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

Pedro Orvalho, Marta Kwiatkowska

Department of Computer Science
University of Oxford
Oxford, UK

Abstract

Python has become the dominant language for general-purpose programming, yet it lacks robust tools for formal verification. In contrast, programmers working in languages such as C benefit from mature model checkers, for example CBMC, which enable exhaustive symbolic reasoning and fault localisation. The inherent complexity of Python, coupled with the verbosity and low-level nature of existing transpilers (e.g., Cython), have historically limited the applicability of formal verification to Python programs.

In this paper, we propose PyVeritas, a novel framework that leverages Large Language Models (LLMs) for high-level transpilation from Python to C, followed by bounded model checking and MaxSAT-based fault localisation in the generated C code. PyVeritas enables verification and bug localisation for Python code using existing model checking tools for C. Our empirical evaluation on two Python benchmarks demonstrates that LLM-based transpilation can achieve a high degree of accuracy, up to 80–90% for some LLMs, enabling effective development environment that supports assertion-based verification and interpretable fault diagnosis for small yet non-trivial Python programs.

1 Introduction

Automated software verification through techniques such as model checking (Clarke 1997; Clarke, Kroening, and Yorav 2003; Cimatti and Griggio 2012) has long been a central topic in programming languages and software engineering research, with widespread industrial adoption in safety-critical systems (Lahtinen et al. 2012; Hartonas-Garmhausen et al. 2000), hardware verification (Clarke and Kroening 2003), and web-services (Chong et al. 2020; Huang et al. 2005). In particular, bounded model checking (BMC) (Biere et al. 2003) has proven to be a highly effective method for exhaustively analysing program behaviours, within fixed bounds on variables and computation steps. Mature tools such as CBMC (Clarke, Kroening, and Lerda 2004) have demonstrated remarkable success in verifying and analysing C programs through symbolic reasoning and constraint solving.

Despite advances, a significant gap remains: there is still no robust, scalable model checker for Python, which is in widespread use today. Recently, ESBMC-Python (Farias et al. 2024) model checker was proposed; however, it remains at the prototype stage, limited to verifying simple functions and supporting only a restricted subset of the language. Although lightweight static analysis and type-checking tools exist, formal verification of general-purpose Python programs remains largely under-developed. This limitation arises from Python's highly dynamic and expressive nature, which make it particularly challenging to encode Python semantics within a formal verification framework. An alternative route via code transpilation, that is, translating source code from one programming language to another, is impractical for Python. Existing transpilers, such as Cython (Cython 2025) and Shedskin (Shedskin 2025), generate thousands of lines of low-level C code even for modest Python functions, preventing effective symbolic analysis.

Recently, *Large Language Models* (LLMs) trained for code (LLMCs) have demonstrated strong performance across a range of code-related tasks, including program synthesis (Li, Parsert, and Polgreen 2024), program repair (Joshi et al. 2023; Zhang et al. 2024; Orvalho, Janota, and Manquinho 2025), and code transpilation (Eniser, Wüstholz, and Christakis 2024; Zhu et al. 2024; Ramos et al. 2024). LLMCs trained on large-scale multilingual code corpora have shown the ability to produce semantically faithful translations between languages such as Python and C, often achieving more concise and high-level translations than traditional transpilers (Eniser, Wüstholz, and Christakis 2024).

In this paper, we propose a novel approach that leverages LLM-based transpilation from Python to C, followed by automated verification of the generated C code using bounded model checking (via CBMC). As concrete applications of this approach, we demonstrate how, based on the transpiled C code, one can: (1) verify the correctness of Python programs using CBMC, and (2) localise bugs in Python code by applying formula-based fault localisation techniques to the C translation using CFaults (Orvalho, Janota, and Manquinho 2024), a MaxSAT-based fault localisation tool for C programs. We implement this pipeline in a tool called PyVeritas. This tool offers an interim solution to the problem of Python verification by integrating LLM-based transpilation with mature C verification tools, bridging the gap until native Python model checking becomes feasible.

Our experiments focus on LLMs with up to 32 bil-

Algorithm 1: Python function `distributeCandies(n: int, limit: int)`.

```python
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
11  assert distributeCandies(n = 5, limit = 2) == 3
```

Algorithm 2: C version of Algorithm 1 transpiled by QWEN-2.5-CODER from Python to C.

```c
int distributeCandies(int n, int limit) {
    limit = (limit < n) ? limit : n;
    int ans = 0;
    ans = 0 + 1;
    for (int i = 0; i <= limit; i++) {
        if (n - i > limit * 2) {
            continue;
        }
        ans += ((limit < n-i) ? limit : (n-i)) -
        ((n-i-limit > 0) ?(n-i-limit) : 0) + 1;
    }
    return ans;
}

int main() {
    assert(distributeCandies(5, 2) == 3);
    return 0;
}
```

lion parameters on two well-studied Python benchmarks: LIVECODEBENCH (Jain et al. 2024) and REFACTORY (Hu et al. 2019). We evaluate four different models: three trained specifically for coding tasks, QWEN2.5-CODER, DEEPSEEK-CODER-V2 and GRANITECODE; and one general-purpose model, LLAMA3.2, for comparison. Our experiments show that LLM-based transpilation can produce semantically faithful C code for Python programs, enabling formally verified Python code in over 80% of cases with some models, notably QWEN2.5-CODER. Moreover, by applying MaxSAT-based fault localisation to the transpiled code, PYVERITAS can effectively identify faults injected into the Python source code.

In summary, this paper makes the following contributions.

- We propose PYVERITAS, a novel framework for verification and fault localisation for Python programs by leveraging LLM-based transpilation to C.
- PYVERITAS combines LLM-based code transpilation, bounded model checking with CBMC, and MaxSAT-based fault localisation using CFAULTS.
- We conduct an empirical evaluation on two Python benchmarks, showing that PYVERITAS accurately verifies small yet non-trivial Python programs where native verification tools fall short.
- PYVERITAS can map localised faults in transpiled C code back to the original Python source, providing interpretable diagnostic feedback.

2 Motivation

Consider the function `distributeCandies(n: int, limit: int)` presented in Algorithm 1, which computes the total number of ways to distribute `n` candies among three children such that no child receives more than `limit` candies. This Python function corresponds to program #463 from LIVECODEBENCH and was obtained from LEETCODE (LeetCode 2025), although line 4, `ans = 0 + 1`, was introduced as an intended fault by duplicating the assignment on line 3 and adding a constant. Now, suppose we want to verify that this program satisfies the assertion on line 11.

Attempting to verify Algorithm 1 using ESBMC-PYTHON (Farias et al. 2024), version 7.9, a recently proposed model checker for Python, proves infeasible. This prototype can only handle a small subset of Python and its functionality is currently limited, supporting only top-level `for` loops (i.e., loops cannot be inside functions), restricting lists and strings, and lacking support for dictionaries. Consequently, it is currently not able to analyse the structures in this program. Alternatively, one might consider using a robust Python-to-C transpiler such as CYTHON (Cython 2025). However, this approach is also impractical. CYTHON translates this 9-line Python function into over 6,100 lines of low-level and complex C code. Analysing and verifying such code would be extremely challenging, and tracing fault information back to the original Python source would be nearly impossible, making it of little help to programmers.

This is where we believe PYVERITAS can offer a practical solution, particularly given that existing Python model checkers are still in early development and do not yet support the full expressiveness of Python. Our approach is to use *Large Language Models* (LLMS) to transpile a Python program to C, whether correct or buggy, guided by a natural language description and a formal specification of the intended behaviour. Once in C, we can apply several mature verification and symbolic analysis tools developed for C to support developers by providing verification and diagnostic feedback at the level of Python.

For instance, if we provide Algorithm 1 along with its natural language description to QWEN2.5-CODER (Qwen 2024), an LLM fine-tuned for coding tasks, and request a C translation using our transpilation prompt (see Appendix A), the model produces the C code shown in Algorithm 2. We can then apply CBMC, a bounded model checker for C, and verify that the assertion fails, confirming that the program is indeed incorrect. Furthermore, given a C version of the original Python program, we can invoke MaxSAT-based fault localisation tools, such as CFAULTS (Orvalho, Janota, and

Manquinho 2024), to identify the root cause of the failure. In this case, CFAULTS reports that line 4 of the C program in Algorithm 2 is faulty. We can then provide this information to the LLM and ask it to map the identified line back to the corresponding faulty statement in the original Python source. QWEN2.5-CODER's response is as follows:

````
Here are the corresponding buggy Python
statements from the original program that
correspond to the buggy C statements
provided:
```python
    ans = 0 + 1
```
````

Thus, we successfully localised line 4 as the fault in the original Python program in Algorithm 1, through LLM-based transpilation, model checking, and MaxSAT-based fault localisation for C. To the best of our knowledge, MaxSAT-based fault localisation techniques have, until now, only been developed for C (Jose and Majumdar 2011; Orvalho, Janota, and Manquinho 2024), with no equivalent methodology yet proposed for Python. Therefore, at present, this remains the only viable way to leverage these powerful localisation approaches, which are capable of precisely identifying the minimal set of faulty statements in a program.

3 Preliminaries

This section defines key concepts used throughout the paper.

Satisfiability. The *Boolean Satisfiability (SAT)* problem is the decision problem for propositional logic (Biere et al. 2009). The *Maximum Satisfiability (MaxSAT)* problem is an optimization version of the SAT problem. Given a CNF formula ϕ, the goal is to find an assignment that satisfies the maximum number of clauses (Bacchus, Järvisalo, and Martins 2021). In partial MaxSAT, ϕ is divided into hard clauses (ϕ_h) and soft clauses (ϕ_s). The goal is to satisfy all clauses in ϕ_h while minimising the number of unsatisfied soft clauses.

Programs and Verification. A *program* is sequential, comprising standard statements such as assignments, conditionals, loops, and function calls, each following conventional Python/C semantics. A program is considered *buggy* if an assertion violation occurs during execution with input I. Otherwise, it is correct for I. When a bug is observed, the corresponding *error trace* is the sequence of statements executed by program P on input I. A *trace formula (TF)* is a SAT formula that is satisfiable if and only if there exists an execution of the program that terminates in an *assertion violation* while respecting all assume statements (Clarke, Kroening, and Yorav 2003).

Formula-Based Fault Localisation. Given a faulty program and a test suite with failing cases, *formula-based fault localisation* (FBFL) methods encode the localisation task into an optimisation problem, aiming to identify a minimal set of faulty statements (diagnoses). Typically, FBFL tools encode a program's trace formula as a MaxSAT formula, and apply the theory of *model-based diagnosis* (MBD) (Reiter 1987; Marques-Silva et al. 2015; Ignatiev et al. 2019; Orvalho, Janota, and Manquinho 2024) to enumerate all diagnoses, each corresponding to a potential bug location. Following this theory, the clauses encoding the trace formula of the buggy program correspond to the *hard clauses* of the MaxSAT formula, while the set of healthy (or relaxation) variables correspond to the *soft clauses*, as the goal is to maximise the number of healthy program statements.

4 PYVERITAS

In this section, we introduce PYVERITAS, a novel framework that enables formal verification and MaxSAT-based fault localisation of Python programs through LLM-based transpilation to C and model checking for C. Figure 1 presents an overview of PYVERITAS.

While symbolic tools, for example, ESBMC-PYTHON (Farias et al. 2024), have attempted to support symbolic reasoning over Python, these tools remain in early stages and are limited to simple functions and subsets of the language. On the other hand, bounded model checking (BMC) tools for C, such as CBMC (Clarke, Kroening, and Lerda 2004), are mature, efficient, and support fault localisation via MaxSAT-based analysis (Orvalho, Janota, and Manquinho 2024; Jose and Majumdar 2011). The goal of PYVERITAS is to bridge the gap in formal verification for Python while a robust native model checker for the language is still lacking. PYVERITAS operates by translating Python to C programs using Large Language Models (LLMS), and then leveraging mature C verification infrastructure to perform symbolic analysis and fault localisation. PYVERITAS implements a verification pipeline that, given a Python program and its specification, returns either a formal correctness/failure verdict or a set of localised buggy statements mapped back to the original Python source. Please refer to Appendix A for all the prompt templates used in the various interactions with the LLM.

Architecture

PYVERITAS takes as input a Python program $\mathcal{P}$, a textual description $\mathcal{D}$ (used to help preserve intent during transpilation), and a set of specifications $\mathcal{S}$, encoded as assertions. Textual descriptions are typically provided to LLMS to indicate intent, whereas assertions are necessary for formal verification. We remark that this provides redundant information, which may contain inconsistencies, particularly when the code to transpile is faulty.

LLM-Based Transpilation. Given $(\mathcal{P}, \mathcal{D}, \mathcal{S})$, PYVERITAS constructs a prompt and queries an LLM to generate a candidate C program $\mathcal{C}$ that aims to preserve the semantics of $\mathcal{P}$ and satisfy the specifications in $\mathcal{S}$.

C Interpreter. Since LLM-generated code can be imprecise or syntactically/semantically incorrect, PYVERITAS executes both $\mathcal{P}$ and $\mathcal{C}$ on the same test inputs and assertions (i.e., $\mathcal{S}$) using a Python interpreter and a C interpreter, respectively. If $\mathcal{C}$ fails to satisfy the assertions in $\mathcal{S}$ that $\mathcal{P}$ satisfies, or if $\mathcal{C}$ fails to compile, the candidate is discarded and a new one is requested from the LLM, up to five failed attempts, indicating the model is likely repeating the same incorrect output; or until the time limit is reached.

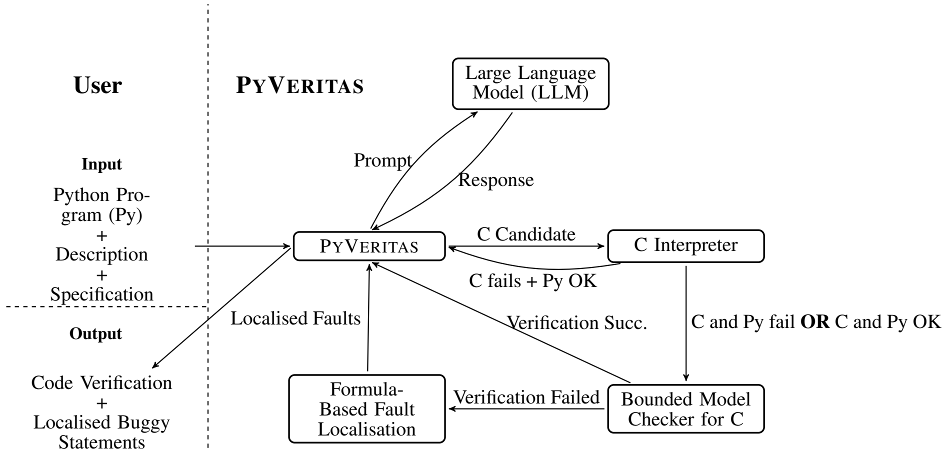

Figure 1: PyVeritas's pipeline for Python verification and bug localisation.

Verification via CBMC. If the C candidate passes the interpreter, PyVeritas verifies it using a bounded model checker for C, CBMC (Clarke, Kroening, and Lerda 2004). If verification succeeds, PyVeritas concludes that the original Python program satisfies the specification $\mathcal{S}$.

MaxSAT-Based Fault Localisation. If verification fails, the candidate $\mathcal{C}$ is analysed using CFaults (Orvalho, Janota, and Manquinho 2024), a formula-based fault localisation tool for C. CFaults encodes the trace formula of $\mathcal{C}$ as a MaxSAT formula and computes a solution that identifies the minimum set of faulty statements in the C program.

Back-Mapping to Python. PyVeritas maps the identified fault locations in $\mathcal{C}$ back to their corresponding statements in the original Python program $\mathcal{P}$. This mapping is performed by querying the LLM with the original code and the localised fault locations. The resulting output is the faulty statements localised in the original Python code.

Output. The output of PyVeritas is twofold: (1) if the C code passes verification with CBMC, the Python program is declared *verified* with respect to $\mathcal{S}$; and (2) if verification fails, the user is presented with a list of localised faulty statements in the original Python source, inferred via MaxSAT-based analysis on the transpiled C code.

5 Experiments

The goal of our experiments was to answer the following research questions (RQs). **RQ1.** How accurate and reliable are Large Language Models (LLMs) at transpiling Python programs into semantically equivalent C code? **RQ2.** Can MaxSAT-based fault localisation in C, when applied to LLM-transpiled Python code, effectively identify faults injected into Python source? **RQ3.** How dependable is LLM-based transpilation as a method for supporting MaxSAT-based fault localisation in Python via C-level analysis tools?

Experimental Setup. All experiments were run using a NVIDIA L40S graphics card with 48GB of memory on an Intel(R) Xeon(R) Gold 5418Y 48-Core CPU Processor and 251GB RAM, using a time limit of 10 minutes.

Evaluation Benchmarks. To evaluate PyVeritas, we use two widely adopted benchmarks of Python programs: LiveCodeBench (Jain et al. 2024) and Refactory (Hu et al. 2019). LiveCodeBench contains 479 correct programs submitted to programming contests across competition platforms, such as `LeetCode` (LeetCode 2025). Refactory is a benchmark consisting of 1783 buggy and 2447 correct Python programs submitted to five different programming assignments. For our evaluation, we focus on correct programs to support our experiment: transpiling the original Python programs to C and formally verifying their correctness using CBMC. Specifically, we randomly selected twenty-five correct programs from each assignment, resulting in a total of 125 Python programs.

To assess the effectiveness of localising bugs in Python programs using MaxSAT-based fault localisation for C through CFaults (Orvalho, Janota, and Manquinho 2024), we injected faults into these Python benchmarks. To obtain precise ground-truth information about the location and nature of the bugs in each program, we applied *semantics-altering mutations*, i.e., syntactically valid code transformations that deliberately change program behaviour. Specifically, we used two mutations that are easy to inject automatically: (1) *Wrong binary operator (WBO)*, where a comparison operator (e.g., ==, <, >) is replaced with a different one, thereby flipping the logic of a conditional expression; and (2) *Assignment duplication with constant (ADC)*, in which a random assignment is duplicated and a constant is added to the right-hand side of the duplicated statement, introducing an unintended state change. We ensure that the mutated Python programs semantically differ from the originals by verifying that they fail at least one assertion from the test suite. For each mutation type, we generate a separate transformed version of the benchmark, producing up to one mutated variant per program, with at most one mutation applied. These code mutations have been previously employed in prior work (Allamanis, Jackson-Flux, and Brockschmidt 2021; Orvalho, Janota, and Manquinho 2022; Maveli, Vergari, and Cohen 2025) to augment datasets with buggy programs, enabling the simulation of realistic faults. Although

Language Model	LiveCodeBench	Refactory
Qwen2.5-Coder (32B)	83.7%	92.0%
DeepSeek-Coder-V2 (16B)	65.1%	64.8%
GraniteCode (8B)	55.9%	52.0%
Llama3.2 (3B)	43.0%	28.0%

Table 1: 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.

these mutations are simple, they enable a systematic evaluation of whether PyVeritas can accurately identify fault locations via MaxSAT-based localisation.

Large Language Models (LLMs). In our evaluation, we exclusively used open-access LLMs available on Hugging Face (HuggingFace 2025) with at most 32 billions parameters, for two main reasons. First, closed-access models, for example, ChatGPT and DeepSeek, are cost-prohibitive and raise concerns regarding data privacy. Second, large-scale models (e.g., those with 70B parameters) demand significant computational resources, rendering them unsuitable for local use and out of reach for many developers. Thus, we evaluated four different LLMs using the iterative querying setup described in Section 4. Three of these models are LLMCs, i.e., LLMs fine-tuned specifically for coding tasks: Alibaba's Qwen2.5-Coder (Qwen 2024) (32B), DeepSeek's DeepSeek-Coder-V2 (DeepSeek-AI 2024) (16B), and IBM's GraniteCode (Granite 2024) (8B). As a sanity check, we used Meta's Llama3.2 (Llama3 2024) (3B), which is a general-purpose LLM not specifically trained for coding tasks. To ensure consistency, all models were run with temperature set to 0.

Program Verification for Python

Our first experiment aimed to assess the reliability of LLMs in transpiling Python to C, taking into account additional indication of intent, and to evaluate how accurately the verification of the transpiled C code reflects the correctness of the original Python program.

Baseline. We tried to use ESBMC-Python (Farias et al. 2024), version 7.9, as a baseline for our experiments. However, it was unable to verify any of the Python programs from LiveCodeBench or Refactory due to unsupported Python features. As discussed in the motivation section, ESBMC-Python currently supports only a limited subset of the language and is designed to verify simple constructs in isolation. When a program combines multiple features, such as, auxiliary functions, or control-flow constructs (e.g., `for` loops), ESBMC-Python is unable to analyse the program, resulting in failed verification or internal errors.

PyVeritas. To answer **RQ1**, we used CBMC to verify the LLM-transpiled C code against the provided assertions. All interactions between the evaluated LLMs and each Python program in both benchmarks were manually inspected to determine whether, in cases where verification succeeded, the transpiled code and assertions were semantically equivalent to the original Python program. This inspection was blind to the identity of the LLMs. Each model response was independently labelled three times by experts with strong programming backgrounds to ensure consistency, and any discrepancies resolved through discussion. Table 1 reports the verification success rate of each model on each benchmark. Qwen2.5-Coder demonstrates the highest reliability, with 83.7% success on LiveCodeBench and 92.0% on Refactory, indicating that its C translations are frequently both syntactically valid and semantically faithful to the original Python programs. DeepSeek-Coder-V2 follows, though with a marked decline in performance, where its verification success falls to around 65% on both benchmarks. This contrast indicates that Qwen-2.5-Coder more consistently produces semantically faithful and verifiable C translations. GraniteCode and Llama3.2 exhibit lower verification success across both benchmarks. Notably, GraniteCode achieves only 52.0% on Refactory. Similarly, Llama3.2 underperforms across both benchmarks, likely due to being a general-purpose model. Interestingly, we observe that larger models yield more verifiable translations, suggesting a correlation between model capacity and verification reliability, which warrants further investigation. Overall, these results indicate that, although LLMs vary considerably in reliability, models such as Qwen2.5-Coder already provide sufficiently dependable transpilation to support formal verification of non-trivial Python programs.

MaxSAT-Based Fault Localisation for Python

To assess the effectiveness of PyVeritas in localising faults in Python programs, we applied MaxSAT-based fault localisation to C code produced by LLM-based transpilation of two benchmarks: Refactory and LiveCodeBench. As described earlier, each correct Python program was first mutated using one of two semantics-altering transformations: *Wrong Binary Operator (WBO)* or *Assignment Duplication with Constant (ADC)*. These mutations are designed to deliberately introduce subtle bugs while preserving syntactic correctness. As described in Section 4, each mutated Python program was then passed through the full PyVeritas pipeline: LLM-based transpilation, verification using CBMC, and fault localisation with CFaults. We note that providing redundant information, such as natural language descriptions and explicit assertions, can strongly signal the intended behaviour of a program, guiding the LLM towards more semantically aligned translations (Ye et al. 2025).

Tables 2 and 3 present the results for Refactory and LiveCodeBench, respectively. For each LLM and bug type, we report: the percentage of cases where CFaults successfully identified the injected fault (*%Correct Bug Localised*), the percentage of cases where unrelated valid fault locations were reported (*%Other Bugs Localised*), the percentage of instances in which the LLM inadvertently corrected the bug during transpilation (*%Transpiled Fixed Code*), and the percentage of transpiled programs that failed to compile (*%Compilation Errors*). Since all prompts include both a natural language description and the ground-

	Bug: **Wrong Binary Operator (WBO)**			
LLMs	**% Correct Bug Localised**	**% Other Bugs Localised**	**% Transpiled Fixed Code**	**% Compilation Errors**
QWEN2.5-CODER	36.2%	14.3%	49.5%	0.0%
DEEPSEEK-CODER-V2	31.4%	19.0%	44.8%	4.8%
GRANITECODE	52.4%	29.5%	8.6%	9.5%
LLAMA3.2	30.5%	23.8%	3.8%	41.9%
	Bug: **Assignment Duplication with Constant (ADC)**			
	% Correct Bug Localised	**% Other Bugs Localised**	**% Transpiled Fixed Code**	**% Compilation Errors**
QWEN2.5-CODER	37.5%	10.0%	52.5%	0.0%
DEEPSEEK-CODER-V2	50.0%	12.5%	32.5%	5.0%
GRANITECODE	45.0%	27.5%	17.5%	10.0%
LLAMA3.2	27.4%	15.0%	7.5%	49.9%

Table 2: Results of MaxSAT-Based Fault Localisation with PYVERITAS on REFACTORY using two fault types: WBO and ADC.

	Bug: **Wrong Binary Operator (WBO)**			
LLMs	**% Correct Bug Localised**	**% Other Bugs Localised**	**% Transpiled Fixed Code**	**% Compilation Errors**
QWEN2.5-CODER	16.9%	20.2%	62.6%	0.3%
DEEPSEEK-CODER-V2	16.1%	36.8%	42.9%	4.2%
GRANITECODE	41.6%	34.6%	15.2%	8.6%
LLAMA3.2	22.7%	47.9%	17.7%	11.6%
	Bug: **Assignment Duplication with Constant (ADC)**			
	% Correct Bug Localised	**% Other Bugs Localised**	**% Transpiled Fixed Code**	**% Compilation Errors**
QWEN2.5-CODER	7.6%	11.0%	81.4%	0.0%
DEEPSEEK-CODER-V2	6.7%	21.4%	69.5%	2.4%
GRANITECODE	39.5%	11.9%	36.7%	11.9%
LLAMA3.2	18.6%	39.0%	34.3%	8.1%

Table 3: Results of MaxSAT-Based Fault Localisation with PYVERITAS on LIVECODEBENCH using WBO and ADC bugs.

truth specification, reasoning-oriented models fine-tuned for code often use this information to correct buggy code during transpilation. This results in the removal of duplicated assignments or the correction of incorrect binary operators, rather than preserving the original faulty logic. This observation aligns with findings from prior work (Ye et al. 2025), which show that providing problem descriptions and specifications improves code generation quality. As this information may contains inconsistencies, our experiments indicate that different LLMs handle these situations differently. We ran an additional experiment (results in Appendix B) by excluding the textual descriptions, observing that the rates of bug fixing decline but the underlying behaviour observed previously persists. Furthermore, we observe that unrelated localised faults may stem from two main factors. First, incorrect transpilation may produce buggy C code that is not semantically equivalent to the original buggy Python program, causing the fault localiser to highlight different statements. Second, the generated C code may introduce low-level issues, such as memory errors, that are correctly flagged but unrelated to the injected fault. Please refer to Appendix C for examples where fault localisation succeeded, as well as cases where the LLMS inadvertently fixed the buggy code during transpilation.

For the WBO bug on REFACTORY, in Table 2, GRANITECODE achieved the highest localisation accuracy at 52.4%, suggesting strong preservation of the faulty semantics. QWEN2.5-CODER and DEEPSEEK-CODER-V2 followed with 36.2% and 31.4%, respectively, while LLAMA3.2 reached 30.5%. Notably, QWEN2.5-CODER exhibited the highest fix rate (49.5%), often rewriting the faulty logic into a correct version, whereas GRANITECODE 's low fix rate (8.6%) indicates stronger fidelity to the original code. LLAMA3.2 showed the highest compilation error rate (41.9%), reflecting its general-purpose nature. For the ADC bug, DEEPSEEK-CODER-V2 performed best on REFACTORY (50.0%), followed by GRANITECODE (45.0%) and QWEN-2.5-CODER (37.5%). LLAMA3.2 again had the lowest success rate (27.4%) and the highest error rate (49.9%). QWEN-2.5-CODER 's 52.5% fix rate suggests it frequently removed the injected bug. GRANITECODE showed a relatively high rate of unrelated fault reports (27.5%), due to incorrect code transpilations. In Table 3, localisation success rates declined across models on LIVECODEBENCH, except for GRANITECODE, which remained strong. For WBO, GRANITECODE again led (41.6%), while QWEN2.5-CODER and DEEPSEEK-CODER-V2 dropped to 16.9% and 16.1%. LLAMA-3.2 reached 22.7%, but with 47.9% unrelated fault reports. QWEN2.5-CODER and DEEPSEEK-CODER-V2 also showed high fix rates (62.6% and 42.9%), often repairing the bug during translation. Compilation errors were low overall, except for LLAMA3.2 (11.6%). For ADC on LIVE-

CodeBench, the trend persisted. GraniteCode again led (39.5%), followed by Llama3.2 (18.6%). Qwen2.5-Coder and DeepSeek-Coder-V2 had the lowest success rates (7.6% and 6.7%) and the highest fix rates (81.4% and 69.5%), indicating frequent bug removal during translation. Llama3.2 also showed a high rate of unrelated fault reports (39.0%), due to incorrect transpilations.

Taken together, these results confirm that MaxSAT-based fault localisation on LLM-transpiled C code is both feasible and effective, answering **RQ2**. When that transpilation preserves the faulty semantics, CFaults can successfully identify the root cause of the bug, even though it originated in a Python program. Notably, this holds across different fault types and benchmarks. In addressing **RQ3**, we find that the reliability of LLM-based transpilation for enabling C-level analysis of Python code varies significantly across models. Qwen2.5-Coder achieves the highest verification success, with 92.0% of programs verified on Refactory and 83.7% on LiveCodeBench, indicating well-structured and semantically coherent output. In contrast, GraniteCode yields lower verification rates (52.0% and 55.9%), but consistently preserves the faulty semantics, resulting in higher localisation success. Thus, we observe that models differ in how they leverage redundant information. Reasoning-oriented models like Qwen2.5-Coder and DeepSeek-Coder-V2 tend to generate chains of thought (CoT), integrating cues from natural language descriptions and assertions to guide their output, often repairing faulty code in the process. In contrast, models like GraniteCode prioritise structural fidelity, preserving the original syntax and semantics (including faults), which is crucial for fault localisation tasks where bug retention is required, underscoring the importance of model selection when using LLM-based transpilation to support verification and debugging workflows in PyVeritas. Moreover, we conducted experiments without providing natural language descriptions to the LLMs (see Appendix B). While localisation accuracy declined slightly, some LLMs continued to exhibit a tendency to fix the buggy code during transpilation.

6 Related Work

Recent work has explored the use of Large Language Models (LLMs) for code transpilation and synthesis, particularly across high-level programming languages. Tools such as CoTran (Jana et al. 2024) and Miracle (Zhu et al. 2024) leverage neural models for transpiling between languages like Python, Java, and C++, with benchmarks that assess functional correctness. However, these approaches typically focus on transpilation in isolation and do not target the specific Python-to-C direction, nor do they integrate downstream formal verification. Large-scale empirical studies (Xue et al. 2024) reveal that LLM-generated transpilations often suffer from high error rates (ranging from 50% to 98%), and they identify common issues such as semantic hallucinations, inconsistent control flow, and undeclared variables. Nonetheless, these studies are limited to error analysis and do not connect transpilation outputs to formal guarantees. Work on verifying C code through deductive or bounded model checking techniques has progressed independently. SynVer (Mukherjee and Delaware 2024), for instance, uses LLMs to generate C implementations from formal specifications, integrating deductive verification backends. However, it does not address the challenge of transpiling from Python while preserving the semantics or intent of the original code. Moreover, there is currently no demonstrated pipeline that translates Python code into verified C annotated with specifications. Recently, BatFix (Ramos et al. 2024) augmented LLM-based code transpilers by applying program repair and synthesis to fix faulty translations, producing C++ code from Python or Java that passes all test cases. Bhatia et al. (2024) use LLMs to transpile code into low-level intermediate representations of target languages, while generating formal proofs to ensure semantic equivalence, enabling verified code translation. Finally, (Zhang et al. 2025) used LLMs to translate large codebases (e.g. Go to Rust) by partitioning files, incorporating feature-mapping rules, and validating input-output equivalence to achieve 73% function-level correctness. A separate body of work uses property-based testing and mutation analysis to assess functional equivalence between original and transpiled programs (Guizzo et al. 2024; Eniser, Wüstholz, and Christakis 2024). These techniques provide empirical evidence of correctness but typically rely on test generation and lack formal guarantees. Finally, several studies have documented the frequent structural and semantic errors introduced by LLM-based transpilers (Zhu et al. 2024; Xue et al. 2024). Our work complements this line of research by examining the feasibility of connecting LLM-based transpilation from Python to C with subsequent formal verification and MaxSAT-based fault localisation, thereby enabling correctness guarantees beyond testing alone.

7 Conclusion

In this paper, we introduced PyVeritas, a framework for verifying Python programs via LLM-based transpilation to C, followed by assertion-based verification using tools such as CBMC, and MaxSAT-based fault localisation via CFaults. Our experiments surprisingly show that LLMs can generate semantically faithful C translations for small but non-trivial Python programs, enabling successful verification in over 80% of cases with some models, notably Qwen2.5-Coder. We also demonstrated that MaxSAT-based fault localisation on the transpiled C code can effectively identify injected faults in the original Python source. Interestingly, we found that reasoning-oriented models, such as Qwen2.5-Coder and DeepSeek-Coder-V2, often leveraged redundant cues, such as natural language descriptions and assertions, to implicitly repair code during transpilation, while models like GraniteCode prioritised structural fidelity, making them better suited for MaxSAT-based fault localisation. These findings suggest that, despite the current limitations of Python model checkers, PyVeritas offers a practical and effective alternative by leveraging mature C verification tools. Furthermore, we demonstrated the feasibility of applying MaxSAT-based fault localisation to Python using simple injected faults as a proof of concept. As LLMs continue to improve, we anticipate even broader applicability to more complex bugs and verification scenarios.

Acknowledgments

This project received funding from the ERC under the European Union's Horizon 2020 research and innovation programme (FUN2MODEL, grant agreement No. 834115) and ELSA: European Lighthouse on Secure and Safe AI project (grant agreement No. 101070617 under UK guarantee).

References

Allamanis, M.; Jackson-Flux, H.; and Brockschmidt, M. 2021. Self-Supervised Bug Detection and Repair. In *NeurIPS*.

Bacchus, F.; Järvisalo, M.; and Martins, R. 2021. Maximum Satisfiability. In Biere, A.; Heule, M.; van Maaren, H.; and Walsh, T., eds., *Handbook of Satisfiability - Second Edition*, volume 336 of *Frontiers in Artificial Intelligence and Applications*, 929–991. IOS Press.

Bhatia, S.; Qiu, J.; Hasabnis, N.; Seshia, S.; and Cheung, A. 2024. Verified Code Transpilation with LLMs. In Globersons, A.; Mackey, L.; Belgrave, D.; Fan, A.; Paquet, U.; Tomczak, J. M.; and Zhang, C., eds., *Advances in Neural Information Processing Systems 38: Annual Conference on Neural Information Processing Systems 2024, NeurIPS 2024, Vancouver, BC, Canada, December 10 - 15, 2024*.

Biere, A.; Cimatti, A.; Clarke, E. M.; Strichman, O.; and Zhu, Y. 2003. Bounded model checking. *Adv. Comput.*, 58: 117–148.

Biere, A.; Heule, M.; van Maaren, H.; and Walsh, T., eds. 2009. *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*. IOS Press.

Chong, N.; Cook, B.; Kallas, K.; Khazem, K.; Monteiro, F. R.; Schwartz-Narbonne, D.; Tasiran, S.; Tautschnig, M.; and Tuttle, M. R. 2020. Code-level model checking in the software development workflow. In Rothermel, G.; and Bae, D., eds., *ICSE-SEIP 2020: 42nd International Conference on Software Engineering, Software Engineering in Practice, Seoul, South Korea, 27 June - 19 July, 2020*, 11–20. ACM.

Cimatti, A.; and Griggio, A. 2012. Software Model Checking via IC3. In Madhusudan, P.; and Seshia, S. A., eds., *Computer Aided Verification - 24th International Conference, CAV 2012, Berkeley, CA, USA, July 7-13, 2012 Proceedings*, volume 7358 of *Lecture Notes in Computer Science*, 277–293. Springer.

Clarke, E. M. 1997. Model Cheking. In Ramesh, S.; and Sivakumar, G., eds., *Foundations of Software Technology and Theoretical Computer Science, 17th Conference, Kharagpur, India, December 18-20, 1997, Proceedings*, volume 1346 of *Lecture Notes in Computer Science*, 54–56. Springer.

Clarke, E. M.; and Kroening, D. 2003. Hardware verification using ANSI-C programs as a reference. In Yasuura, H., ed., *Proceedings of the 2003 Asia and South Pacific Design Automation Conference, ASP-DAC '03, Kitakyushu, Japan, January 21-24, 2003*, 308–311. ACM.

Clarke, E. M.; Kroening, D.; and Lerda, F. 2004. A Tool for Checking ANSI-C Programs. In Jensen, K.; and Podelski, A., eds., *Tools and Algorithms for the Construction and Analysis of Systems, 10th International Conference, TACAS 2004, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2004, Barcelona, Spain, March 29 - April 2, 2004, Proceedings*, volume 2988 of *Lecture Notes in Computer Science*, 168–176. Springer.

Clarke, E. M.; Kroening, D.; and Yorav, K. 2003. Behavioral consistency of C and verilog programs using bounded model checking. In *Proceedings of the 40th Design Automation Conference, DAC 2003, Anaheim, CA, USA, June 2-6, 2003*, 368–371. ACM.

Cython. 2025. . https://cython.org. [Online; accessed 10-October-2025].

DeepSeek-AI. 2024. DeepSeek-Coder-V2: Breaking the Barrier of Closed-Source Models in Code Intelligence. *CoRR*, abs/2406.11931.

Eniser, H. F.; Wüstholz, V.; and Christakis, M. 2024. Automatically Testing Functional Properties of Code Translation Models. In Wooldridge, M. J.; Dy, J. G.; and Natarajan, S., eds., *Thirty-Eighth AAAI Conference on Artificial Intelligence, AAAI 2024, Vancouver, Canada*, 21055–21062. AAAI Press.

Farias, B.; Menezes, R.; de Lima Filho, E. B.; Sun, Y.; and Cordeiro, L. C. 2024. ESBMC-Python: A Bounded Model Checker for Python Programs. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2024*, 1836–1840. ACM.

Granite. 2024. Granite Code Models: A Family of Open Foundation Models for Code Intelligence. *CoRR*, abs/2405.04324.

Guizzo, G.; Zhang, J. M.; Sarro, F.; Treude, C.; and Harman, M. 2024. Mutation analysis for evaluating code translation. *Empir. Softw. Eng.*, 29(1): 19.

Hartonas-Garmhausen, V.; Campos, S. V. A.; Cimatti, A.; Clarke, E. M.; and Giunchiglia, F. 2000. Verification of a safety-critical railway interlocking system with real-time constraints. *Sci. Comput. Program.*, 36(1): 53–64.

Hu, Y.; Ahmed, U. Z.; Mechtaev, S.; Leong, B.; and Roychoudhury, A. 2019. Re-Factoring Based Program Repair Applied to Programming Assignments. In *ASE 2019*, 388–398. IEEE.

Huang, H.; Tsai, W.; Paul, R. A.; and Chen, Y. 2005. Automated Model Checking and Testing for Composite Web Services. In *Eighth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC 2005), 18-20 May 2005, Seattle, WA, USA*, 300–307. IEEE Computer Society.

HuggingFace. 2025. . https://huggingface.co. [Online; accessed 15-July-2025].

Ignatiev, A.; Morgado, A.; Weissenbacher, G.; and Marques-Silva, J. 2019. Model-Based Diagnosis with Multiple Observations. In Kraus, S., ed., *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence, IJCAI 2019, Macao, China, August 10-16, 2019*, 1108–1115.

Jain, N.; Han, K.; Gu, A.; Li, W.; Yan, F.; Zhang, T.; Wang, S.; Solar-Lezama, A.; Sen, K.; and Stoica, I. 2024. LiveCodeBench: Holistic and Contamination Free Evaluation of Large Language Models for Code. *CoRR*, abs/2403.07974.

Jana, P.; Jha, P.; Ju, H.; Kishore, G.; Mahajan, A.; and Ganesh, V. 2024. CoTran: An LLM-Based Code Translator Using Reinforcement Learning with Feedback from Compiler and Symbolic Execution. In Endriss, U.; Melo, F. S.; Bach, K.; Diz, A. J. B.; Alonso-Moral, J. M.; Barro, S.; and Heintz, F., eds., *27th European Conference on Artificial Intelligence, ECAI 2024*, volume 392 of *Frontiers in Artificial Intelligence and Applications*, 4011–4018. IOS Press.

Jose, M.; and Majumdar, R. 2011. Cause clue clauses: error localization using maximum satisfiability. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011*, 437–446. ACM.

Joshi, H.; Sánchez, J. P. C.; Gulwani, S.; Le, V.; Verbruggen, G.; and Radicek, I. 2023. Repair Is Nearly Generation: Multilingual Program Repair with LLMs. In *AAAI 2023*, 5131–5140. AAAI Press.

Lahtinen, J.; Valkonen, J.; Björkman, K.; Frits, J.; Niemelä, I.; and Heljanko, K. 2012. Model checking of safety-critical software in the nuclear engineering domain. *Reliab. Eng. Syst. Saf.*, 105: 104–113.

LeetCode. 2025. https://leetcode.com. Accessed: 2025-07-15.

Li, Y.; Parsert, J.; and Polgreen, E. 2024. Guiding Enumerative Program Synthesis with Large Language Models. In Gurfinkel, A.; and Ganesh, V., eds., *Computer Aided Verification - 36th International Conference, CAV 2024, Montreal, QC, Canada, July 24-27, 2024, Proceedings, Part II*, volume 14682 of *Lecture Notes in Computer Science*, 280–301. Springer.

Llama3. 2024. The Llama 3 Herd of Models. arXiv:2407.21783.

Marques-Silva, J.; Janota, M.; Ignatiev, A.; and Morgado, A. 2015. Efficient Model Based Diagnosis with Maximum Satisfiability. In *IJCAI 2015*, 1966–1972. AAAI Press.

Maveli, N.; Vergari, A.; and Cohen, S. B. 2025. What can Large Language Models Capture about Code Functional Equivalence? In Chiruzzo, L.; Ritter, A.; and Wang, L., eds., *Findings of the Association for Computational Linguistics: NAACL 2025, Albuquerque, New Mexico, USA, April 29 - May 4, 2025*, 6865–6903. Association for Computational Linguistics.

Mukherjee, P.; and Delaware, B. 2024. Towards Automated Verification of LLM-Synthesized C Programs. *CoRR*, abs/2410.14835.

Orvalho, P.; Janota, M.; and Manquinho, V. 2024. CFaults: Model-Based Diagnosis for Fault Localization in C Programs with Multiple Test Cases. In *Formal Methods - 26th International Symposium, FM 2024*, volume 14933 of *Lecture Notes in Computer Science*, 463–481. ISBN 978-3-031-71162-6.

Orvalho, P.; Janota, M.; and Manquinho, V. M. 2022. MultIPAs: Applying Program Transformations To Introductory Programming Assignments For Data Augmentation. In *ESEC/FSE 2022*, 1657–1661. ACM.

Orvalho, P.; Janota, M.; and Manquinho, V. M. 2025. Counterexample Guided Program Repair Using Zero-Shot Learning and MaxSAT-based Fault Localization. *Proceedings of the AAAI Conference on Artificial Intelligence*, 39(1): 649–657.

Qwen. 2024. Qwen2.5-Coder Technical Report. arXiv:2409.12186.

Ramos, D.; Lynce, I.; Manquinho, V.; Martins, R.; and Le Goues, C. 2024. BatFix: Repairing language model-based transpilation. *ACM Trans. Softw. Eng. Methodol.*, 33(6): 161.

Reiter, R. 1987. A Theory of Diagnosis from First Principles. *Artif. Intell.*, 32(1): 57–95.

Shedskin. 2025. . https://shedskin.github.io. [Online; accessed 23-July-2025].

Xue, P.; Wu, L.; Wang, C.; Li, X.; Yang, Z.; Jin, R.; Zhang, Y.; Li, J.; Pei, Y.; Shen, Z.; and Lyu, X. 2024. Escalating LLM-based Code Translation Benchmarking into the Class-level Era. *CoRR*, abs/2411.06145.

Ye, Z.; Yan, Z.; He, J.; Kasriel, T.; Yang, K.; and Song, D. 2025. VERINA: Benchmarking Verifiable Code Generation. *CoRR*, abs/2505.23135.

Zhang, H.; David, C.; Wang, M.; Paulsen, B.; and Kroening, D. 2025. Scalable, Validated Code Translation of Entire Projects using Large Language Models. *Proc. ACM Program. Lang.*, 9(PLDI).

Zhang, J.; Cambronero, J. P.; Gulwani, S.; Le, V.; Piskac, R.; Soares, G.; and Verbruggen, G. 2024. PyDex: Repairing Bugs in Introductory Python Assignments using LLMs. ., 8(OOPSLA): 1100–1124.

Zhu, M.; Karim, M.; Lourentzou, I.; and Yao, D. 2024. Semi-Supervised Code Translation Overcoming the Scarcity of Parallel Code Data. In Filkov, V.; Ray, B.; and Zhou, M., eds., *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering, ASE 2024, Sacramento, CA, USA, October 27 - November 1, 2024*, 1545–1556. ACM.

A Prompts

In this section, we present the prompt templates used for all interactions with the Large Language Models (LLMs).

- Transpilation prompt:

  Given a natural language description (NL_Description), a Python program (Python code), and one or more Python assertions (assertion), the transpilation prompt asks the model to transpile the Python code and assertion to semantically equivalent C code, as follows:

```
Transpile Python to C Code With Assertion:
You are an exceptionally intelligent coding
assistant who consistently produces accurate
and reliable <C code> by transpiling the given
<Python code> into semantically equivalent
<C code>. <NL_Description> gives a natural
language description of the python code.
Do not forget to  also transpile the given
Python assertion  into a C assertion!

<NL_Description>
{description}
```

Language Model	LiveCodeBench	Refactory
Qwen2.5-Coder (32B)	80.4%	87.2%
DeepSeek-Coder-V2 (16B)	63.5%	61.6%
GraniteCode (8B)	53.0%	49.6%
Llama3.2 (3B)	30.5%	22.4%

Table 4: Verification success rates for each LLM on both benchmarks **without providing any natural language description**. Percentages indicate the proportion of C programs that were successfully verified by CBMC and judged semantically equivalent to the original Python code.

````
<Python Code>
```python
{python_code}
{assertion}
```

<C Code>
```c
````

- Prompt to ask for a different C candidate:

  We provide an explanation for why the C translation was incorrect, either indicating that the C code failed to compile (along with the corresponding error message), or that one or more assertions was not successful. This feedback is included in the prompt, and the original transpilation prompt is appended to the end, allowing the model to attempt the translation again.

```
Your previous code translation was INCORRECT!
Reason: {Reason}
Try again.
{Transpilation Prompt}
```

- Prompt to map the buggy C statements back to the original Python code:

````
Map C program statements back to the original python
program:
We have detected that both the Python and C programs
are buggy. We have localised the following faulty
statements in the C program:
{list of faulty C statements}

Provide us only with the corresponding Python
statements from the original program that
correspond to these buggy statements.
```python
````

B Experiments Without Natural Language Descriptions

To understand the role of natural language descriptions in guiding LLM-based transpilation, we repeated our experiments without providing any such descriptions to the models. This setup allows us to assess the reliability of PyVeritas when the LLM must rely solely on the Python code and assertion-based specifications.

As shown in Table 4, verification success rates declined only modestly for larger models. For instance, Qwen2.5-Coder maintained high success rates at 80.4% on LiveCodeBench and 87.2% on Refactory (compared to 83.7% and 92.0% with descriptions). In contrast, smaller models such as Llama3.2 experienced a more significant drop, with success rates falling to 30.5% and 22.4%, respectively. These findings suggest that high-capacity models are still capable of producing semantically faithful C code, even without natural language guidance.

Tables 5 and 6 show the results of MaxSAT-based fault localisation under the same conditions. While localisation accuracy declined slightly, some LLMs continued to exhibit a tendency to fix the buggy code during transpilation. For instance, Qwen2.5-Coder still fixed 52.5% of *Assignment duplication with constant (ADC)* cases on Refactory. This suggests that the tendency to repair code is not exclusively driven by natural language cues, but can also emerge from the models' reasoning approach.

Despite the absence of descriptions, GraniteCode maintained strong localisation performance. For example, it correctly identified the bug in 42.5% of ADC cases on Refactory and 39.5% on LiveCodeBench, only slightly lower than with descriptions. This further reinforces GraniteCode's preference for structural fidelity over semantic correction, making it particularly well-suited for fault localisation tasks.

In summary, it appears that natural language descriptions enhance verification and often prompt models to repair faults during transpilation. Removing them reduces this behaviour only marginally, sometimes improving localisation accuracy.

C Interactions with Large Language Models (LLMs)

In this section, we present representative examples of some interactions with LLMs.

- Example of DeepSeek-Coder-V2 inadvertently fixing the buggy Python program #188 from LiveCodeBench during transpilation to C:

  – Natural Language Description: You are given a string `moves` of length *n* consisting only of characters `'L'`, `'R'`, and `'_'`. The string represents your movement on a number line starting from the origin 0. In the ith move, you can choose one of the following directions: Move to the left if `moves[i] = 'L'` or `moves[i] = '_'`; or; move to the right if `moves[i] = 'R'` or `moves[i] = '_'`. You should return the distance from the origin of the furthest point you can get to after `n` moves.

  – Python code:

LLMs	% Correct Bug Localised	% Other Bugs Localised	% Transpiled Fixed Code	% Compilation Errors
	Bug: **Wrong Binary Operator (WBO)**			
QWEN2.5-CODER	28.6%	32.4%	39.0%	0.0%
DEEPSEEK-CODER-V2	20.0%	49.5%	18.1%	12.4%
GRANITECODE	41.0%	32.4%	13.3%	13.3%
LLAMA3.2	25.2%	26.1%	0.9%	47.8%
	Bug: **Assignment Duplication with Constant (ADC)**			
	% Correct Bug Localised	**% Other Bugs Localised**	**% Transpiled Fixed Code**	**% Compilation Errors**
QWEN2.5-CODER	22.5%	22.5%	52.5%	2.5%
DEEPSEEK-CODER-V2	25.0%	20.0%	47.5%	7.5%
GRANITECODE	42.5%	35.0%	15.0%	7.5%
LLAMA3.2	19.9%	15.0%	2.5%	62.3%

Table 5: Results of MaxSAT-Based Fault Localisation with PYVERITAS on REFACTORY using two fault types: WBO and ADC **without providing any natural language description**.

LLMs	% Correct Bug Localised	% Other Bugs Localised	% Transpiled Fixed Code	% Compilation Errors
	Bug: **Wrong Binary Operator (WBO)**			
QWEN2.5-CODER	29.6%	46.0%	23.5%	0.8%
DEEPSEEK-CODER-V2	18.0%	58.2%	19.4%	4.4%
GRANITECODE	42.9%	36.0%	11.4%	9.7%
LLAMA3.2	27.7%	51.5%	8.3%	12.5%
	Bug: **Assignment Duplication with Constant (ADC)**			
	% Correct Bug Localised	**% Other Bugs Localised**	**% Transpiled Fixed Code**	**% Compilation Errors**
QWEN2.5-CODER	16.2%	29.5%	53.3%	1.0%
DEEPSEEK-CODER-V2	3.3%	37.6%	54.8%	4.3%
GRANITECODE	35.7%	14.3%	37.6%	12.4%
LLAMA3.2	16.2%	36.2%	30.5%	17.1%

Table 6: Results of MaxSAT-Based Fault Localisation with PYVERITAS on LIVECODEBENCH using WBO and ADC bugs **without providing any natural language description**.

Algorithm 3: Program #188 from LIVECODEBENCH, with a duplicated assignment with the added constant introduced on line 3.

```python
1  def furthestDistanceFromOrigin(moves: str)->int:
2      left = 0
3      left = 0 + 1
4      right = 0
5      a = 0
6      ret = 0
7      for c in moves:
8          if c == 'L':
9              left += 1
10         elif c == 'R':
11             right += 1
12         else:
13             a += 1
14     ret = max(ret, abs(left - right) + a)
15     return ret
16
17 assert furthestDistanceFromOrigin(moves='L_RL__R')==3
```

- We fed this information to DEEPSEEK-CODER-V2 using our transpilation prompt.
- DEEPSEEK-CODER-V2's response:

````
```c
#include <assert.h>
#include <stdio.h>
#include <stdlib.h>

int furthestDistanceFromOrigin(const char* moves){
  int left = 0;
  int right = 0;
  int a = 0;
  int ret = 0;
  for (int i = 0; moves[i] != '\0'; i++) {
      if (moves[i] == 'L') {
        left++;
      } else if (moves[i] == 'R') {
        right++;
      } else {
        a++;
      }
  }
  ret = abs(left - right) + a;
  return ret;
}
int main() {
  const char* moves = "L_RL__R";
  assert(furthestDistanceFromOrigin(moves)==3);
  return 0;
}
```
This C code transpiles the given Python code
into a C program. It includes the necessary
C library for assertions (`assert.h`) and
ensures that the `furthestDistanceFromOrigin`
function is tested with the provided example
`moves = "L_RL__R"`, which should return `3`.
The `main` function includes an assertion to
check the result.
````

- Thus, DEEPSEEK-CODER-V2 transpiled Algorithm 3 to C, and during this process the model corrected the program's semantics by removing the duplicated assignment with the added constant that we introduced on line 3, `left = 0 + 1`.

- Example of program #57 from LiveCodeBench, where MaxSAT-based fault localisation for C successfully identified a *wrong binary operator (WBO)* in the original Python code:
  - Natural Language Description: A truck has two fuel tanks. You are given two integers, `mainTank` representing the fuel present in the main tank in liters and `additionalTank` representing the fuel present in the additional tank in liters. The truck has a mileage of `10` KM per liter. Whenever `5` liters of fuel get used up in the main tank, if the additional tank has at least `1` liters of fuel, `1` liters of fuel will be transferred from the additional tank to the main tank. You should return the maximum distance which can be traveled. Note: Injection from the additional tank is not continuous. It happens suddenly and immediately for every `5` liters consumed.
  - Python code:

Algorithm 4: Program #57 from LiveCodeBench, with a wrong binary operator introduced on line 4.

```python
1  def distanceTraveled(a: int, b: int) -> int:
2      res = 0
3      while a:
4          if a < 5:
5              res += 5 * 10
6              a -= 5
7              if b:
8                  b -= 1
9                  a += 1
10         else:
11             res += a * 10
12             a = 0
13     return res
14
15 assert distanceTraveled(a = 1, b = 2) == 10
```

  - We fed this information to GraniteCode using our transpilation prompt.
  - GraniteCode's response:

```
Here is the C code that transpiles the given
Python code:
```c
#include <assert.h>
int distanceTraveled(int mainTank,
int additionalTank){
    int res = 0;
    while (mainTank) {
        if (mainTank < 5) {
            res += 5 * 10;
            mainTank -= 5;
            if (additionalTank) {
                additionalTank -= 1;
                mainTank += 1;
            }
        } else {
            res += mainTank * 10;
            mainTank = 0;
        }
    }
    return res;
}
int main() {
    assert(distanceTraveled(1, 2) == 10);
    return 0;
}
```
In this C code, we have defined a function
`distanceTraveled` that takes two integer
arguments `mainTank` and `additionalTank`
and returns an integer value.
The function uses a `while` loop to iterate
until `mainTank` becomes zero.
Inside the loop, we have used an `if`
statement to check if `mainTank` is less
than 5.
If it is, we add 50 to the `res` variable
and subtract 5 from `mainTank`.
If `additionalTank` is not zero, we
subtract 1 from `additionalTank` and
add 1 to `mainTank`.
If `mainTank` is not less than 5, we add
the current value of `mainTank` multiplied
by 10 to the `res` variable and
set `mainTank` to zero.
Finally, we have defined a `main` function
that uses the `assert` macro from the
`assert.h` library to check if the
`distanceTraveled` function returns the
expected value for the given input.
```

  - GraniteCode transpiled Algorithm 4 to C (note that GraniteCode actually took the variable names from the description despite renaming the variables during this process); it preserved the program's semantics by retaining the buggy binary operator on line 4 of Python code.
  - We then invoke CFaults to perform MaxSAT-based fault localisation on the C program, which identifies line 6 as faulty.
  - This information is then provided to GraniteCode using our mapping prompt, in order to trace the buggy C statement back to its corresponding line in the original Python program.
  - GraniteCode's response:

```
Here are the corresponding Python statements
from the original program that correspond
to these buggy statements:
```python
        if a < 5:
```
```

  - Thus, PyVeritas successfully localised line 4 of the original Python program using CFaults.

- On the other hand, Qwen2.5-Coder, when transpiling the same program (#57 from LiveCodeBench), fixed the binary operator, thereby removing the injected fault.

- Example of program #76 from LIVECODEBENCH, where MaxSAT-based fault localisation for C successfully identified a *wrong binary operator (WBO)* in the original Python code using QWEN2.5-CODER:
  - Natural Language Description: You are given two integers `num1` and `num2`. In one operation, you can choose integer `i` in the range $[0, 60]$ and subtract `2i + num2` from `num1`. Return the integer denoting the minimum number of operations needed to make `num1` equal to 0. If it is impossible to make `num1` equal to 0, return -1.
  - Python code:

Algorithm 5: Program #76 from LIVECODEBENCH, with a wrong binary operator introduced on line 4.

```python
1  def makeTheIntegerZero(num1:int, num2:int)->int:
2      a = 0
3      x, y = (num1, num2)
4      while x < y:
5          x -= y
6          a += 1
7          b = x
8          c = 0
9          cc = 0
10         z = 1
11         while b:
12             c += b % 2 == 1
13             cc += (b % 2 == 1) * z
14             z *= 2
15             b //= 2
16         if c <= a and cc >= a:
17             return a
18     return -1
19 assert makeTheIntegerZero(num1=5, num2=7)==-1
```

  - We fed this information to QWEN2.5-CODER using our transpilation prompt.
  - QWEN2.5-CODER's response:

```
To transpile the given Python code into C,
we need to carefully translate each part of
the code, including the logic and the assertion.
Here's the C code equivalent of the provided
Python code:
```c
#include <stdio.h>
#include <assert.h>
int makeTheIntegerZero(int num1, int num2) {
    int a = 0;
    int x = num1;
    int y = num2;
    while (x < y) {
        x -= y;
        a += 1;
        int b = x;
        int c = 0;
        int cc = 0;
        int z = 1;
        while (b) {
            c += (b \% 2 == 1);
            cc += (b \% 2 == 1) * z;
            z *= 2;
            b /= 2;
        }
        if (c <= a && cc >= a) {
            return a;
        }
    }
    return -1;
}
int main() {
    assert(makeTheIntegerZero(5, 7) == -1);
    printf("All assertions passed.\n");
    return 0;
}
```
Explanation:
1. **Function Definition**: The function
`makeTheIntegerZero` takes two integers
`num1` and `num2` as input and returns
an integer.
2. **Variable Initialization**: We initialize
`a` to 0, and `x` and `y` to `num1` and `num2`
respectively.
3. **While Loop**: The loop continues as long
as `x` is less than `y`. Inside the loop:
   - `x` is decremented by `y`.
   - `a` is incremented by 1.
   - Variables `b`, `c`, `cc`, and `z` are
   initialized. `b` is set to `x`, `c` and `cc`
   to 0, and `z` to 1.
4. **Inner While Loop**: This loop calculates
the number of 1s in the binary representation
of `b` (`c`) and the sum of powers of 2
corresponding to those 1s (`cc`).
   - `c` counts the number of 1s in the binary
   representation of `b`.
   - `cc` calculates the sum of powers of 2
   corresponding to the positions of 1s in `b`.
5. **Condition Check**: If `c` is less than or
equal to `a` and `cc` is greater than or equal
to `a`, the function returns `a`.
6. **Return Statement**: If the loop exits
without finding a valid `a`, the function
returns `-1`.
7. **Main Function**: The `main` function
includes an assertion to test the function
with the example provided. If the assertion
passes, it prints "All assertions passed."
This C code should behave the same way as the
provided Python code.
```

  - QWEN2.5-CODER transpiled Algorithm 5 to C; note that it preserved the program's semantics by retaining the buggy binary operator on line 4 of Python code.
  - We then invoke CFAULTS to perform MaxSAT-based fault localisation on the C program, which identifies line 7, `while (x < y)`, as faulty.
  - This information is then provided to QWEN2.5-CODER using our mapping prompt, in order to trace the buggy C statement back to its corresponding line in the original Python program.
  - QWEN2.5-CODER's response:

```
Certainly! Let's map the faulty statement in the
C program back to the corresponding statement in
the original Python program.
```python
    while x < y:
```
```

  - Thus, PYVERITAS successfully localised line 4 of the original Python program using CFAULTS.

- Finally, we present an interaction with an LLM, DeepSeek-Coder-V2, where the model not only fixed the original bug, a duplicated assignment, during code transpilation, but also introduced a new semantic error due to incorrect translation. This new fault was subsequently identified by CFaults. This example corresponds to program #463 from LiveCodeBench.
  - Natural Language Description: You are given two positive integers `n` and `limit`. Return the total number of ways to distribute `n` candies among 3 children such that no child gets more than `limit` candies.
  - Python code:

Algorithm 6: Program #463 from LiveCodeBench, with a duplicated assignment introduced on line 4.

```python
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
11 assert distributeCandies(n=5, limit=2) == 3
```

  - We fed this information to DeepSeek-Coder-V2 using our transpilation prompt.
  - DeepSeek-Coder-V2's response:

````
```c
#include <assert.h>
#include <stdio.h>
int distributeCandies(int n, int limit) {
    limit = (limit < n) ? limit : n;
    int ans = 0;
    for (int i = 0; i <= limit; i++) {
        if (n - i > limit * 2) {
            continue;
        }
        ans += (limit < n - i) ? limit :
            n - i - max(0, n - i - limit) + 1;
    }
    return ans;
}
int main() {
    assert(distributeCandies(5, 2) == 3);
    return 0;
}
```
Explanation:
1. The `distributeCandies` function calculates
the number of ways to distribute `n` candies
among 3 children such that no child gets more
than `limit` candies.
2. The `main` function includes the `assert`
statement to check if the `distributeCandies`
function returns the expected result for the
given input `n = 5` and `limit = 2`.
3. The `assert` statement in the `main`
function ensures that the result of
`distributeCandies(5, 2)` is equal to 3,
as per the example provided.
````

  - DeepSeek-Coder-V2 transpiled Algorithm 6 to C, and during this process the model fixed the injected bug, by removing line 4 (i.e., `ans = 0 + 1`), although it introduced a new semantic bug on line 10: `ans += (limit < n - i) ? limit : n - i - max(0, n - i - limit) + 1`. This C statements is buggy since it is not semantically equivalent to the Python statements shown on line 8. A semantically equivalent statement would be: `ans += ((limit < n-i) ? limit : (n-i)) - ((n-i-limit > 0) ?(n-i-limit) : 0) + 1;`
  - However, PyVeritas continues and invokes CFaults to perform MaxSAT-based fault localisation on the C program, which identifies line 10 as faulty.
  - This information is then provided to DeepSeek-Coder-V2 using our mapping prompt, in order to trace the buggy C statement back to its corresponding line in the original Python program.
  - DeepSeek-Coder-V2's response:

````
The buggy statement in the C program corresponds
to the Python statements from the original
program:
```python
   ans += min(limit, n-i)-max(0, n-i-limit)+1
```
````

  - Thus, PyVeritas localised line 8 of the original Python program using CFaults, due to DeepSeek-Coder-V2's incorrect code transpilation.