Towards Assessing and Repairing LLM-Generated Code via Model Checking and MaxSAT-Based Fault Localisation

Pedro Orvalho

MSCA Postdoctoral Fellow
Institut d'Investigació en Intelligència Artificial (**IIIA-CSIC**), Barcelona, Spain

Previously at:
Department of Computer Science, University of Oxford

Dagstuhl Seminar 26192: Evaluation of AI Models in Software Engineering
Dagstuhl, May 5, 2026

IIIA-CSIC INSTITUT D'INVESTIGACIÓ EN INTEL·LIGÈNCIA ARTIFICIAL

CSIC Consejo Superior de Investigaciones Científicas

Marie Skłodowska-Curie Actions Postdoctoral Fellowship

Outline

- Robustness of Code Models Agaisnt Code Mutations.

Outline

- Robustness of Code Models Agaisnt Code Mutations.
- Verifying LLM-based Code with Formal Methods.

Outline

- Robustness of Code Models Agaisnt Code Mutations.
- Verifying LLM-based Code with Formal Methods.
- On Verifying Python via LLM-Based Transpilation to C.

Motivation

- Large Language Models (LLMs) have rapidly become **integral to a wide range of programming and SE tasks**.

Motivation

- Large Language Models (LLMs) have rapidly become **integral to a wide range of programming and SE tasks**.

- LLMs **are widely used, and often blindly**, with developers placing significant trust in their capabilities [Oh et al., 2024].

Motivation

- Large Language Models (LLMs) have rapidly become **integral to a wide range of programming and SE tasks**.
- LLMs **are widely used, and often blindly**, with developers placing significant trust in their capabilities [Oh et al., 2024].
- However, this growing reliance on LLMs for coding tasks raises a fundamental question:

Motivation

- Large Language Models (LLMs) have rapidly become **integral to a wide range of programming and SE tasks**.
- LLMs **are widely used, and often blindly**, with developers placing significant trust in their capabilities [Oh et al., 2024].
- However, this growing reliance on LLMs for coding tasks raises a fundamental question:
  - To what extent do LLMs **truly understand code and the underlying semantics of programs**?

Motivation

- If LLMs outputs are simply the result of statistical associations, then **their reliability in critical development tasks could be overestimated** [Gu et al., 2024].

Motivation

- If LLMs outputs are simply the result of statistical associations, then **their reliability in critical development tasks could be overestimated** [Gu et al., 2024].

- In other domains, such as mathematical competitions [Petrov et al., 2025], LLMs tend to **provide accurate predictions, but based on flawed reasoning**.

Our work

- Conduct a **manual expert evaluation to assess whether LLMs' code output predictions** are based on logically sound reasoning, flawed reasoning, or mere guesses.

Pedro Orvalho, Marta Kwiatkowska. Are Large Language Models Robust in Understanding Code Against Semantics-Preserving Mutations? arXiv:2505.10443 (2025)

Our work

- Conduct a **manual expert evaluation to assess whether LLMs' code output predictions** are based on logically sound reasoning, flawed reasoning, or mere guesses.

- Evaluate LLMs' **output prediction stability** across five different semantics-preserving code mutations.

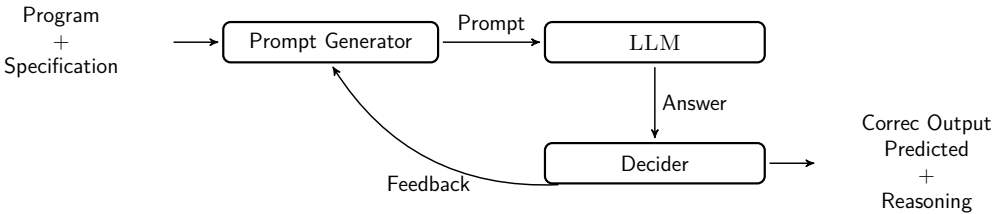
Pedro Orvalho, Marta Kwiatkowska. Are Large Language Models Robust in Understanding Code Against Semantics-Preserving Mutations? arXiv:2505.10443 (2025)

Semantics-Preserving Code Mutations

We introduce **five semantics-preserving code mutations** designed to syntactically modify Python programs without altering their semantics:

- variable renaming;
- comparison expression mirroring;
- swapping if-else statements;
- loop conversion
- partial loop unrolling.

LLM-Based Program Output Prediction



Experimental Setup

- **Evaluation Benchmarks:**
  - LiveCodeBench [Jain et al., 2024] **contains 479 programs** submitted to programming contests across competition platforms, such as LeetCode;
  - CruxEval [Gu et al., 2024] **contains 800 functions** generated by CodeLlama, each accompanied by a set of input-output examples for evaluation.

Experimental Setup

- **Evaluation Benchmarks:**
  - LiveCodeBench [Jain et al., 2024] **contains 479 programs** submitted to programming contests across competition platforms, such as `LeetCode`;
  - CruxEval [Gu et al., 2024] **contains 800 functions** generated by CodeLlama, each accompanied by a set of input-output examples for evaluation.
- For each program mutation, **we generate a separate transformed version of the benchmark**, each program containing at most one mutation;

Experimental Setup

- **Evaluation Benchmarks:**
  - LiveCodeBench [Jain et al., 2024] **contains 479 programs** submitted to programming contests across competition platforms, such as LeetCode;
  - CruxEval [Gu et al., 2024] **contains 800 functions** generated by CodeLlama, each accompanied by a set of input-output examples for evaluation.
- For each program mutation, **we generate a separate transformed version of the benchmark**, each program containing at most one mutation;
- We also **check that the semantics of the original program is preserved** in the mutated versions.

Experimental Setup

- **Large Language Models (LLMs):** We evaluated **eight different LLMs**.

Experimental Setup

- **Large Language Models (LLMs):** We evaluated **eight different LLMs**.
  - **Seven of these models are LLMCs**, i.e., LLMs fine-tuned for coding tasks:
    - IBM's Granite;
    - Google's CodeGemma;
    - Mistral's Mistral;
    - SemCoder;
    - Alibaba's Qwen3-Coder;
    - OpenAI's GPT-5.2;
    - Google's Gemini-3;

Experimental Setup

- **Large Language Models (LLMs):** We evaluated **eight different LLMs**.
  - **Seven of these models are LLMCs**, i.e., LLMs fine-tuned for coding tasks:
    - IBM's Granite;
    - Google's CodeGemma;
    - Mistral's Mistral;
    - SemCoder;
    - Alibaba's Qwen3-Coder;
    - OpenAI's GPT-5.2;
    - Google's Gemini-3;
  - **The other model is a general-purpose LLM**: Meta's Llama3;

Experimental Setup

- **Large Language Models (LLMs):** We evaluated **eight different LLMs**.
  - **Seven of these models are LLMCs**, i.e., LLMs fine-tuned for coding tasks:
    - IBM's GRANITE;
    - Google's CODEGEMMA;
    - Mistral's MISTRAL;
    - SEMCODER;
    - Alibaba's QWEN3-CODER;
    - OpenAI's GPT-5.2;
    - Google's GEMINI-3;
  - **The other model is a general-purpose LLM**: Meta's LLAMA3;
- Experiments were conducted using a timeout of **90s**.

Analysis of LLMs' Reasoning About Code

Large Language Models	CodeGemma	Granite	Qwen3-Coder	Mistral	SemCoder	Llama3	GPT-5.2	Gemini-3
%Failed Predictions	66.8	65.1	49.5	67.4	52.0	61.4	2.5	0.0
%Correct Predictions	33.2	34.9	50.5	32.6	48.0	38.6	97.5	100.0

Robustness to Semantics-Preserving Mutations

LiveCodeBench

LLMs	Original Code	Loop Conversion	Expression Mirroring	Variable Renaming	Swap If-Else	Loop Unrolling
CodeGemma	33.2%	26.3 (-6.9)	31.7 (-1.5)	38.0 (+4.8)	28.4 (-4.8)	10.9 (-22.3)
Granite	34.9%	27.1 (-7.7)	31.3 (-3.5)	38.8 (+4.0)	29.2 (-5.6)	8.4 (-26.5)
Llama3	38.6%	34.4 (-4.2)	34.2 (-4.4)	46.8 (+8.1)	30.1 (-8.6)	9.4 (-29.2)
Mistral	32.6%	24.0 (-8.6)	29.2 (-3.3)	38.2 (+5.6)	27.1 (-5.4)	10.4 (-22.1)
Qwen3-Coder	50.5%	37.8 (-12.7)	44.1 (-6.5)	54.7 (+4.2)	33.2 (-17.3)	15.4 (-35.1)
SemCoder	48.0%	40.1 (-7.9)	48.9 (+0.8)	62.4 (+14.4)	40.3 (-7.7)	15.2 (-32.8)
GPT-5.2	97.1%	**72.4 (-24.6)**	81.0 (-16.1)	98.5 (+1.5)	78.3 (-18.8)	29.0 (-68.1)
Gemini-3	100.0%	**55.3 (-44.7)**	76.0 (-24.0)	68.1 (-31.9)	71.4 (-28.6)	29.6 (-70.4)

Table 1: Output prediction correction rate of each LLM on LiveCodeBench when applying different code mutations: converting for to while loops (F2W), mirroring comparison expressions (MCE), renaming variables (RV), swap if-else statements (SIE), and unroll loops (UL).

Robustness to Semantics-Preserving Mutations

CruxEval

LLMs	Original Code	Loop Conversion	Expression Mirroring	Variable Renaming	Swap If-Else	Loop Unrolling
CodeGemma	30.9%	15.8 (-15.1)	17.6 (-13.3)	34.9 (+4.0)	20.1 (-10.8)	8.9 (-22.0)
Granite	32.6%	14.2 (-18.4)	17.1 (-15.5)	35.4 (+2.8)	19.8 (-12.9)	8.1 (-24.5)
Llama3	26.5%	15.0 (-11.5)	17.4 (-9.1)	37.5 (+11.0)	19.2 (-7.2)	6.6 (-19.9)
Mistral	23.8%	11.0 (-12.8)	13.4 (-10.4)	25.5 (+1.8)	13.9 (-9.9)	6.5 (-17.2)
Qwen3-Coder	48.9%	23.1 (-25.8)	26.2 (-22.6)	49.9 (+1.0)	26.2 (-22.6)	12.5 (-36.4)
SemCoder	50.6%	25.2 (-25.4)	29.9 (-20.8)	55.8 (+5.1)	30.4 (-20.2)	12.8 (-37.9)
GPT-5.2	86.0%	**42.2 (-43.8)**	47.9 (-38.1)	87.0 (+1.0)	52.9 (-33.1)	23.0 (-63.0)
Gemini-3	76.8%	**29.1 (-47.6)**	50.9 (-25.9)	66.2 (-10.5)	41.2 (-35.5)	24.4 (-52.4)

Table 2: Output prediction correction rate of each LLM on CruxEval when applying different code mutations: converting for to while loops (F2W), mirroring comparison expressions (MCE), renaming variables (RV), swap if-else statements (SIE), and unroll loops (UL).

Robustness to Semantics-Preserving Mutations

Are LLMs robust in understanding code against semantics-preserving mutations?

Robustness to Semantics-Preserving Mutations

Are LLMs robust in understanding code against semantics-preserving mutations?

- LLMs often change predictions in response to our code mutations, indicating **limited robustness in their semantic understanding**.

Verifying LLM-based Code with Formal Methods

P. Orvalho, M. Janota, and V. Manquinho. Counterexample Guided Program Repair Using Zero-Shot Learning and MaxSAT-based Fault localisation. AAAI 2025

Motivation

1: Semantically incorrect program. Faults: {4,8}.

```
1  int main(){ //finds max of 3 nums
2     int f,s,t;
3     scanf("%d%d%d",&f,&s,&t);
4     if (f < s && f >= t)
5        printf("%d",f);
6     else if (s > f && s >= t)
7        printf("%d",s);
8     else if (t < f && t < s)
9        printf("%d",t);
10
11    return 0;
12 }
```

- Symbolic approaches demand an **excessive amount of time to produce an answer**;

Motivation

2: Semantically incorrect program. Faults: {4,8}.

```
1  int main(){ //finds max of 3 nums
2     int f,s,t;
3     scanf("%d%d%d",&f,&s,&t);
4     if (f < s && f >= t)
5        printf("%d",f);
6     else if (s > f && s >= t)
7        printf("%d",s);
8     else if (t < f && t < s)
9        printf("%d",t);
10
11    return 0;
12 }
```

- Symbolic approaches demand an **excessive amount of time to produce an answer**;
- LLMs, while fast, **often produce incorrect or non-minimal fixes**.

Program Sketches

3: Semantically incorrect program. Faults :{4,8}.

```
1  int main(){ //finds max of 3 nums
2     int f,s,t;
3     scanf("%d%d%d",&f,&s,&t);
4     if (f < s && f >= t)
5        printf("%d",f);
6     else if (s > f && s >= t)
7        printf("%d",s);
8     else if (t < f && t < s)
9        printf("%d",t);
10
11    return 0;
12 }
```

4: Program sketch with holes.

```
1  int main(){
2     int f,s,t;
3     scanf("%d%d%d",&f,&s,&t);
4     @ HOLE 1 @
5        printf("%d",f);
6     else if (s > f && s >= t)
7        printf("%d",s);
8     @ HOLE 2 @
9        printf("%d",t);
10
11    return 0;
12 }
```

Our Work

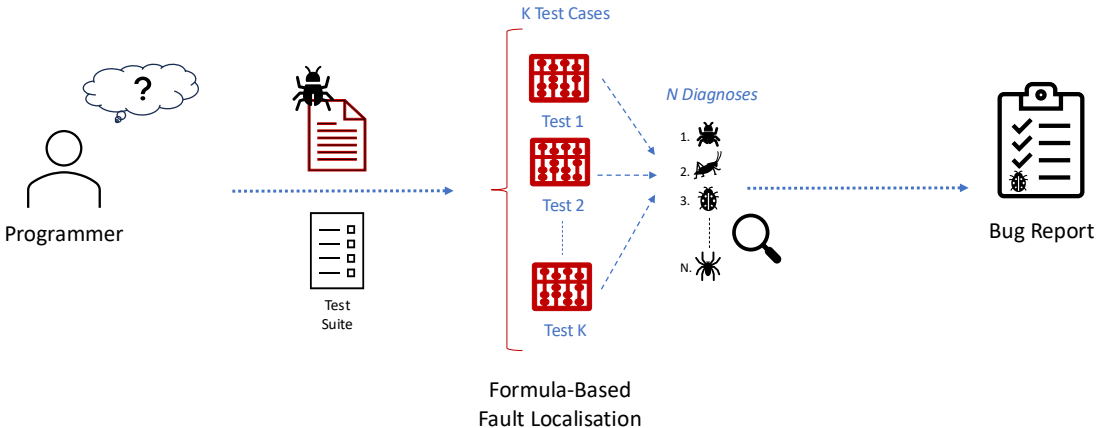
- Combines the strengths of **Formal Methods (FM) and LLM-based approaches**;

Our Work

- Combines the strengths of **Formal Methods (FM) and LLM-based approaches**;
- Uses **MaxSAT-based fault localisation to rigorously identify buggy lines**, which can then be highlighted in the LLM prompt;

Formula-Based Fault Localisation (FBFL)

- FBFL methods encode the localisation problem into **several optimisation problems** to identify a minimal set of bugs (diagnoses).



Formula-Based Fault Localisation (FBFL)

- We formulate the FL problem as a **single optimisation problem**;

P. Orvalho, M. Janota, and V. Manquinho (2024). CFaults: Model-Based Diagnosis for Fault localisation in C with Multiple Test Cases. *Formal Methods (FM) 2024*

Formula-Based Fault Localisation (FBFL)

- We formulate the FL problem as a **single optimisation problem**;
- We leverage MaxSAT and the theory of *Model-Based Diagnosis (MBD)* [Reiter et al., 1987], **integrating all failing test cases simultaneously**;

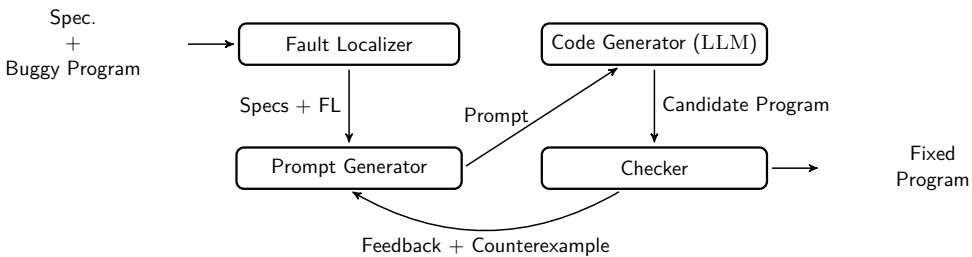
P. Orvalho, M. Janota, and V. Manquinho (2024). CFaults: Model-Based Diagnosis for Fault localisation in C with Multiple Test Cases. *Formal Methods (FM) 2024*

Formula-Based Fault Localisation (FBFL)

- We formulate the FL problem as a **single optimisation problem**;
- We leverage MaxSAT and the theory of *Model-Based Diagnosis (MBD)* [Reiter et al., 1987], **integrating all failing test cases simultaneously**;
- We implement this MBD approach in a publicly available tool called CFaults [Orvalho et al., 2024].

P. Orvalho, M. Janota, and V. Manquinho (2024). CFaults: Model-Based Diagnosis for Fault localisation in C with Multiple Test Cases. *Formal Methods (FM) 2024*

Counterexample Guided Automated Repair



Experimental Setup (2024)

- **Evaluation Benchmark:** C-Pack-IPAs, a set of twenty-five **introductory programming assignments**, comprising **1431 faulty programs**;

Experimental Setup (2024)

- **Evaluation Benchmark:** C-Pack-IPAs, a set of twenty-five **introductory programming assignments**, comprising **1431 faulty programs**;
- **Large Language Models (LLMs):** We evaluated **six different LLMs**.
  - **Three of these models are LLMCs**, i.e., LLMs fine-tuned for coding tasks:
    - IBM's Granite;
    - Google's CodeGemma;
    - Meta's CodeLlama.
  - **The other three models are general-purpose LLMs**:
    - Google's Gemma;
    - Meta's Llama3;
    - Microsoft's Phi3.

LLM-Driven APR with CFaults

LLMs	De-TS	De-TS-CE	FIXME_De-TS	FIXME_De-TS-CE	Sk_De-TS	Sk_De-TS-CE	Portfolio (All Configurations)
CodeGemma	597 (41.7%)	606 (42.3%)	592 (41.4%)	601 (42.0%)	682 (47.7%)	**688 (48.1%)**	823 (57.5%)
CodeLlama	492 (34.4%)	500 (34.9%)	481 (33.6%)	463 (32.4%)	**573 (40.0%)**	561 (39.2%)	712 (49.8%)
Gemma	496 (34.7%)	492 (34.4%)	446 (31.2%)	444 (31.0%)	532 (37.2%)	**534 (37.3%)**	670 (46.8%)
Granite	626 (43.7%)	624 (43.6%)	566 (39.6%)	583 (40.7%)	**691 (48.3%)**	681 (47.6%)	846 (59.1%)
Llama3	564 (39.4%)	590 (41.2%)	535 (37.4%)	557 (38.9%)	578 (40.4%)	**591 (41.3%)**	851 (59.5%)
Phi3	494 (34.5%)	489 (34.2%)	460 (32.1%)	474 (33.1%)	**547 (38.2%)**	535 (37.4%)	621 (43.4%)
Portfolio (All LLMs)	842 (58.8%)	846 (59.1%)	796 (55.6%)	820 (57.3%)	900 (62.9%)	**907 (63.4%)**	1013 (70.8%)

Table 3: The number of programs fixed by each LLM under various configurations. Mapping abbreviations to configuration names: **De** - IPA *Description*, **TS** - *Test Suite*, **CE** - *Counterexample*, **FIXME** - *FIXME Comments*, **SK** - *Sketches*.

LLM-Driven APR with CFaults

LLMs	De-TS	De-TS-CE	FIXME_De-TS	FIXME_De-TS-CE	Sk_De-TS	Sk_De-TS-CE	Portfolio (All Configurations)
CodeGemma	597 (41.7%)	606 (42.3%)	592 (41.4%)	601 (42.0%)	682 (47.7%)	**688 (48.1%)**	823 (57.5%)
CodeLlama	492 (34.4%)	500 (34.9%)	481 (33.6%)	463 (32.4%)	**573 (40.0%)**	561 (39.2%)	712 (49.8%)
Gemma	496 (34.7%)	492 (34.4%)	446 (31.2%)	444 (31.0%)	532 (37.2%)	**534 (37.3%)**	670 (46.8%)
Granite	626 (43.7%)	624 (43.6%)	566 (39.6%)	583 (40.7%)	**691 (48.3%)**	681 (47.6%)	846 (59.1%)
Llama3	564 (39.4%)	590 (41.2%)	535 (37.4%)	557 (38.9%)	578 (40.4%)	**591 (41.3%)**	851 (59.5%)
Phi3	494 (34.5%)	489 (34.2%)	460 (32.1%)	474 (33.1%)	**547 (38.2%)**	535 (37.4%)	621 (43.4%)
Portfolio (All LLMs)	842 (58.8%)	846 (59.1%)	796 (55.6%)	820 (57.3%)	900 (62.9%)	**907 (63.4%)**	1013 (70.8%)

Table 3: The number of programs fixed by each LLM under various configurations. Mapping abbreviations to configuration names: **De** - IPA *Description*, **TS** - *Test Suite*, **CE** - *Counterexample*, **FIXME** - *FIXME Comments*, **SK** - *Sketches*.

Incorporating **MaxSAT-based Sketches allows LLMs to repair more programs**

On Verifying Python via LLM-Based Transpilation and Bounded Model Checking for C

Pedro Orvalho, Marta Kwiatkowska. On Verifying Python via LLM-Based Transpilation and Bounded Model Checking for C. arXiv:2508.08171 (2025)

Motivation

- Python has become the **dominant language** for general-purpose programming.

Motivation

- Python has become the **dominant language** for general-purpose programming.
- Yet it **lacks robust tools for formal verification**.

Motivation

- Python has become the **dominant language** for general-purpose programming.
- Yet it **lacks robust tools for formal verification**.
- **Languages such as C benefit from mature model checkers**, for example CBMC [Clarke et al., 2004], which enable exhaustive symbolic reasoning and FL.

Motivation

- Python has become the **dominant language** for general-purpose programming.
- Yet it **lacks robust tools for formal verification**.
- **Languages such as C benefit from mature model checkers**, for example CBMC [Clarke et al., 2004], which enable exhaustive symbolic reasoning and FL.
- The complexity of Python, coupled with the verbosity of existing transpilers (e.g., Cython), have historically limited the applicability of formal verification to Python.

Motivation

```
1  def distributeCandies(n: int, limit: int) -> int:
2      limit = min(limit, n)
3      ans = 0
4      ans = 0 + 1
5      for i in range(limit + 1):
6          if n - i > limit * 2:
7              continue
8          ans += min(limit, n-i)-max(0, n-i-limit)+1
9      return ans
10 assert distributeCandies(n = 5, limit = 2) == 3
```

Motivation

```
1  def distributeCandies(n: int, limit: int) -> int:
2      limit = min(limit, n)
3      ans = 0
4      ans = 0 + 1
5      for i in range(limit + 1):
6          if n - i > limit * 2:
7              continue
8          ans += min(limit, n-i)-max(0, n-i-limit)+1
9      return ans
10 assert distributeCandies(n = 5, limit = 2) == 3
```

- Line 4 is an accidental duplicate that introduces a subtle off-by-one bug.

Motivation

```
1  def distributeCandies(n: int, limit: int) -> int:
2      limit = min(limit, n)
3      ans = 0
4      ans = 0 + 1
5      for i in range(limit + 1):
6          if n - i > limit * 2:
7              continue
8          ans += min(limit, n-i)-max(0, n-i-limit)+1
9      return ans
10 assert distributeCandies(n = 5, limit = 2) == 3
```

- Line 4 is an accidental duplicate that introduces a subtle off-by-one bug.
- Existing model checkers (e.g., ESBMC-Python [Farias et al., 2024]) **support only small subsets** of Python.

Motivation

```
1  def distributeCandies(n: int, limit: int) -> int:
2      limit = min(limit, n)
3      ans = 0
4      ans = 0 + 1
5      for i in range(limit + 1):
6          if n - i > limit * 2:
7              continue
8          ans += min(limit, n-i)-max(0, n-i-limit)+1
9      return ans
10 assert distributeCandies(n = 5, limit = 2) == 3
```

- Line 4 is an accidental duplicate that introduces a subtle off-by-one bug.
- Existing model checkers (e.g., ESBMC-Python [Farias et al., 2024]) **support only small subsets** of Python.
- For instance, ESBMC-Python **cannot verify this Python program**.

PyVeritas

A novel tool for **verification and fault localisation for Python programs** by leveraging Large Language Model (LLM)-based transpilation to C.

PyVeritas

A novel tool for **verification and fault localisation for Python programs** by leveraging Large Language Model (LLM)-based transpilation to C.

1. **Transpiles Python, using an LLM**, to a semantics-preserving C version.

PyVeritas

A novel tool for **verification and fault localisation for Python programs** by leveraging Large Language Model (LLM)-based transpilation to C.

1. **Transpiles Python, using an LLM**, to a semantics-preserving C version.
2. Runs C model checkers (e.g., CBMC) to **check assertions and produce counterexamples through bounded model checking**.

PyVeritas

A novel tool for **verification and fault localisation for Python programs** by leveraging Large Language Model (LLM)-based transpilation to C.

1. **Transpiles Python, using an LLM**, to a semantics-preserving C version.
2. Runs C model checkers (e.g., CBMC) to **check assertions and produce counterexamples through bounded model checking**.
3. Runs **MaxSAT-based fault localisation** tools (e.g., CFaults [Orvalho et al., 2024]) to find the bugs in the C version.

PyVeritas

A novel tool for **verification and fault localisation for Python programs** by leveraging Large Language Model (LLM)-based transpilation to C.

1. **Transpiles Python, using an LLM**, to a semantics-preserving C version.
2. Runs C model checkers (e.g., CBMC) to **check assertions and produce counterexamples through bounded model checking**.
3. Runs **MaxSAT-based fault localisation** tools (e.g., CFaults [Orvalho et al., 2024]) to find the bugs in the C version.
4. Maps the buggy statements back to the Python code, providing **useful feedback**.

PyVeritas

A novel tool for **verification and fault localisation for Python programs** by leveraging Large Language Model (LLM)-based transpilation to C.

1. **Transpiles Python, using an LLM**, to a semantics-preserving C version.
2. Runs C model checkers (e.g., CBMC) to **check assertions and produce counterexamples through bounded model checking**.
3. Runs **MaxSAT-based fault localisation** tools (e.g., CFaults [Orvalho et al., 2024]) to find the bugs in the C version.
4. Maps the buggy statements back to the Python code, providing **useful feedback**.

Note: This is an **interim approach** until a robust, native Python symbolic model checker is available.

PyVeritas

A novel tool for **verification and fault localisation for Python programs** by leveraging Large Language Model (LLM)-based transpilation to C.

1. **Transpiles Python, using an LLM**, to a semantics-preserving C version.
2. Runs C model checkers (e.g., CBMC) to **check assertions and produce counterexamples through bounded model checking**.
3. Runs **MaxSAT-based fault localisation** tools (e.g., CFaults [Orvalho et al., 2024]) to find the bugs in the C version.
4. Maps the buggy statements back to the Python code, providing **useful feedback**.

Note: This is an **interim approach** until a robust, native Python symbolic model checker is available. Because PyVeritas **relies on LLMs** for transpilation and for mapping localised statements back to Python, **no completeness guarantees** are provided.

Running Example

- PyVeritas asks an LLM to transpiled this buggy Python program

```
1  def distributeCandies(n: int, limit: int) -> int:
2      limit = min(limit, n)
3      ans = 0
4      ans = 0 + 1
5      for i in range(limit + 1):
6          if n - i > limit * 2:
7              continue
8          ans += min(limit, n-i)-max(0, n-i-limit)+1
9      return ans
10 assert distributeCandies(n = 5, limit = 2) == 3
```

Running Example

- PyVeritas runs CBMC to check that the program is failing the assertion.

```
1  int distributeCandies(int n, int limit) {
2      limit = (limit < n) ? limit : n;
3      int ans = 0;
4      ans = 0 + 1;
5      for (int i = 0; i <= limit; i++) {
6          if (n - i > limit * 2) {
7              continue;
8          }
9          ans += ((limit < n-i) ? limit : (n-i)) -
10         ((n-i-limit > 0) ?(n-i-limit) : 0) + 1;
11     }
12     return ans;
13 }
14 int main() {
15     assert(distributeCandies(5, 2) == 3);
16     return 0;
17 }
```

Running Example

- PyVeritas uses CFaults to localise the buggy assignment in the C code.

```
1  int distributeCandies(int n, int limit) {
2      limit = (limit < n) ? limit : n;
3      int ans = 0;
4      ans = 0 + 1;
5      for (int i = 0; i <= limit; i++) {
6          if (n - i > limit * 2) {
7              continue;
8          }
9          ans += ((limit < n-i) ? limit : (n-i)) -
10         ((n-i-limit > 0) ?(n-i-limit) : 0) + 1;
11     }
12     return ans;
13 }
14 int main() {
15     assert(distributeCandies(5, 2) == 3);
16     return 0;
17 }
```

Running Example

- PyVeritas maps the localised buggy statement back to the original Python source code by leveraging the same LLM used during transpilation.

```python
    ans = 0 + 1
```

Running Example

- PyVeritas maps the localised buggy statement back to the original Python source code by leveraging the same LLM used during transpilation.

```python
    ans = 0 + 1
```

Thus, PyVeritas provides precise localisation for simple bugs in Python.

LLM-based Transpilation of Correct Code

Language Model	LiveCodeBench	Refactory
GPT-5.2	95.2%	97.6%
Qwen3-Coder (30B)	86.6%	95.2%
DeepSeek-Coder-V2 (16B)	65.1%	64.8%
Granite (8B)	55.9%	52.0%
Llama3 (3B)	43.0%	28.0%

Table 4: Verification success rates for each LLM on both benchmarks. Percentages indicate the proportion of C programs that were successfully verified by CBMC and judged semantically equivalent to the original Python code.

MaxSAT-Based Fault Localisation

	Bug: **Wrong Binary Operator (WBO)**			
LLMs	**% Correct Bug Found**	**% Other Bugs**	**% Transpiled Fixed Code**	**% Compilation Err**
GPT-5.2	**77.3%**	14.4%	8.3%	0.0%
Qwen3-Coder (30B)	33.8%	24.4%	41.8%	0.0%
DeepSeek-Coder-V2 (16B)	16.1%	36.8%	42.9%	4.2%
Granite (8B)	41.6%	34.6%	15.2%	8.6%
Llama3 (3B)	22.7%	47.9%	17.7%	11.6%
	Bug: **Assignment Duplication with Constant (ADC)**			
	% Correct Bug Found	**% Other Bugs**	**% Transpiled Fixed Code**	**% Compilation Err**
GPT-5.2	**81.9%**	9.5%	8.6%	0.0%
Qwen3-Coder (30B)	25.7%	6.2%	**68.1%**	0.0%
DeepSeek-Coder-V2 (16B)	6.7%	21.4%	**69.5%**	2.4%
Granite (8B)	39.5%	11.9%	36.7%	11.9%
Llama3 (3B)	18.6%	39.0%	34.3%	8.1%

Table 5: Results of MaxSAT-Based FL with PyVeritas on LiveCodeBench using WBO and ADC bugs.

Takeaway Message

- LLMs often change predictions in response to our code mutations, indicating **limited robustness in their semantic understanding**.

Takeaway Message

- LLMs often change predictions in response to our code mutations, indicating **limited robustness in their semantic understanding**.
- Incorporating **MaxSAT-based FL Sketches allows LLMs to repair more programs**.

Takeaway Message

- LLMs often change predictions in response to our code mutations, indicating **limited robustness in their semantic understanding**.
- Incorporating **MaxSAT-based FL Sketches allows LLMs to repair more programs**.
- Our novel approach for **verification and fault localisation for Python programs** by leveraging LLM-based transpilation to C.

Takeaway Message

- LLMs often change predictions in response to our code mutations, indicating **limited robustness in their semantic understanding**.
- Incorporating **MaxSAT-based FL Sketches allows LLMs to repair more programs**.
- Our novel approach for **verification and fault localisation for Python programs** by leveraging LLM-based transpilation to C.
- We combine **LLM-based code transpilation, bounded model checking** with CBMC, and **MaxSAT-based fault localisation** using CFaults.

Takeaway Message

- LLMs often change predictions in response to our code mutations, indicating **limited robustness in their semantic understanding**.
- Incorporating **MaxSAT-based FL Sketches allows LLMs to repair more programs**.
- Our novel approach for **verification and fault localisation for Python programs** by leveraging LLM-based transpilation to C.
- We combine **LLM-based code transpilation, bounded model checking** with CBMC, and **MaxSAT-based fault localisation** using CFaults.
- We are able to accurately **verify small yet non-trivial Python programs where native verification tools fall short**.

Thank you!

https://pmorvalho.github.io

This work was supported by grant PID2022-139835NB-C21 funded by MCIN/AEI/10.13039/501100011033 and by ERDF, EU; and by HORIZON-MSCA-2025-PF, project 101269051 — Sherlock4Py funded by REA, EU.

References

Oh, Sanghak and Lee, Kiho and Park, Seonhye and Kim, Doowon and Kim, Hyoungshick (2024)

Poisoned ChatGPT Finds Work for Idle Hands: Exploring Developers' Coding Practices with Insecure Suggestions from Poisoned AI Model.

IEEE Symposium on Security and Privacy, *SP 2024*.

Ivo Petrov and Jasper Dekoninck and Lyuben Baltadzhiev and Maria Drencheva and Kristian Minchev and Mislav Balunovic and Nikola Jovanovic and Martin T. Vechev (2025)

Proof or Bluff? Evaluating LLMs on 2025 USA Math Olympiad.

CoRR 2025.

Alex Gu and Wen-Ding Li and Naman Jain and Theo Olausson and Celine Lee and Koushik Sen and Armando Solar-Lezama (2024)

The Counterfeit Conundrum: Can Code Language Models Grasp the Nuances of Their Incorrect Generations?

ACL 2024.

References

Naman Jain and King Han and Alex Gu and Wen-Ding Li and Fanjia Yan and Tianjun Zhang and Sida Wang and Armando Solar-Lezama and Koushik Sen and Ion Stoica (2024)

LiveCodeBench: Holistic and Contamination Free Evaluation of Large Language Models for Code.

CoRR 2024.

Alex Gu and Baptiste Rozière and Hugh Leather and Armando Solar-Lezama and Gabriel Synnaeve and Sida I. Wang (2024)

CRUXEval: A Benchmark for Code Reasoning, Understanding and Execution.

CoRR 2024.

CodeLlama Team (2023).

Code Llama: Open Foundation Models for Code.

CoRR 2023.

References

Armando Solar-Lezama and Liviu Tancau and Rastislav Bodík and Sanjit A. Seshia and Vijay A. Saraswat (2018)

Combinatorial sketching for finite programs.

ASPLOS 2006.

Reiter, Raymond (1987)

A Theory of Diagnosis from First Principles.

Artif. Intell. 1987.

P. Orvalho and M. Janota and V. Manquinho (2024)

C-Pack of IPAs: A C90 Program Benchmark of Introductory Programming Assignments.

Automated Program Repair (APR) 2024.

P. Orvalho and M. Janota and V. Manquinho (2025)

Counterexample Guided Program Repair Using Zero-Shot Learning and MaxSAT-based Fault localisation.

AAAI 2025.

References


Edmund M. Clarke and Daniel Kroening and Flavio Lerda (2004)

A Tool for Checking ANSI-C Programs

TACAS 2004.


Bruno Farias and Rafael Menezes and Eddie B. de Lima Filho and Youcheng Sun and Lucas C. Cordeiro (2024)

ESBMC-Python: A Bounded Model Checker for Python Programs.

ISSTA 2024.


P. Orvalho and M. Janota and V. Manquinho (2024)

CFaults: Model-Based Diagnosis for Fault localisation in C with Multiple Test Cases.

Formal Methods (FM) 2024.