounding errors and relaxes our Definition 4 of program equivalence. The epsilon values are provided in our artifact [7]. When differential testing detects non-equivalence between generated code and reference solution, non-equivalence is guaranteed. However, differential testing cannot guarantee equivalence since it explores only a subset of the input space. In a manual review of 100 generated programs, we found no cases where non-equivalent programs were incorrectly classified as equivalent.

Hyperparameter for Attack Strategies: For the Top-K attack strategy, we have chosen the top 50% mutations with the most impact on the syntactic robustness degree of LLM (calculated based on Gpt-3.5). The other strategies choose the mutations randomly based on the weights (Algorithm 2).

Benchmark Models: Our evaluation includes Gpt-3.5, Gpt-4o [3, 77], Llama-3.1-70B [32] and CodeLlama [86]. Gpt-3.5 offers cost-efficiency, while Gpt-4 consistently performs well in both code generation and mathematical reasoning [9, 99]. Llama models are included for their open-source accessibility and low cost. We also evaluate o4-mini [5], a reasoning model that balances performance with reduced computational overhead compared to other OpenAI reasoning models like o3, o1-pro. We experiment with Chain-of-thought (CoT) [100] and Deeply Understanding Problems (DUP) [116] approaches as they are zero-shot prompting techniques that are generalizable, and cost-efficient. For CoT, we use trigger-based prompts (e.g., "Let's solve this step-by-step") to preserve automation [60]. DUP, which addresses semantic misunderstandings for mathematical reasoning, has achieved state-of-the-art results on arithmetic tasks [6] and is adapted here for mathematical code generation. In code generation accuracy, Gpt-3.5 and Gpt-4 frameworks outperform others, achieving $\geq 85\%$ on HumanEval [21], a trend reflected in our results; thus, framework-based techniques are applied only to Gpt models, excluding Llama models due to lower performance.

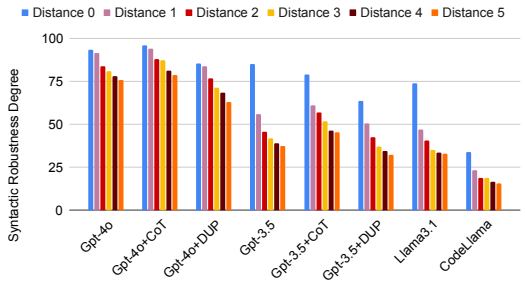

Figure 5: Syntactic Robustness Degree Vs. Distance for LLMs

Artifact: Our artifact, including implementation, datasets, results, mutation and reduction rules, is available in GitHub [7].

8 Experimental Evaluation

We address four research questions:

RQ1: Assessment 1: Does LLM-based code generation achieve syntactic robustness?

RQ2: Assessment 2: Do LLMs show different levels of syntactic robustness for translation and reasoning prompts in code generation?

RQ3: Exploitation: Do the attacks help in decreasing the syntactic robustness of the LLM-based code generation?

RQ4: Mitigation: Does prompt pre-processing with formula reduction improve the syntactic robustness of code generation and help in mitigating our attacks?

Below we present the results of our experiments and discuss the four research questions. For our assessment (RQ1-2), we use the Gpt-3.5, Gpt-4o, Llama-3.1-70B, CodeLlama, Gpt-based framework models and we use the 33 prompts from Table 4 (Row 1-5). For RQ3, we have chosen Gpt-3.5 as it performs better than open source models and cheaper than other Gpt-based models. To evaluate our reduction technique (RQ4), we have extended our dataset to Clover dataset and also included o4-mini model.

RQ1: Assessment 1: Figure 5 illustrates the syntactic robustness percentages for all foundational and framework models across different syntactic distances, based on the average robustness scores across the 33 prompts used. Distance 0 shows the robustness of code generation for the original formula, while Distance 1 through Distance 5 shows the average syntactic robustness degrees for the mutated variations at each respective syntactic distance. As shown in Figure 5, the syntactic robustness degree decreases consistently across all models as syntactic distance increases. Despite varying performance levels, none of the models demonstrate complete

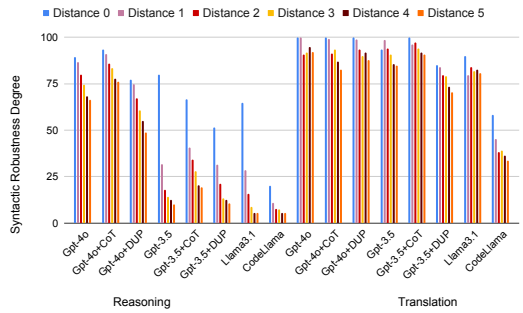

Figure 6: Syntactic robustness degree of reasoning prompts (left 8) and translation prompts (right 8)

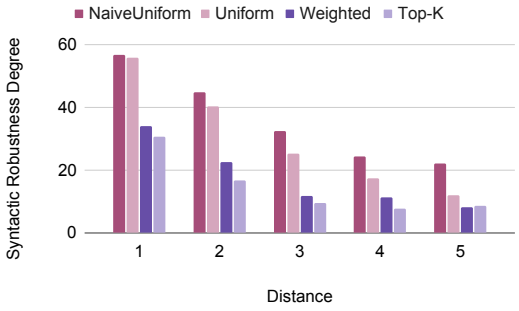

Figure 7: Syntactic robustness degree of different attacks for reasoning prompts (Gpt-3.5)

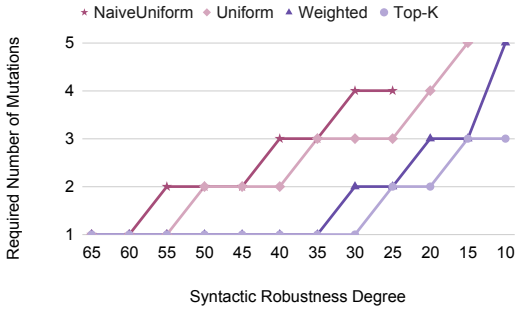

Figure 8: Required number of mutations for different attacks

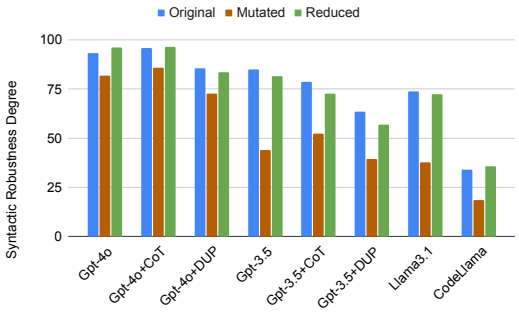

Figure 9: Syntactic robustness degree with reduced form

syntactic robustness. Gpt-4o+CoT performs best, with an average syntactic robustness of 87.4%, followed by Gpt-4o with 83.87%.

Moreover, Figure 5 shows average syntactic robustness percentages across mutation distances from 0 to 5 and clearly demonstrates that as the number of mutation increases, syntactic robustness decreases across all foundational and framework-based models. Notably, the rate of decline varies among different LLMs. Gpt-4o+CoT shows the slowest decline in syntactic robustness degree per mutation distance: -3.45% on average. Gpt-3.5 experiences a more significant drop of -9.52%, as its syntactic robustness degree with the original prompt is higher than the robustness degree of mutated prompts. In summary, the syntactic robustness decreases for all models as mutation distance increases. Thus, we conclude that **LLMs do not achieve syntactic robustness** and **increasing number of mutations (equivalently, increasing syntactic distance) leads to a decrease in syntactic robustness**.

RQ2: Assessment 2: Figure 6 shows the average syntactic robustness of all reasoning prompts (left side) and translation prompts (right side) across eight models. The results indicate that translation prompts consistently exhibit higher robustness than reasoning prompts. As mutation distance increases, robustness declined more steeply for reasoning prompts. On average, reasoning prompts achieve a syntactic robustness of 42.86%, compared to 83.80% for translation prompts. The robustness drops by approximately 30% for reasoning prompts and 8% for translation ones relative to their original forms. Based on our experimental evaluation, we conclude that **LLMs exhibit different levels of syntactic robustness across translation and reasoning prompts, with greater robustness for translation prompts**.

RQ3: Exploitation: As described in RQ2, translation prompts are more robust than the reasoning prompts. Attacks on the translation prompts also show similar results (details in artifact [7]). Thus, to answer this research question, we only focus on reasoning prompts. Figure 7 shows the impact of our attack strategies on Gpt-3.5 model. It illustrates that for all attack strategies, increasing syntactic distance decreases the syntactic robustness degree, which follows the findings in RQ1. On average, across all distances, the syntactic robustness degree is 36.13%, 30.17%, 17.54% and 14.64% respectively for NaiveUniform, Uniform, Weighted and Top-K. The Weighted and Top-K attack strategies are most effective in reducing syntactic robustness degree across all tested distances. In Figure 8, we show how many mutations are required for different attack strategies to reduce the syntactic robustness degree to a target level (x-axis). With NaiveUniform, 4 mutations are required to reduce syntactic robustness to 25% and more than 5 would be necessary for further reduction. On the other hand, similar syntactic robustness can be achieved with a lower number of mutations for the other attacks (3 for Uniform and 2 for Weighted and Top-K). To reduce the syntactic robustness down to 10%, Weighted requires 5 mutations whereas Top-K requires only 3 mutations. Thus, Figure 8 illustrates that Top-K is most effective in getting the highest reduction in syntactic robustness with the least number of mutations. Our experimental evaluation shows that **our attack strategies are effective in decreasing the syntactic robustness degree**.

RQ4: Mitigation: We demonstrate the mitigating effect of applying our reduction rules to mutated formulas before querying the LLM-based code generator. Figure 9 presents the syntactic robustness of the models for the original formulas, the average

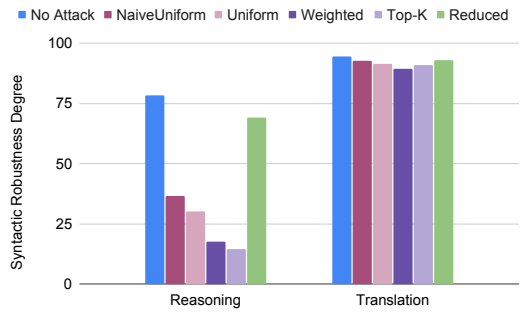

Figure 10: Mitigation of attack strategies (Gpt-3.5)

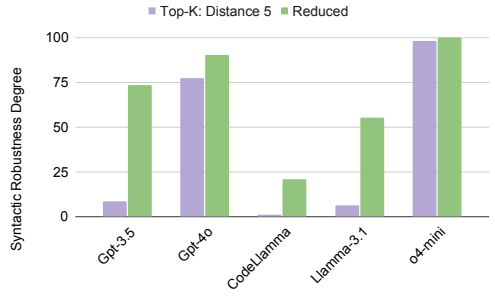

Figure 11: Impact of Top-K attacks on reasoning prompts at distance 5

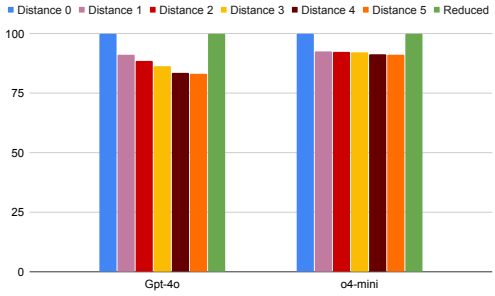

Figure 12: Top-K attack for Clover Dataset

robustness for mutated formulas, and the robustness of the reduced prompts. Across all foundational and framework models, the reduced prompts exhibit robustness degrees comparable to the original prompts and show notable improvement over the mutated prompts. The improvement in syntactic robustness when comparing reduced prompts to mutated prompts is substantial, with increases between 10.63% and 37.55%, resulting in an overall average improvement of 20.37%. Among all foundational and framework models, the reduced prompts for Gpt-4o+CoT perform the best on average for all the prompts we have used.

For Gpt-4o and Gpt-4o+CoT, the reduced prompts show improvements of 2.73% and 0.6%, respectively, even over the original prompts. Moreover, CodeLlama achieves an improvement of 5.1% for reduced prompts compared to the original ones. In contrast, other models show a slight degradation in performance with the reduced prompts compared to the original prompts.

Figure 10 shows the syntactic robustness degrees of different attacks along with the original and our reduced prompts for both reasoning and translation prompts in Gpt-3.5. It is evident that our pre-processing step helps in mitigating the impact of attacks, especially for reasoning prompts. Moreover, in Figure 11, we have shown the results for the Top-K attack at distance 5 and the robustness for the corresponding reduced prompts across different LLMs (including o4-mini). In all cases, reduction improves the syntactic robustness. Even though o4-mini is very robust against the attack, there is a 1.95% drop in robustness which can be mitigated using our reduction step. Figure 12 also shows the result across different distances for Top-K attack on Gpt-4o and o4-mini on Clover dataset. This result on a different dataset also follows the trend of decreasing robustness with distances which was successfully mitigated by our reduction steps. Overall, our reduction rules are effective in pre-processing the prompts for reliable code generation, and the experimental results confirm that **applying our pre-processing step to simplify formulas enhances the syntactic robustness of LLM-based code generation**.

Discussion. All foundational models were evaluated using the same prompts from our dataset. However, the framework models, such as Gpt+CoT and Gpt+DUP, require additional text in the prompts. Although this extra text does not alter the instructions about the input and output format of the code, we observed that the framework models, particularly Gpt-3.5+DUP and Gpt-3.5+CoT, often fail to follow the given instructions. For instance, Gpt-3.5+DUP produced only the instruction to generate code in the comment section of the C program in 22 cases, while Gpt-3.5+CoT did the same in 3 cases. Additionally, in 18 cases, Gpt-3.5+DUP generated code without incorporating the input parameters, which disrupts the automated differential testing process. This issue is even more pronounced in Llama-based models, where code is generated without user input processing in approximately 150 cases for Llama-3.1-70B and 600 cases for CodeLlama. These factors negatively impact the overall performance of Llama and the framework-based models in code generation. We also observe that Llama-3.1 and CodeLlama often choose to present results in other languages such as C++ and Python despite clear instructions to use C, which lowers the syntactic robustness. Although we selected o4-mini for its efficiency over larger reasoning models like o3 and o1-pro, our experiments reveal that it still takes approximately 4× more time than Gpt-4o and about 10× more time than Gpt-3.5 for comparable tasks.

Threats to Validity. We discuss possible internal and external threats to validity and the efforts we have made to mitigate them. *Internal:* There are a few possible internal threats to validity within our experimental design. First, there is the possibility that the Gpt API (for Gpt-3.5, Gpt-4o, and o4-mini) may be learning from prior requests we send. However, we believe this is not the case, as we have switched APIs during these experiments multiple times and found no observable difference between code returned from a previously-used API and code returned from a fresh one. Second, we note that differential testing cannot prove equivalence, and due to needing an epsilon value in 31 prompts to avoid incorrectly marking nonequivalence due to floating point error, it is possible some incorrect results could be marked correct. This is a possible risk for any testing-based equivalence checking approach. However, we have manually

investigated a subset of LLM generated programs and found no case where differential testing falsely marked two non-equivalent codes as equivalent. Moreover, when we declare generated code non-equivalent, non-equivalence is guaranteed since we produce a test case that demonstrates the non-equivalence.
External: Analyzing all formulas or mutations is not possible. However, we believe that our evaluation shows syntactic robustness issues of LLMs for an important class of formulas and mutations. Furthermore, our evaluation and attack approaches can be applied to different classes of formulas and syntactically equivalent forms of those formulas using the framework we present in this paper.

Limitations. Our work focuses on the robustness of LLMs specifically for code generation prompts involving mathematical formulas and constraints. Consequently, our transformation and reduction techniques are also limited to prompts containing such mathematical formulas. However, mathematical computation–based code generation is fundamental to many domains, including scientific computing and contract-based programming. While our benchmark consists of a limited number of prompts, we believe they represent a diverse set of prompts with mathematical formulas and can be further extended. Additionally, while we evaluate code generation for the C programming language, our approach can be extended to assess other LLMs for different programming languages.

9 Related Work

Prior work on LLM-based code generation primarily focuses on improving performance on software engineering tasks [21, 68, 76, 104], often evaluated on benchmarks such as HumanEval, MBPP, and APPS [14, 21, 42]. In parallel, substantial research studies mathematical problem solving with LLMs [40, 94, 116], using datasets such as GSM8K, SVAMP, and MultiArith [23, 80, 85]. Recent work in scientific computing and numerical analysis [10, 31, 55, 88] shows that these capabilities can be combined to solve complex tasks, including contract programming [91]. However, prior work does not examine the robustness of LLM-based code generators under syntactic variations, which is the focus of this work.

Robustness has been studied for classification and regression neural networks, using techniques such as symbolic reasoning [17, 56, 92], abstraction [89, 98], and testing/fuzzing [15, 103].

Prior work has examined the correctness of LLMs for code generation tasks [8, 16, 29, 38, 71, 90, 96, 108]. Other studies investigate robustness against non-determinism in LLMs [79], perturbations in natural language descriptions [50, 74], and variations in coding components [18, 84, 104, 106, 114]. ReCode [97] introduces a robustness evaluation benchmark for code generation by perturbing prompts, including both natural language components and code segments via partial code syntax refactoring in code completion settings. Semantic preservation in ReCode relies on human annotation. In contrast, our work evaluates robustness through perturbations of mathematical formulas that are unambiguously semantics-preserving and do not require human annotation. Moreover, ReCode's prompts, transformations, and datasets do not target mathematical reasoning and do not include prompts containing mathematical formulas. Robustness has also been studied for mathematical problem solving by LLMs [11, 37, 61, 67, 109, 117] which are not focused on code generation by LLMs. Compared to prior work and surveys [33, 107], our work formally defines syntactic robustness and focuses on prompts containing mathematical formulas, targeting the robustness of LLM-based code generation.

Our syntactic transformation approach, which mutates mathematical formulas, is conceptually similar to metamorphic testing [22], a property-based technique also applied to LLMs [12, 13]. For example, LAMPION [13] generates equivalent code snippets to test the robustness of LLM-based program analysis models. In contrast, we target LLM-based code generation.

Recent multi-agent code generation frameworks like AgentCoder, LDB, and MapCoder [46, 48, 115] achieve over 90% accuracy on benchmarks such as HumanEval and MBPP [6], but require multiple LLM queries per prompt [1, 54]. Studies show that when both accuracy and cost are considered, these frameworks do not outperform baseline foundational models [54], and they also rely on input-output test cases [42, 57]. In contrast, our work focuses on prompt-engineering frameworks that rely solely on prompts.

Beyond multi-agent methods, significant progress has been made with prompt engineering frameworks [8, 30, 39, 52, 62, 64, 66, 70, 73, 75, 101, 112]. CoT has been applied to code generation by decomposing problems into sequential, branch, and loop structures [63, 87]. Other reasoning-oriented prompting strategies, such as DUP and AceCoder [11, 26, 53, 65, 102, 116], also achieve strong performance, with DUP attaining state-of-the-art results on arithmetic reasoning benchmarks [6]. Since our dataset combines mathematical reasoning and code generation, we evaluate both CoT and DUP.

Prior robustness attacks on deep code models [83] primarily target misclassification in tasks like authorship attribution [34, 72]. In contrast, we focus on LLM-based code generators, which differ from classifiers. While black-box prompt attacks [105, 117] exist, they do not address code generation. Other work has explored code completion or generation attacks using adversarial comments (INSEC [49]) or non-functional code changes [82]. Our approach instead mutates mathematical formulas to produce semantically equivalent prompts, exposing failures in syntactic robustness.

There are also works which focus on training strategies or architectural changes for improving the robustness of LLM code generations [20, 28, 81, 113]. Our approach is distinct as we focus on black-box prompt pre-processing for improving this robustness, without altering or retraining the models.

10 Conclusion

The use of LLM-based code generation is expanding in domains involving mathematical computations. In this paper, we expose limitations of state-of-the-art LLMs when handling prompts with mathematical formulas under semantic-preserving syntactic mutations. We formalize this concept as syntactic robustness and evaluate multiple benchmark models using both original and mutated prompts. Our results show a significant decrease in robustness as syntactic distance increases, with reasoning-intensive prompts particularly affected. We further design and compare attack strategies that effectively reduce robustness. To mitigate this issue, we introduce a formula reduction-based prompt pre-processing technique that consistently improves robustness across all evaluated models.

References

[1] [n. d.]. AI Leaderboards are no longer useful. https://www.aisnakeoil.com/p/ai-leaderboards-are-no-longer-useful#footnote-3-144156093.
[2] [n. d.]. gcc. https://gcc.gnu.org/.
[3] [n. d.]. GPT-3.5 Turbo. https://platform.openai.com/docs/models/gpt-3-5-turbo.
[4] [n. d.]. HackerRank. https://www.hackerrank.com/contests/projecteuler/challenges.
[5] [n. d.]. o4-mini. https://platform.openai.com/docs/models/o4-mini.
[6] [n. d.]. State of the art code generation benchmarks. https://paperswithcode.com/task/code-generation.
[7] 2025. https://github.com/laboni68/LLMEvaluation.git.
[8] Toufique Ahmed, Kunal Suresh Pai, Premkumar Devanbu, and Earl T Barr. 2023. Improving few-shot prompts with relevant static analysis products. *arXiv:2304.06815* (2023).
[9] Janice Ahn, Rishu Verma, et al. 2024. Large Language Models for Mathematical Reasoning: Progresses and Challenges. In *Proceedings of the 18th Conference of the European Chapter of the Association for Computational Linguistics: Student Research Workshop*. doi:10.18653/v1/2024.eacl-srw.17
[10] Microsoft Research AI4Science and Microsoft Azure Quantum. 2023. The Impact of Large Language Models on Scientific Discovery: a Preliminary Study using GPT-4. arXiv:2311.07361 [cs.CL] https://arxiv.org/abs/2311.07361
[11] Ujjwala Anantheswaran, Himanshu Gupta, et al. 2024. Investigating the Robustness of LLMs on Math Word Problems. *arXiv:2406.15444* (2024).
[12] Leonhard Applis, Annibale Panichella, and Ruben Marang. 2023. Searching for Quality: Genetic Algorithms and Metamorphic Testing for Software Engineering ML. In *Proceedings of the Genetic and Evolutionary Computation Conference*.
[13] Leonhard Applis, Annibale Panichella, and Arie van Deursen. 2021. Assessing robustness of ml-based program analysis tools using metamorphic program transformations. In *International Conference on Automated Software Engineering*.
[14] Jacob Austin, Augustus Odena, et al. 2021. Program synthesis with large language models. *arXiv:2108.07732* (2021).
[15] Teodora Baluta, Zheng Leong Chua, Kuldeep S Meel, and Prateek Saxena. 2021. Scalable quantitative verification for deep neural networks. In *IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE.
[16] Ali Borji. 2023. A categorical archive of chatgpt failures. *arXiv:2302.03494* (2023).
[17] Rudy Bunel, P Mudigonda, et al. 2020. Branch and bound for piecewise linear neural network verification. *Journal of Machine Learning Research* 21 (2020).
[18] Alessio Buscemi. 2023. A comparative study of code generation using chatgpt 3.5 across 10 programming languages. *arXiv:2308.04477* (2023).
[19] Emir Catir, Robin Claesson, and Rodothea Myrsini Tsoupidi. 2025. Evaluating Code Generation of LLMs in Advanced Computer Science Problems. *arXiv preprint arXiv:2504.14964* (2025).
[20] Saikat Chakraborty, Toufique Ahmed, et al. 2022. Natgen: generative pre-training by "naturalizing" source code. In *the 30th ACM joint european software engineering conference and symposium on the foundations of software engineering*.
[21] Mark Chen, Jerry Tworek, et al. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374* (2021).
[22] Tsong Y Chen, Shing C Cheung, and Shiu Ming Yiu. 2020. Metamorphic testing: a new approach for generating next test cases. *arXiv:2002.12543* (2020).
[23] Karl Cobbe, Vineet Kosaraju, , et al. 2021. Training verifiers to solve math word problems. *arXiv:2110.14168* (2021).
[24] Tristan Coignion, Clément Quinton, and Romain Rouvoy. 2024. A performance study of llm-generated code on leetcode. In *Proceedings of the 28th International Conference on Evaluation and Assessment in Software Engineering*. 79–89.
[25] Arash Gholami Davoodi, Seyed Pouyan Mousavi Davoudi, and Pouya Pezeshkpour. [n. d.]. LLMs Are Not Intelligent Thinkers: Introducing Mathematical Topic Tree Benchmark for Comprehensive Evaluation of LLMs. In *Proceedings of the 2025 Conference of the Nations of the Americas Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL 2025*. doi:10.18653/V1/2025.NAACL-LONG.161
[26] Aniket Didolkar, Anirudh Goyal, et al. 2024. Metacognitive Capabilities of LLMs: An Exploration in Mathematical Problem Solving. In *Advances in Neural Information Processing Systems, NeurIPS*.
[27] Xi Ding, Yuan Huang, Xiangping Chen, and Jing Bian. 2024. Adversarial Attack and Robustness Improvement on Code Summarization. In *Proceedings of the 28th International Conference on Evaluation and Assessment in Software Engineering*.
[28] Yangruibo Ding, Jinjun Peng, et al. 2024. SemCoder: Training Code Language Models with Comprehensive Semantics Reasoning. In *The Thirty-eighth Annual Conference on Neural Information Processing Systems*.
[29] Tuan Dinh, Jinman Zhao, et al. 2024. Large language models of code fail at completing code with potential bugs. *Advances in Neural Information Processing Systems* (2024).
[30] Jean-Baptiste Döderlein, Nguessan Hermann Kouadio, et al. 2025. Piloting Copilot, Codex, and StarCoder2: Hot temperature, cold prompts, or black magic? *J. Syst. Softw.* (2025).
[31] Mengge Du, Yuntian Chen, et al. 2024. LLM4ED: Large Language Models for Automatic Equation Discovery. *arXiv:2405.07761* (2024).
[32] Abhimanyu Dubey, Abhinav Jauhri, et al. 2024. The llama 3 herd of models. *arXiv:2407.21783* (2024).
[33] Angela Fan, Beliz Gokkaya, et al. 2023. Large Language Models for Software Engineering: Survey and Open Problems. In *IEEE/ACM International Conference on Software Engineering: Future of Software Engineering, ICSE-FoSE*. IEEE, 31–53.
[34] Fengjuan Gao, Yu Wang, and Ke Wang. 2023. Discrete adversarial attack to models of code. *Proceedings of the ACM on Programming Languages (PLDI)*.
[35] Timon Gehr, Matthew Mirman, et al. 2018. Ai2: Safety and robustness certification of neural networks with abstract interpretation. In *Symposium on Security and Privacy (SP)*. IEEE.
[36] Georg Hager and Gerhard Wellein. 2010. *Introduction to High Performance Computing for Scientists and Engineers* (1st ed.). CRC Press, Inc., USA.
[37] Yuren Hao, Xiang Wan, and Chengxiang Zhai. 2025. An Investigation of Robustness of LLMs in Mathematical Reasoning: Benchmarking with Mathematically-Equivalent Transformation of Advanced Mathematical Problems. *arXiv preprint arXiv:2508.08833* (2025).
[38] Fusen He, Juan Zhai, and Minxue Pan. 2024. Beyond Code Generation: Assessing Code LLM Maturity with Postconditions. *arXiv:2407.14118* (2024).
[39] Jingxuan He and Martin Vechev. 2023. Controlling large language models to generate secure and vulnerable code. *arXiv e-prints* (2023).
[40] Joy He-Yueya, Gabriel Poesia, et al. 2023. Solving math word problems by combining language models with symbolic solvers. *arXiv:2304.09102* (2023).
[41] Paul S. Heckbert. 1998. Fourier Transforms and the Fast Fourier Transform (FFT) Algorithm. https://api.semanticscholar.org/CorpusID:6022157
[42] Dan Hendrycks, Steven Basart, et al. 2021. Measuring Coding Challenge Competence With APPS. In *Proceedings of the Neural Information Processing Systems Track on Datasets and Benchmarks 1, NeurIPS Datasets and Benchmarks*.
[43] Dan Hendrycks, Collin Burns, et al. 2021. Measuring Mathematical Problem Solving With the MATH Dataset. *35th Conference on Neural Information Processing Systems (NeurIPS 2021) Track on Datasets and Benchmarks* (03 2021).
[44] Max Hort, Linas Vidziunas, and Leon Moonen. 2025. Semantic-Preserving Transformations as Mutation Operators: A Study on Their Effectiveness in Defect Detection. In *2025 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. IEEE, 337–346.
[45] Xinyi Hou, Yanjie Zhao, et al. 2024. Large Language Models for Software Engineering: A Systematic Literature Review. *ACM Trans. Softw. Eng. Methodol.*.
[46] Dong Huang, Qingwen Bu, et al. 2023. AgentCoder: Multi-Agent-based Code Generation with Iterative Testing and Optimisation. *arXiv:2312.13010* (2023).
[47] A Iserles. 1989. Numerical recipes in C—the art of scientific computing, by WH Press, BP Flannery, SA Teukolsky and WT Vetterling. Pp 735.£ 27· 50. 1988. ISBN 0-521-35465-X (Cambridge University Press). *The Mathematical Gazette* (1989).
[48] Md. Ashraful Islam, Mohammed Eunus Ali, and Md. Rizwan Parvez. 2024. MapCoder: Multi-Agent Code Generation for Competitive Problem Solving. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics ACL 2024*. Association for Computational Linguistics.
[49] Slobodan Jenko, Jingxuan He, Niels Mündler, Mark Vero, and Martin Vechev. 2024. Black-Box Adversarial Attacks on LLM-Based Code Completion. *arXiv preprint arXiv:2408.02509* (2024).
[50] Ellen Jiang, Edwin Toh, et al. 2022. Discovering the syntax and strategies of natural language programming with generative language models. In *Proceedings of the 2022 CHI Conference on Human Factors in Computing Systems*. 1–19.
[51] Juyong Jiang, Fan Wang, et al. 2024. A Survey on Large Language Models for Code Generation. arXiv:2406.00515 [cs.CL] https://arxiv.org/abs/2406.00515
[52] Shuyang Jiang, Yuhao Wang, and Yu Wang. 2023. Selfevolve: A code evolution framework via large language models. *arXiv:2306.02907* (2023).
[53] Xue Jiang, Yihong Dong, et al. 2023. Self-planning Code Generation with Large Language Models. *ACM Transactions on Software Engineering and Methodology*.
[54] Sayash Kapoor, Benedikt Stroebl, et al. 2025. AI Agents That Matter. *Trans. Mach. Learn. Res.* (2025).
[55] Ali Kashefi and Tapan Mukerji. 2023. ChatGPT for programming numerical methods. *Journal of Machine Learning for Modeling and Computing* 4, 2 (2023).
[56] Guy Katz, Derek A Huang, et al. 2019. The marabou framework for verification and analysis of deep neural networks. In *International Conference on Computer Aided Verification*.
[57] Mohammad Abdullah Matin Khan, M. Saiful Bari, et al. 2024. XCodeEval: An Execution-based Large Scale Multilingual Multitask Benchmark for Code Understanding, Generation, Translation and Retrieval. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics*.
[58] Saso Koceski, Natasa Koceska, et al. 2023. Can ChatGPT be used for solving ordinary differential equations. *International Electronic Journal of Mathematics Education* 6 (12 2023). https://eprints.ugd.edu.mk/32896/
[59] Saso Koceski, Natasa Koceska, Limonka Koceva Lazarova, Marija Miteva, and Biljana Zlatanovska. 2023. Using ChatGPT for numerical solution of first and second order ordinary differential equations. Presentation at the 10th Jubilee International Conference of FMNS (FMNS-2023), Blagoevgrad, Bulgaria.
[60] Takeshi Kojima, Shixiang Shane Gu, et al. 2022. Large language models are zero-shot reasoners. *Advances in neural information processing systems* (2022).
[61] Vivek Kumar, Rishabh Maheshwary, and Vikram Pudi. 2021. Adversarial Examples for Evaluating Math Word Problem Solvers. In *Findings of the Association*

for Computational Linguistics: EMNLP 2021.

[62] Jia Li, Ge Li, Yongmin Li, and Zhi Jin. 2023. Enabling programming thinking in large language models toward code generation. *arXiv:2305.06599* (2023).

[63] Jia Li, Ge Li, Yongmin Li, and Zhi Jin. 2023. Structured chain-of-thought prompting for code generation. *ACM Transactions on Software Engineering and Methodology* (2023).

[64] Jia Li, Yongmin Li, et al. 2023. Skcoder: A sketch-based approach for automatic code generation. In *ACM 45th International Conference on Software Engineering*.

[65] Jia Li, Yunfei Zhao, et al. 2024. AceCoder: An Effective Prompting Technique Specialized in Code Generation. *ACM Trans. Softw. Eng. Methodol.* (2024).

[66] Jia Li, Yunfei Zhao, Yongmin Li, Ge Li, and Zhi Jin. 2023. Towards enhancing in-context learning for code generation. *arXiv:2303.17780* (2023).

[67] Qintong Li, Leyang Cui, Xueliang Zhao, Lingpeng Kong, and Wei Bi. 2024. GSM-Plus: A Comprehensive Benchmark for Evaluating the Robustness of LLMs as Mathematical Problem Solvers. *arXiv:2402.19255* (2024).

[68] Yujia Li, David Choi, et al. 2022. Competition-level code generation with alphacode. *Science* 378, 6624 (2022), 1092–1097.

[69] Yixuan Li, Julian Parsert, and Elizabeth Polgreen. 2024. Guiding Enumerative Program Synthesis with Large Language Models. In *Computer Aided Verification*, Arie Gurfinkel and Vijay Ganesh (Eds.). Springer Nature Switzerland, Cham.

[70] Chao Liu, Bao Xuanlin, Hongyu Zhang, Neng Zhang, Haibo Hu, Xiaohong Zhang, and Meng Yan. 2023. Improving ChatGPT Prompt for Code Generation. *arXiv preprint arXiv:2305.08360* (05 2023).

[71] Jiawei Liu, Chunqiu Steven Xia, et al. 2024. Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation. *Advances in Neural Information Processing Systems* (2024).

[72] Qianjun Liu, Shouling Ji, Changchang Liu, and Chunming Wu. 2021. A practical black-box attack on source code authorship identification classifiers. *IEEE Transactions on Information Forensics and Security* 16 (2021), 3620–3633.

[73] Ggaliwango Marvin, Nakayiza Hellen, Daudi Jjingo, and Joyce Nakatumba-Nabende. 2023. Prompt engineering in large language models. In *International conference on data intelligence and cognitive informatics*. Springer.

[74] Antonio Mastropaolo, Luca Pascarella, et al. 2023. On the Robustness of Code Generation Techniques: An Empirical Study on GitHub Copilot. In *45th IEEE/ACM International Conference on Software Engineering, ICSE 2023*.

[75] Fangwen Mu, Lin Shi, et al. 2024. ClarifyGPT: A Framework for Enhancing LLM-Based Code Generation via Requirements Clarification. *Proceedings of the ACM on Software Engineering* FSE (2024).

[76] Erik Nijkamp, Bo Pang, et al. 2023. CodeGen: An Open Large Language Model for Code with Multi-Turn Program Synthesis. In *The Eleventh International Conference on Learning Representations, ICLR*.

[77] OpenAI, Josh Achiam, Steven Adler, et al. 2024. GPT-4 Technical Report.

[78] Giuseppe Orlando. 2023. Assessing ChatGPT for coding finite element methods. *Journal of Machine Learning for Modeling and Computing* 4 (07 2023).

[79] Shuyin Ouyang, Jie Zhang, et al. 2025. An Empirical Study of the Non-Determinism of ChatGPT in Code Generation. *ACM Trans. Softw. Eng. Methodol.*.

[80] Arkil Patel, Satwik Bhattamishra, and Navin Goyal. 2021. Are NLP Models really able to Solve Simple Math Word Problems?. In *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT*.

[81] Kexin Pei, Weichen Li, Qirui Jin, Shuyang Liu, Scott Geng, Lorenzo Cavallaro, Junfeng Yang, and Suman Jana. 2024. Exploiting Code Symmetries for Learning Program Semantics. In *Forty-first International Conference on Machine Learning*.

[82] Qiulu Peng, Chi Zhang, et al. 2025. Random Perturbation Attack on LLMs for Code Generation. In *4th IEEE/ACM International Conference on AI Engineering - Software Engineering for AI, CAIN*. doi:10.1109/CAIN66642.2025.00052

[83] Yubin Qu, Song Huang, and Yongming Yao. 2024. A survey on robustness attacks for deep code models. *Automated Software Engineering* 31 (2024), 65.

[84] Fazle Rabbi, Zishuo Ding, and Jinqiu Yang. 2025. A Multi-Language Perspective on the Robustness of LLM Code Generation. *arXiv preprint arXiv:2504.19108*.

[85] Subhro Roy and Dan Roth. 2015. Solving General Arithmetic Word Problems. In *2015 Conference on Empirical Methods in Natural Language Processing, EMNLP*.

[86] Baptiste Roziere, Jonas Gehring, et al. 2023. Code llama: Open foundation models for code. *arXiv:2308.12950* (2023).

[87] Zhihong Shao, Yeyun Gong, et al. 2023. Synthetic prompting: Generating chain-of-thought demonstrations for large language models. In *International Conference on Machine Learning*. PMLR.

[88] Parshin Shojaee, Kazem Meidani, et al. 2025. LLM-SR: Scientific Equation Discovery via Programming with Large Language Models. In *The Thirteenth International Conference on Learning Representations, ICLR 2025*.

[89] Gagandeep Singh, Timon Gehr, Markus Püschel, and Martin Vechev. 2019. An abstract domain for certifying neural networks. *Proceedings of the ACM on Programming Languages* 3, POPL (2019), 1–30.

[90] Claudio Spiess, David Gros, et al. 2025. Calibration and Correctness of Language Models for Code. In *Proceedings of the IEEE/ACM 47th International Conference on Software Engineering*. doi:10.1109/ICSE55347.2025.00040

[91] Chuyue Sun, Ying Sheng, Oded Padon, and Clark Barrett. 2024. Clover: Closed-Loop Verifiable Code Generation. In *AI Verification*, Guy Avni, Mirco Giacobbe, Taylor T. Johnson, Guy Katz, Anna Lukina, Nina Narodytska, and Christian Schilling (Eds.). Springer Nature Switzerland, Cham, 134–155.

[92] Youcheng Sun, Min Wu, et al. 2018. Concolic testing for deep neural networks. In *the 33rd International Conference on Automated Software Engineering*.

[93] Zhao Tian, Junjie Chen, and Zhi Jin. 2023. Code difference guided adversarial example generation for deep code models. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 850–862.

[94] Jonathan Uesato, Nate Kushman, et al. 2022. Solving math word problems with process-and outcome-based feedback. *arXiv:2211.14275* (2022).

[95] Sarthak Vishnu, Sahil, and Naman Garg. 2025. Unveiling the Role of GPT-4 in Solving LeetCode Programming Problems. *Computer Applications in Engineering Education* 33, 1 (2025), e22815.

[96] Karen D Wang, Eric Burkholder, et al. 2024. Examining the potential and pitfalls of ChatGPT in science and engineering problem-solving. In *Frontiers in Education*, Vol. 8. Frontiers Media SA.

[97] Shiqi Wang, Zheng Li, et al. 2023. ReCode: Robustness Evaluation of Code Generation Models. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics*. doi:10.18653/v1/2023.acl-long.773

[98] Shiqi Wang, Kexin Pei, et al. 2018. Formal security analysis of neural networks using symbolic intervals. In *27th {USENIX} Security Symposium*.

[99] Zhilong Wang, Lan Zhang, et al. 2024. How Does Naming Affect LLMs on Code Analysis Tasks? arXiv:2307.12488 [cs.CR] https://arxiv.org/abs/2307.12488

[100] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Xia, et al. 2022. Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems* 35 (2022).

[101] Jules White, Sam Hays, Quchen Fu, Jesse Spencer-Smith, and Douglas C Schmidt. 2023. Chatgpt prompt patterns for improving code quality, refactoring, requirements elicitation, and software design. *arXiv:2303.07839* (2023).

[102] Yiran Wu, Feiran Jia, et al. 2024. MathChat: Converse to Tackle Challenging Math Problems with LLM Agents. In *ICLR 2024 Workshop on Large Language Model (LLM) Agents*.

[103] Xiaofei Xie, Lei Ma, et al. 2019. Deephunter: a coverage-guided fuzz testing framework for deep neural networks. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 146–157.

[104] Frank F Xu, Uri Alon, Graham Neubig, and Vincent Josua Hellendoorn. 2022. A systematic evaluation of large language models of code. In *Proceedings of the 6th ACM SIGPLAN International Symposium on Machine Programming*. 1–10.

[105] Xilie Xu, Keyi Kong, et al. 2024. An LLM can Fool Itself: A Prompt-Based Adversarial Attack. In *12th International Conference on Learning Representations*.

[106] Ming Yan, Junjie Chen, et al. 2023. Coco: Testing code generation systems via concretized instructions. *arXiv:2308.13319* (2023).

[107] Zhou Yang, Zhensu Sun, Terry Yue Zhuo, Prem Devanbu, and David Lo. 2024. Robustness, Security, Privacy, Explainability, Efficiency, and Usability of Large Language Models for Code. *ArXiv* abs/2403.07506 (2024).

[108] Burak Yetiştiren, Işık Özsoy, et al. 2023. Evaluating the Code Quality of AI-Assisted Code Generation Tools: An Empirical Study on GitHub Copilot, Amazon CodeWhisperer, and ChatGPT. *arXiv.2304.10778* (04 2023).

[109] Boning Zhang, Chengxi Li, and Kai Fan. 2024. MARIO Eval: Evaluate Your Math LLM with your Math LLM–A mathematical dataset evaluation toolkit. *arXiv:2404.13925* (2024).

[110] Huangzhao Zhang, Zhiyi Fu, et al. 2022. Towards Robustness of Deep Program Processing Models – Detection, Estimation and Enhancement. *ACM Transactions on Software Engineering and Methodology* 31 (01 2022). doi:10.1145/3511887

[111] Kechi Zhang, Jia Li, et al. 2024. CodeAgent: Enhancing Code Generation with Tool-Integrated Agent Systems for Real-World Repo-level Coding Challenges. In *the 62nd Annual Meeting of the Association for Computational Linguistics*.

[112] Kechi Zhang, Zhuo Li, et al. 2023. Self-Edit: Fault-Aware Code Editor for Code Generation. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics ACL 2023*. Association for Computational Linguistics.

[113] Yuhao Zhang, Shiqi Wang, et al. 2024. CodeFort: Robust Training for Code Generation Models. In *Findings of the Association for Computational Linguistics: EMNLP*. doi:10.18653/V1/2024.FINDINGS-EMNLP.303

[114] Li Zhong and Zilong Wang. 2024. Can LLM Replace Stack Overflow? A Study on Robustness and Reliability of Large Language Model Code Generation. *Proceedings of the AAAI Conference on Artificial Intelligence* (03 2024).

[115] Li Zhong, Zilong Wang, and Jingbo Shang. 2024. Ldb: A large language model debugger via verifying runtime execution step-by-step. *arXiv:2402.16906* (2024).

[116] Qihuang Zhong, Kang Wang, et al. 2026. Achieving >97% on GSM8K: deeply understanding the problems makes LLMs better solvers for math word problems. *Frontiers Comput. Sci.* (2026).

[117] Zihao Zhou, Qiufeng Wang, et al. 2024. Mathattack: Attacking large language models towards math solving ability. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 38.

[118] Kaijie Zhu, Jindong Wang, et al. 2023. Promptrobust: Towards evaluating the robustness of large language models on adversarial prompts. In *the 1st ACM Workshop on Large AI Systems and Models with Privacy and Safety Analysis*.

[119] Tomasz G Zieli. 1992. Introduction to finite element method. (1992).