Multi-Programming Language Sandbox for LLMs (Supplementary Material)

1 Additional Related Work

1.1 Large language models for code

Recently, the advancement of LLMs (Jiang et al., 2023; Bai et al., 2023; DeepSeek-AI, 2024; gpt, 2023; OpenAI, 2023; Anthropic, 2024; Touvron et al., 2023; lla, 2023, 2024; Lu et al., 2021; Allal et al., 2023; Chen et al., 2021; Abdin et al., 2024) has significantly propelled the field of software engineering (Dou et al., 2024a; Jin et al., 2024; Xu et al., 2024; Nam et al., 2024). For instance, in code generation and program repair, state-of-the-art approaches improve the correctness and quality of LLM-generated codes by learning from compiler feedback. Specifically, researchers integrate compiler information into prompt templates to improve the performance of LLMs on these code-related tasks (Ren et al., 2024; Jiang et al., 2024a; Le et al., 2023; Shin et al., 2023; Denny et al., 2023). Some work also transforms compiler information into feedback signals to optimize LLMs to enhance their performance (Dou et al., 2024b; Yu et al., 2023; Le et al., 2022; Liu et al., 2023; Shojaee et al., 2023). For security, stability, reliability, and providing robust monitoring capability, these compilation and execution processes are needed in an isolated sandbox (Garfinkel et al., 2003; Guo et al., 2024; Liang et al., 2003). However, the development of open-source sandboxes is still in its early stages. Meanwhile, existing sandboxes developing for LLM-generated code mostly focus on a single programming language, such as Python (Engelberth et al., 2012; Ter, 2024; pro, 2024) or a few programming languages (LLM, 2024), which lacks numerous commonly used dependency libraries. The absence of an easy-to-use sandbox necessitates that researchers spend considerable time on installing environments and dependency packages, as well as constructing a distributed sandbox to support multiple programming languages.

Some researchers also combine compiler feedback with other analysis results of generated codes to enhance LLM's performance on more code-related tasks (Lin et al., 2020; Korel, 1990; Lowry and Medlock, 1969; Roziere et al., 2020; Ahmad et al., 2020; Antoy et al., 1990). For instance, some work focuses on developing LLM-based software vulnerability detection approaches by using compiler feedback and traditional code features (Lu et al., 2024; Du et al., 2024b). Some work also aims to enhance LLM's performance in generating more diverse unit test samples (*i.e.,* software test generation) (Gu et al., 2024; Ryan et al., 2024; Chen et al., 2024), generating more comprehensive code specification (*i.e.,* specification generation) (Ma et al., 2024; Zheng et al., 2023; Jin et al., 2024), optimizing the code snippets (*i.e.,* code efficiency optimization) (Gao et al., 2024; Du et al., 2024a), summarizing the code with natural language (*i.e.,* code summary) (Virk et al., 2024; Kumar and Chimalakonda, 2024; Nam et al., 2024), and translating code from one programming language to another programming language (*i.e.,* code translation) (Pan et al., 2024; Yin et al., 2024; Bhattarai et al., 2024). The application of LLMs to code-related tasks necessitates leveraging a vast array of traditional analysis tools from the field of software engineering. Researchers will expend much time and effort on trivial work such as constructing the environment and resolving versioning and dependency issues. An out-of-the-box framework that integrates a multilingual isolated compilation and execution sandbox with multilingual analysis tools remains unexplored.

1.2 Program analysis tools

The analysis of structural information, intermediate variable flow during execution, and resource usage can help researchers comprehensively understand and improve the quality of code (Wang et al., 2024; Ryan et al., 2024). For instance, various tools are employed to obtain code structure information and code smells, such as abstract syntax trees (AST) (*e.g.,* Python's ASTPretty (ast, 2024) and Java's Javalang (jav, 2024)), control flow graphs (CFG) (*e.g.,* C++'s Clang (cla, 2024) and JS's Joern (joe, 2024)), complexity measurement tools (*e.g.,* Python's Radon (rad, 2024) and Java's Pmd (pmd, 2024)), and unit test coverage analysis tools such as Coverage (cov, 2024). Fuzzing test tools employ techniques such as coverage-based feedback-driven testing and fault injection to discover security vulnerabilities in programs (Xia et al., 2024). Additionally, tools that evaluate code efficiency, such as performance profilers (*e.g.,* Line_profile (lin, 2024) and Jprofile (jpr, 2024)), help researchers analyze, optimize, and maintain high-quality programs.

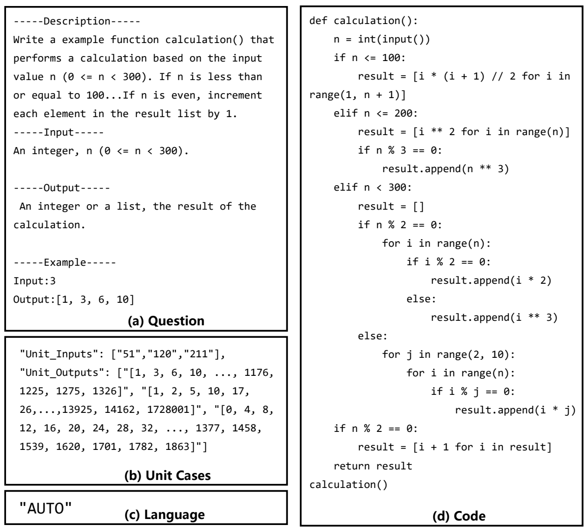
These program analysis tools can also be utilized by LLMs to enhance various downstream code tasks (Du et al., 2024b; Cheng et al., 2024). We have predefined over 40 open-source mainstream tools including five categories for each programming language, redirecting the output of these tools to text. We also design templates that allow users to integrate their own tools with ease. MPLSandbox can unify results from compiler feedback and a variety of powerful analysis tools to streamline LLM workflows in a wide range of code tasks.

2 Case Study on Usage

This section showcases the usage of various analysis function of MPLSandbox through a configuration example shown in Figure 1, where:

- Question field presents the code generation problem posed to the LLM, requiring a function named calculation (n) that performs complex calculations based on different input values of n. The problem description provides the input requirements (an integer, ranging from 0 to 299), output requirements (an integer or a list, depending on the result of the calculation), and an example (when the input is 3, the output should be [1, 3, 6, 10]).
- Code field is the code provided by the LLM in response to the Question field.
- Unit Cases field provides 3 unit test cases, which includes two sub-fields: Unit Inputs and Unit Outputs. Unit Inputs shows three test inputs provided: "51", "120", "211". Unit Outputs shows the corresponding expected output results provided.
- Language field specifies the language of the code, which is set to "AUTO" here, meaning the code language is automatically detected.

Moreover, it still supports specifying docker client instance, docker image, and dockerfile for building custom Docker images. For more details about parameter configuration, please move to: https://github.com/Ablustrund/MPLSandbox



```
-----Description-----
Write a example function calculation() that
performs a calculation based on the input
value n (0 <= n < 300). If n is less than
or equal to 100...If n is even, increment
each element in the result list by 1.
-----Input-----
An integer, n (0 <= n < 300).

-----Output-----
 An integer or a list, the result of the
calculation.

-----Example-----
Input:3
Output:[1, 3, 6, 10]
```

(a) Question

```
"Unit_Inputs": ["51","120","211"],
"Unit_Outputs": ["[1, 3, 6, 10, ..., 1176,
1225, 1275, 1326]", "[1, 2, 5, 10, 17,
26,...,13925, 14162, 1728001]", "[0, 4, 8,
12, 16, 20, 24, 28, 32, ..., 1377, 1458,
1539, 1620, 1701, 1782, 1863]"]
```

(b) Unit Cases

```
"AUTO"
```

(c) Language

```
def calculation():
    n = int(input())
    if n <= 100:
        result = [i * (i + 1) // 2 for i in
range(1, n + 1)]
    elif n <= 200:
        result = [i ** 2 for i in range(n)]
        if n % 3 == 0:
            result.append(n ** 3)
    elif n < 300:
        result = []
        if n % 2 == 0:
            for i in range(n):
                if i % 2 == 0:
                    result.append(i * 2)
                else:
                    result.append(i ** 3)
        else:
            for j in range(2, 10):
                for i in range(n):
                    if i % j == 0:
                        result.append(i * j)
    if n % 2 == 0:
        result = [i + 1 for i in result]
    return result
calculation()
```

(d) Code

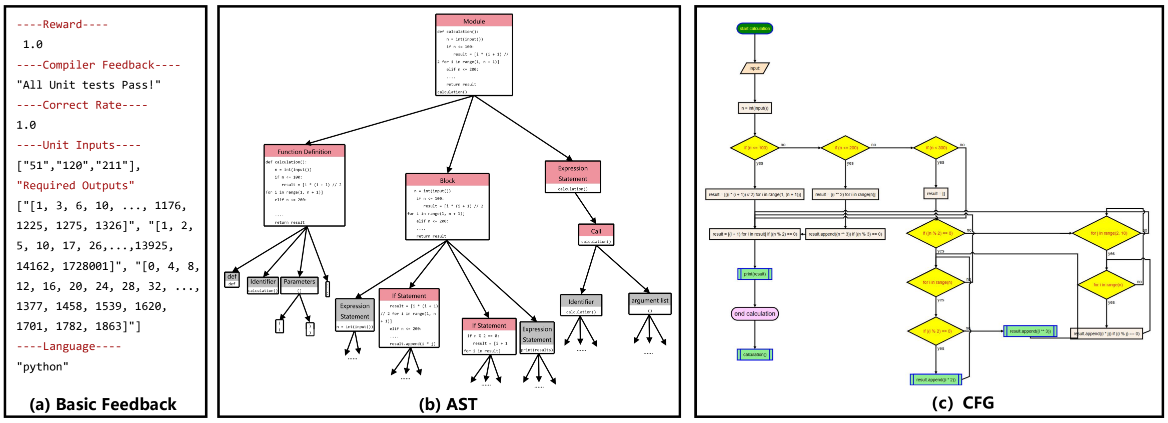
Figure 1: One case used to analyze.

After initializing MPLSandbox as an executor using the configuration, it can be run by calling the run () method. MPLSandbox analyzes the code across five levels by calling the Code Analysis Module and integrates the analysis results through the Information Integration Module. Users can choose the required related information according to their needs. The following sections will conduct corresponding case studies on the example code in the configuration.

2.1 Basic Information Analysis

As shown in Figure 2, Code Basic Analysis returns a Basic Feedback along with Abstract Syntax Tree (AST) and Control Flow Graph (CFG). The basic feedback includes fields such as Reward, Compiler Feedback, Correct Rate, Unit Inputs, Required Outputs and Language. From the compiler feedback, it can be seen that the code has successfully passed all unit tests, achieving a correct rate of 1.0 and a reward of 1.0. These values can serve as signals for downstream analysis or training tasks.

Additionally, MPLSandbox parses the code into the forms of an AST and a CFG. The AST presents the syntactic structure of the code in a tree diagram, where each node represents a syntactic element in the code, such as function definitions, assignment operations, conditional judgments, and loops. This structure helps to understand the logic and hierarchical relationships of the code, facilitating code optimization and error detection. The CFG, on the other hand, graphically displays the execution paths and decision points of the code, including basic blocks (representing a series of consecutive



```
----Reward----
 1.0
----Compiler Feedback----
"All Unit tests Pass!"
----Correct Rate----
1.0
----Unit Inputs----
["51","120","211"],
"Required Outputs"
["[1, 3, 6, 10, ..., 1176,
1225, 1275, 1326]", "[1, 2,
5, 10, 17, 26,...,13925,
14162, 1728001]", "[0, 4, 8,
12, 16, 20, 24, 28, 32, ...,
1377, 1458, 1539, 1620,
1701, 1782, 1863]"]
----Language----
"python"
```

Figure 2: Report of code basic analysis.

instructions) and edges (indicating the direction of control flow), which helps to reveal the execution order of the program and potential branching conditions. The CFG is beneficial for identifying loop dependencies, potential performance bottlenecks, and logical errors in the program.

2.2 Code Smell Analysis & Code Bug Analysis

Code Smell Analysis and Code Bug Analysis identify potential issues or vulnerabilities within the code, reporting the specific line numbers as well as the categories of smells or bugs. To better demonstrate Code Smell Analysis and Code Bug Analysis, we have artificially introduced some code smells and vulnerabilities into the code. The yellow warning boxes in Figure 3 represent the parts of the code where MPLSandbox has detected code smells, which include:

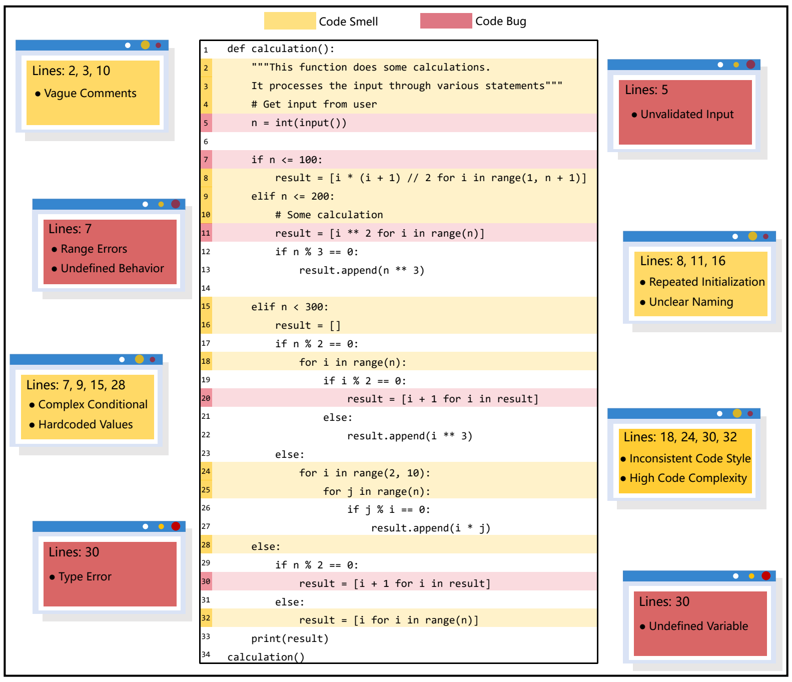
- Line 2, 3, and 10 exhibit the Vague Comments smell due to the docstring and comments being filled with vague and abstract descriptions, not providing concrete information to help understand the actual functionality of the function.

- The Complex Conditional smell on lines 7, 9, 15, 28 is due to the use of multiple if-elif statements within the function, making the conditional logic complex and difficult to trace. The Hardcoded Values smell refers to the values 100, 200, and 300 being hardcoded within the function, requiring changes in multiple places if these values need to be modified.

- The Repeated Initialization smell on lines 8, 11, 16 is due to the variable `result` being initialized in multiple branches, violating the DRY (Don't Repeat Yourself) principle. Unclear Naming refers to the variable `result` not clearly expressing its meaning and a more descriptive name might be better.

- The Inconsistent Code Style smell on lines 18, 24, 30, 32 refers to the inconsistent use of variables `i` and `j` in for loops across different branches, which may cause confusion. The High Code Complexity smell indicates that the function contains multiple conditional branches and nested loops, making it difficult to understand and maintain the code.

It is worth noting that the bad smell determines the overall maintainability of the code, which is often difficult to quantify. Therefore, Code Smell Analysis provides a sub-report specifically for analyzing the maintainability of the code. As shown in Figure 4, the report is divided into three parts: Raw Analysis, Halstead Metrics, and Maintainability Index. The raw analysis provides the distribution of source code, comments, multiline comments, and blank lines, showing a preliminary state of the code. Halstead Metrics offer various indicators of the Halstead volume to quantify the complexity of the code. The Maintainability Index, on the other hand, is a comprehensive calculation of the overall Maintainability Index based on the number of lines of source code, the volume indicator in the Halstead volume, and the cyclomatic complexity of the code, providing a comprehensive assessment of the code's complexity. It's important to note that the Maintainability Index in the report is a standardized result, with its range limited to 0-100. This threshold can be broken down into 0-9 (red), indicating that the code is difficult to maintain. Code within this range may have many issues and requires sig-



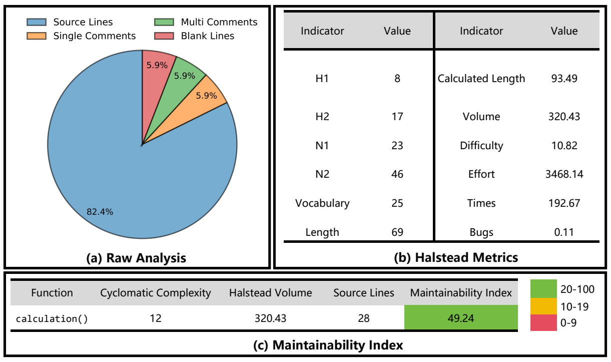
```
1   def calculation():
2       """This function does some calculations.
3       It processes the input through various statements"""
4       # Get input from user
5       n = int(input())
6
7       if n <= 100:
8           result = [i * (i + 1) // 2 for i in range(1, n + 1)]
9       elif n <= 200:
10          # Some calculation
11          result = [i ** 2 for i in range(n)]
12          if n % 3 == 0:
13              result.append(n ** 3)
14
15      elif n < 300:
16          result = []
17          if n % 2 == 0:
18              for i in range(n):
19                  if i % 2 == 0:
20                      result = [i + 1 for i in result]
21                  else:
22                      result.append(i ** 3)
23          else:
24              for i in range(2, 10):
25                  for j in range(n):
26                      if j % i == 0:
27                          result.append(i * j)
28      else:
29          if n % 2 == 0:
30              result = [i + 1 for i in result]
31          else:
32              result = [i for i in range(n)]
33      print(result)
34  calculation()
```

Figure 3: Report of code smell analysis and code bug analysis.

nificant effort to improve its maintainability. 10-19 (yellow) indicates that the code's maintainability is moderate. Although not the worst, code within this range may still require some improvements to enhance its maintainability. 20-100 (green), which is the range where the code falls, indicates that the code has a good structure and clear coding style, making it easy to maintain.

The red warning boxes in Figure 3 represent the parts of the code where MPLSandbox has detected code bugs, which include:

- The Unvalidated Input bug on line 5 is due to user input being used directly in calculations without any validation or restriction.
- The Range Errors bug on line 7 occurs when n=0, causing range(1, n + 1) to raise a ValueError. The Undefined Behavior bug is when n is negative, the behavior of range(n) is undefined.
- The Type Error bug on line 20 occurs when the variable result is empty, executing result = [i + 1 for i in result] will raise a TypeError.
- The occurrence of an Undefined Variable on line 30 is due to the fact that the variable result was not initialized within this conditional branch. Consequently, the result = [i + 1 for i in result] is invalid code.



Indicator	Value	Indicator	Value
H1	8	Calculated Length	93.49
H2	17	Volume	320.43
N1	23	Difficulty	10.82
N2	46	Effort	3468.14
Vocabulary	25	Times	192.67
Length	69	Bugs	0.11

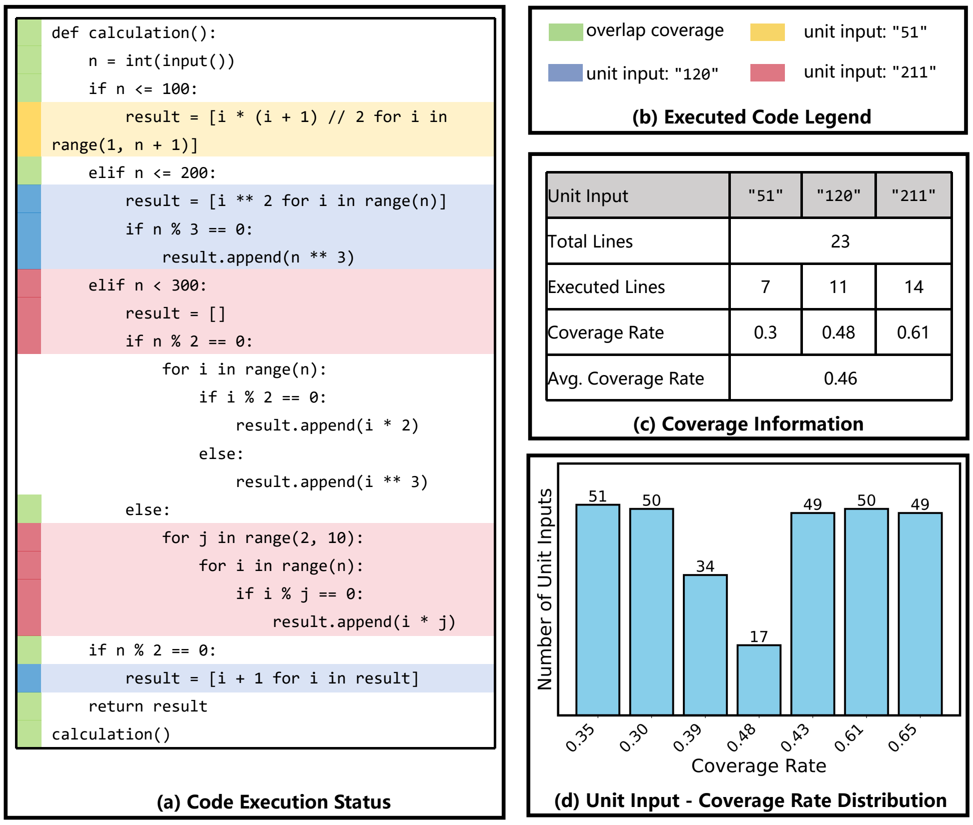
Function	Cyclomatic Complexity	Halstead Volume	Source Lines	Maintainability Index
calculation()	12	320.43	28	49.24

🟩	20-100
🟨	10-19
🟥	0-9

Figure 4: Sub report of code maintainability analysis.

2.3 Unit Test Analysis

Unit Test Analysis returns a comprehensive coverage report for the given unit tests. As shown in Figure 5, green lines represent the overlapping parts of the executed lines for different unit inputs, while yellow, blue, and red lines represent the non-overlapping parts of the executed lines for the test cases "51", "120", and "210", respectively. For the unit input "51", 7 lines of code were executed, with



```
def calculation():
    n = int(input())
    if n <= 100:
        result = [i * (i + 1) // 2 for i in
range(1, n + 1)]
    elif n <= 200:
        result = [i ** 2 for i in range(n)]
        if n % 3 == 0:
            result.append(n ** 3)
    elif n < 300:
        result = []
        if n % 2 == 0:
            for i in range(n):
                if i % 2 == 0:
                    result.append(i * 2)
                else:
                    result.append(i ** 3)
        else:
            for j in range(2, 10):
                for i in range(n):
                    if i % j == 0:
                        result.append(i * j)
    if n % 2 == 0:
        result = [i + 1 for i in result]
    return result
calculation()
```

(a) Code Execution Status

🟩 overlap coverage 🟨 unit input: "51"
🟦 unit input: "120" 🟥 unit input: "211"

(b) Executed Code Legend

Unit Input	"51"	"120"	"211"
Total Lines	23		
Executed Lines	7	11	14
Coverage Rate	0.3	0.48	0.61
Avg. Coverage Rate	0.46		

(c) Coverage Information

Figure 5: Report of unit test analysis.

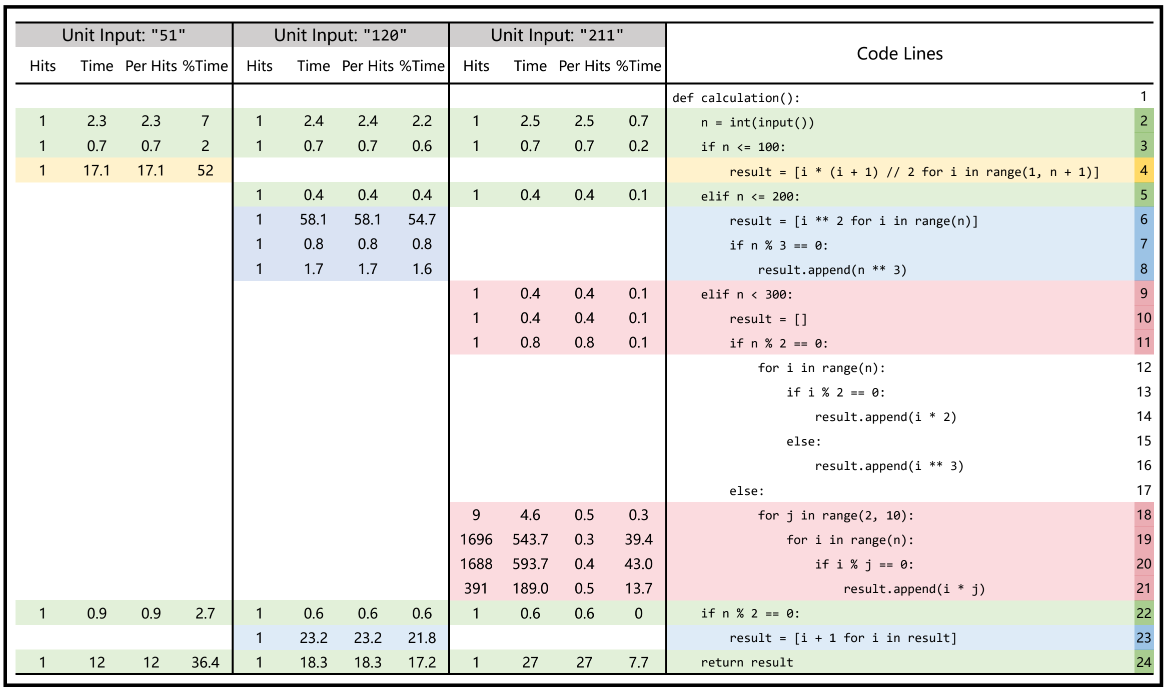
a coverage rate of 0.3. For the unit input "120", 11 lines of code were executed, with a coverage rate of 0.48. For the unit input "211", 14 lines of code were executed, with a coverage rate of 0.61. For a total of 23 lines of code, the overall average coverage rate is 0.46. This indicates that the current test cases do not fully cover the code paths.

Furthermore, Unit Test Analysis has conducted a complete coverage statistics for all test inputs within the given range. It can be observed that within the range of unit input 0 <= n < 300, this set of code has resulted in 7 different coverage possibilities, with the highest being 0.65 and the lowest being 0.35. The distribution of unit inputs across various coverage rates is relatively even. It is evident that after iterating through all possible test inputs, the code coverage remains at a relatively low level, suggesting that the logical framework of the code itself still has significant room for improvement.

2.4 Code Efficiency Evaluation

Code Efficient Evaluation provides an analysis of code execution efficiency for different test cases. Figure 6 reports the Hits (the number of times a code line is executed), Time (the total execution time of the code line in milliseconds), Per Hits (the average time required for each execution of the code line in milliseconds), and %Time (the percentage of the total execution time taken by the execution time of the code line).

As shown in Figure 6, code lines 2, 3, 5, 22 and 24 have common execution records under different test inputs, with some code lines taking a longer execution time under specific inputs. For example, code line 6 takes 58.1 milliseconds to execute under the input "120" because in this case, line 6 is a loop that iterates 120 times. Code line 23 takes 33.2 milliseconds to execute under the input "210" because this line of code contains a loop that iterates based on the variable result , which is strongly related to the input 210. Code lines 12, 13, and 14 have a large number of executions under the input "210" (211 times, 210 times, and 105 times, respectively), because this part involves the processing of a large range loop. Therefore, these perceptions of code line execution efficiency undoubtedly provide very important basis for further performance optimization.



Unit Input: "51"				Unit Input: "120"				Unit Input: "211"				Code Lines	
Hits	Time	Per Hits	%Time	Hits	Time	Per Hits	%Time	Hits	Time	Per Hits	%Time		
												`def calculation():`	1
1	2.3	2.3	7	1	2.4	2.4	2.2	1	2.5	2.5	0.7	`    n = int(input())`	2
1	0.7	0.7	2	1	0.7	0.7	0.6	1	0.7	0.7	0.2	`    if n <= 100:`	3
1	17.1	17.1	52									`        result = [i * (i + 1) // 2 for i in range(1, n + 1)]`	4
				1	0.4	0.4	0.4	1	0.4	0.4	0.1	`    elif n <= 200:`	5
				1	58.1	58.1	54.7					`        result = [i ** 2 for i in range(n)]`	6
				1	0.8	0.8	0.8					`        if n % 3 == 0:`	7
				1	1.7	1.7	1.6					`            result.append(n ** 3)`	8
								1	0.4	0.4	0.1	`    elif n < 300:`	9
								1	0.4	0.4	0.1	`        result = []`	10
								1	0.8	0.8	0.1	`        if n % 2 == 0:`	11
												`            for i in range(n):`	12
												`                if i % 2 == 0:`	13
												`                    result.append(i * 2)`	14
												`                else:`	15
												`                    result.append(i ** 3)`	16
												`        else:`	17
								9	4.6	0.5	0.3	`            for j in range(2, 10):`	18
								1696	543.7	0.3	39.4	`                for i in range(n):`	19
								1688	593.7	0.4	43.0	`                    if i % j == 0:`	20
								391	189.0	0.5	13.7	`                        result.append(i * j)`	21
1	0.9	0.9	2.7	1	0.6	0.6	0.6	1	0.6	0.6	0	`    if n % 2 == 0:`	22
				1	23.2	23.2	21.8					`        result = [i + 1 for i in result]`	23
1	12	12	36.4	1	18.3	18.3	17.2	1	27	27	7.7	`    return result`	24

Figure 6: Report of code efficiency evaluation.

3 Detailed Experimental Setup

3.1 Datasets

To validate the effectiveness of MPLSandbox, we conduct all experiments on the **TACO** dataset (Li et al., 2023), which contains two widely used datasets, *i.e.,* APPS (Hendrycks et al., 2021) and CodeContests (Li et al., 2022), and a portion of the newly crawled data from the contest sites. **APPS** is a code generation benchmark collected from open-access sites, including Codewars, AtCoder, Kattis, and Codeforces. **CodeContests** is a comprehensive collection of competitive programming problems, designed to facilitate research and development in the fields of code synthesis. For the APPS part of TACO, we replace the data with APPS+ (Dou et al., 2024b), which is the curated version of APPS. It excludes instances lacking input, output, or canonical solutions, and eliminates issues such as incomplete code and syntax errors. The final dataset contains 12,000, 1,000, and 1,000 programming problems for training, validation, and testing, respectively. We can train and evaluate LLMs under different programming languages by modifying the system template, as illustrated in Appendix 6.

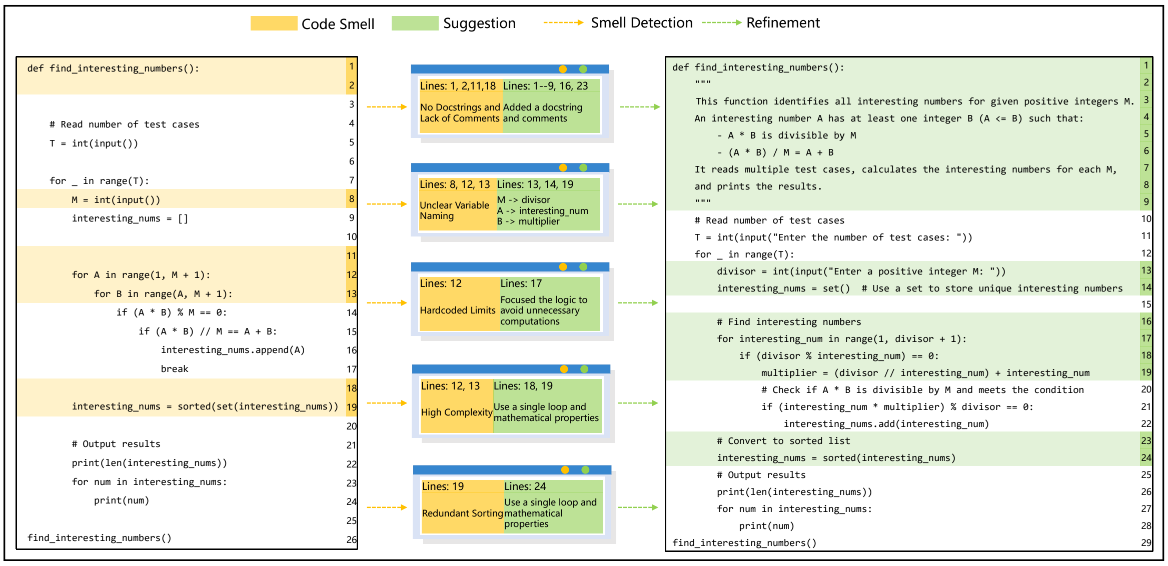
3.2 Foundation moodels

To validate our tool, we integrate it into a wide range of LLMs, including:

- **DeepSeek-Coder-Instruct-6.7B** (Guo et al., 2024), developed by DeepSeek AI, demonstrates state-of-the-art performance among open-source code models across various programming languages. The training corpus for these models comprises an impressive 2 trillion tokens, combining code and natural language texts. Each model is trained to utilize a window size of 16K.

- **DeepSeek-Coder-V2-Lite-Instruct-16B** (Zhu et al., 2024) is an open-source Mixture-of-Experts (MoE) code language model that achieves performance comparable to GPT-4 Turbo in code-specific tasks. It has 2.4 billion active parameters. The model is further pre-trained from an intermediate checkpoint of DeepSeek-V2 with an additional 6 trillion tokens. DeepSeek-Coder-V2 expands its support for programming languages from 86 to 338, while extending the context length from 16K to 128K.

- **Qwen2.5-Coder-1.5B-Instruct** (Team, 2024) and **Qwen2.5-Coder-7B-Instruct** (Team, 2024) are members of the Qwen2.5 series. They are built on the Qwen2.5 architecture and have been further pre-trained on an extensive dataset exceeding 5.5 trillion tokens, which includes source code, text-code grounding data, and synthetic data. These two LLMs



```
def find_interesting_numbers():                          1
                                                         2
                                                         3
    # Read number of test cases                          4
    T = int(input())                                     5
                                                         6
    for _ in range(T):                                   7
        M = int(input())                                 8
        interesting_nums = []                            9
                                                         10
                                                         11
        for A in range(1, M + 1):                        12
            for B in range(A, M + 1):                    13
                if (A * B) % M == 0:                     14
                    if (A * B) // M == A + B:            15
                        interesting_nums.append(A)       16
                        break                            17
                                                         18
        interesting_nums = sorted(set(interesting_nums)) 19
                                                         20
        # Output results                                 21
        print(len(interesting_nums))                     22
        for num in interesting_nums:                     23
            print(num)                                   24
                                                         25
find_interesting_numbers()                               26
```

Lines: 1, 2,11,18	Lines: 1--9, 16, 23
No Docstrings and Lack of Comments	Added a docstring and comments
Lines: 8, 12, 13	Lines: 13, 14, 19
Unclear Variable Naming	M -> divisor A -> interesting_num B -> multiplier
Lines: 12	Lines: 17
Hardcoded Limits	Focused the logic to avoid unnecessary computations
Lines: 12, 13	Lines: 18, 19
High Complexity	Use a single loop and mathematical properties
Lines: 19	Lines: 24
Redundant Sorting	Use a single loop and mathematical properties

```
def find_interesting_numbers():                                                           1
    """                                                                                   2
    This function identifies all interesting numbers for given positive integers M.       3
    An interesting number A has at least one integer B (A <= B) such that:                4
        - A * B is divisible by M                                                         5
        - (A * B) / M = A + B                                                             6
    It reads multiple test cases, calculates the interesting numbers for each M,          7
    and prints the results.                                                               8
    """                                                                                   9
    # Read number of test cases                                                           10
    T = int(input("Enter the number of test cases: "))                                    11
    for _ in range(T):                                                                    12
        divisor = int(input("Enter a positive integer M: "))                              13
        interesting_nums = set()  # Use a set to store unique interesting numbers         14
                                                                                          15
        # Find interesting numbers                                                        16
        for interesting_num in range(1, divisor + 1):                                     17
            if (divisor % interesting_num) == 0:                                          18
                multiplier = (divisor // interesting_num) + interesting_num               19
                # Check if A * B is divisible by M and meets the condition                20
                if (interesting_num * multiplier) % divisor == 0:                         21
                    interesting_nums.add(interesting_num)                                 22
        # Convert to sorted list                                                          23
        interesting_nums = sorted(interesting_nums)                                       24
        # Output results                                                                  25
        print(len(interesting_nums))                                                      26
        for num in interesting_nums:                                                      27
            print(num)                                                                    28
find_interesting_numbers()                                                                29
```

Figure 7: Self-refinement process through MPLSandbox.

support up to 128K tokens of context and cover 92 programming languages.

- **Codestral-v0.1-22B** (mis, 2024) is trained by Mistral AI. This model is trained on a diverse dataset encompassing over 80 programming languages, including popular ones like Python, Java, C, C++, JavaScript, and Bash. It also excels in more specialized languages such as Swift and Fortran. This model supports up to 32K context window.

- **Llama-3.1-Instruct-70B** (Dubey et al., 2024) is developed by Meta AI. It is pre-trained and instruction-tuned generative models that handle text input and output. The Llama 3.1 instruction-tuned models are specifically optimized for multilingual dialogue applications, surpassing many open-source and proprietary chat models on standard industry benchmarks. The context window of this model is 128K.

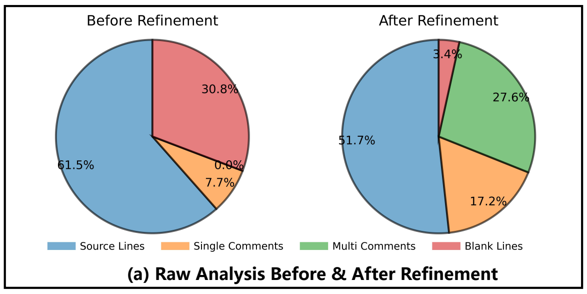
- **GPT-4o** (OpenAI, 2023) is an advanced iteration of OpenAI's language model, designed to offer more refined and contextually aware responses. GPT-4o is exceptionally versatile, and capable of excelling in a wide range of applications from creative writing to technical problem-solving. The context window and maximum output tokens are 128K and 16,384, respectively.

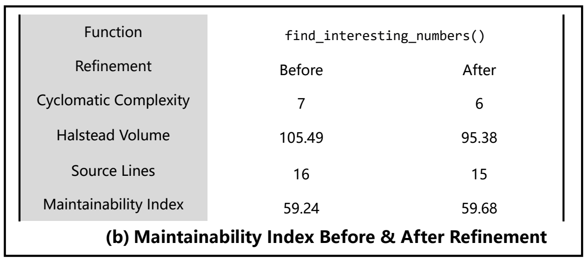
3.3 Implementation Details

First, we utilize MPLSandbox as a verifier to verify LLM-generated code at inference time. We employed the Best-of-N (BoN) strategy to sample LLM's responses multiple times for the same programming problem, and then evaluate the code using Pass@k metric (Chen et al., 2021). In the Pass@1 and Pass@10 settings, the sample temperature is set to 0.2 and 0.8, respectively. All inference experiments are conducted on a single node equipped with eight NVIDIA A100-80G GPUs.

Second, we use the compiler feedback provided by MPLSandbox as a supervised signal to optimize LLMs through reinforcement learning. We use DeepSeek-Coder-Instruct-6.7B (Guo et al., 2024) as the foundation LLM. In each programming language experiment, we modify system prompt templates to enable LLM to solve problems using the according programming language. All training experiments are conducted on 16 training nodes, totaling 128 NVIDIA A100-80G GPUs, and two MPLSandbox nodes. The global batch size is set to 512. The system prompt template used for constructing multi-programming language instructions is illustrated in Appendix 6. The global training step is set to 8000, with a 0.1 ratio of warmup. We report the accuracy point on the test set using the checkpoint at which the model achieves its best performance on the validation set. The learning rate for the policy model and the critic model is $5e^{-7}$ and $1.5e^{-6}$, respectively. For each example, we collect a 16 roll-out code using nucleus sampling. The sampling temperature is set to 0.8, top-p is set to 0.9, and the maximum output token length is set to 2048. The token-level KL penalty coefficient β is set to 0.02, with a clip value of 0.8. In the decoding phase, the temperature and top-p are set to 0.2 and 0.95, respectively.

Finally, through self-reflection and self-correction, we utilize LLMs to correct their generated errors, and improve the quality of codes using the results of analysis tools. The system prompt templates are shown in Appendix 6. The temperature is set to 0.8.





Function	find_interesting_numbers()	
Refinement	Before	After
Cyclomatic Complexity	7	6
Halstead Volume	105.49	95.38
Source Lines	16	15
Maintainability Index	59.24	59.68

(b) Maintainability Index Before & After Refinement

Figure 8: Code maintainability results before and after self-refinement.

4 Reinforcement learning from compiler feedback

Following prior work (Dou et al., 2024b; Liu et al., 2023) in code generation, we use Proximal Policy Optimization (PPO) algorithm (Schulman et al., 2017) to optimize the policy model π_θ with parameter θ. Consider a programming problem x, the policy model generates the response $\hat{y}$ according to x. The reward model provides the reward r according to the correctness of the code snippet τ extracted from response $\hat{y}$, where we use the same setting as previous approaches (Le et al., 2022; Shojaee et al., 2023; Dou et al., 2024b) as follows:

$$r(x,\tau) = \begin{cases} +1, & \text{if } \tau \text{ passed all unit tests} \\ -0.3, & \text{if } \tau \text{ failed any unit test} \\ -0.6, & \text{if } \tau \text{ happened runtime error} \\ -1, & \text{if } \tau \text{ happened compile error.} \end{cases} \quad (1)$$

The policy model π_θ can be optimized by maximizing the objection function as follows:

$$\text{Objective}(\theta) = E_{(x,\hat{y})\sim D_{\pi_\theta}}[r(x,\tau) - \beta \log(\pi_\theta(\hat{y}|x)/\pi^{\text{ref}}(\hat{y}|x))] \quad (2)$$

where π^{ref} is the reference model in PPO, which is initialized with the parameters of the initial policy model and kept frozen throughout the training process.

5 Additional Experiments

5.1 Case study on self-optimization

We utilize a instance from the test set to illustrates the process of self-refinement, as shown in Figure 7. It begins with Code Smell Analysis for smell detection, identifying code issues such as No Docstrings and Lack of Comments, Unclear Variable Naming, Hardcoded Limits, High Complexity, and Redundant Sorting. Subsequently, the built-in LLM-based system proposes corresponding improvements for these suggestions. Finally, these suggestions are incorporated into system prompts to achieve self-refinement.

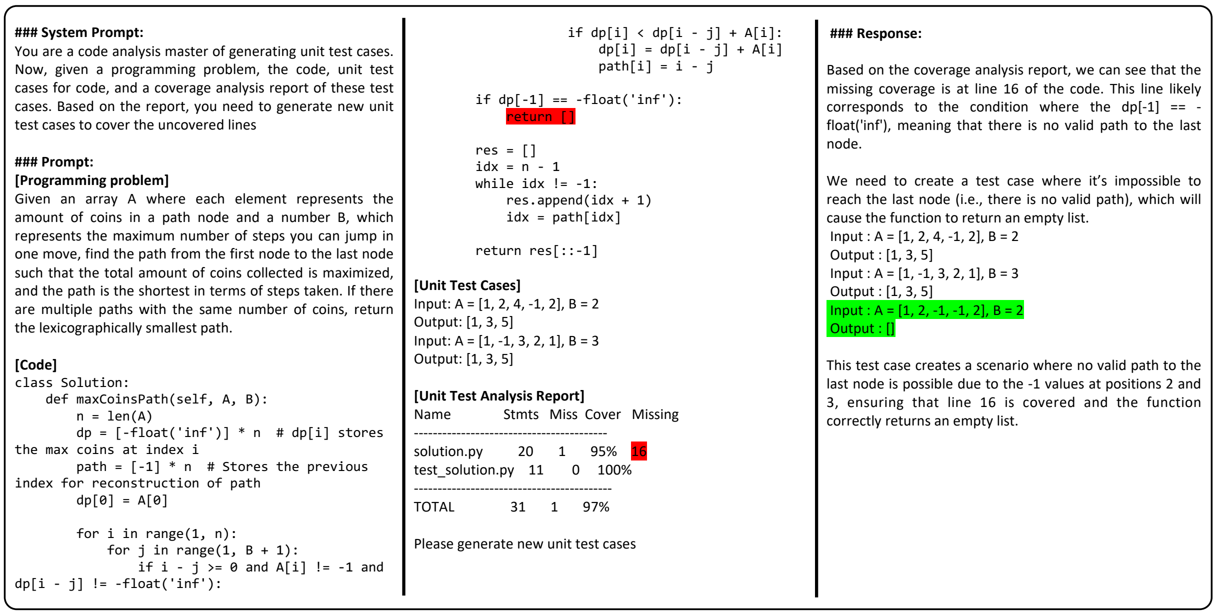
Figure 8 shows the results of analyzing certain metrics of the code before and after refinement using maintainability analysis, quantifying the effectiveness of the refinement. As can be seen from Figure 8(a), there is a significant increase in the ratio of single-line and multi-line comments in the code compared to before, demonstrating it enhances the interpretability of the code. Furthermore, as shown in Figure 8(b), after refinement, the overall cyclomatic complexity and Halstead Volume of the code have decreased, resulting in an increase in the Maintainability Index, further showing the positive feedback of the entire refinement on code maintainability.

5.2 Additional three application scenarios

To further demonstrate the effectiveness and efficiency of our tool, we present it in three downstream code-related tasks. Our tool can significantly streamline LLM workflows in these tasks, greatly reducing developers' time and costs.

5.2.1 Unit test generation

In this experiment, we show how to generate more effective unit test samples through MPLSandbox to evaluate the user's code. User-provided unit test samples may not cover the entire code, *i.e.,* the unit test coverage is not 100%. This means there could be errors in code snippets that have not been verified. Following prior work (Jiang et al., 2024b), we can utilize the unit test analysis tool to identify the uncovered code snippets. We then integrate these snippets into the prompt to enable the LLM



### System Prompt:
You are a code analysis master of generating unit test cases. Now, given a programming problem, the code, unit test cases for code, and a coverage analysis report of these test cases. Based on the report, you need to generate new unit test cases to cover the uncovered lines

### Prompt:
[Programming problem]
Given an array A where each element represents the amount of coins in a path node and a number B, which represents the maximum number of steps you can jump in one move, find the path from the first node to the last node such that the total amount of coins collected is maximized, and the path is the shortest in terms of steps taken. If there are multiple paths with the same number of coins, return the lexicographically smallest path.

[Code]
```
class Solution:
    def maxCoinsPath(self, A, B):
        n = len(A)
        dp = [-float('inf')] * n  # dp[i] stores
the max coins at index i
        path = [-1] * n  # Stores the previous
index for reconstruction of path
        dp[0] = A[0]

        for i in range(1, n):
            for j in range(1, B + 1):
                if i - j >= 0 and A[i] != -1 and
dp[i - j] != -float('inf'):
                    if dp[i] < dp[i - j] + A[i]:
                        dp[i] = dp[i - j] + A[i]
                        path[i] = i - j

        if dp[-1] == -float('inf'):
            return []

        res = []
        idx = n - 1
        while idx != -1:
            res.append(idx + 1)
            idx = path[idx]

        return res[::-1]
```

[Unit Test Cases]
Input: A = [1, 2, 4, -1, 2], B = 2
Output: [1, 3, 5]
Input: A = [1, -1, 3, 2, 1], B = 3
Output: [1, 3, 5]

[Unit Test Analysis Report]
```
Name              Stmts  Miss  Cover  Missing
-----------------------------------------
solution.py         20     1    95%    16
test_solution.py    11     0   100%
-----------------------------------------
TOTAL               31     1    97%
```

Please generate new unit test cases

### Response:

Based on the coverage analysis report, we can see that the missing coverage is at line 16 of the code. This line likely corresponds to the condition where the dp[-1] == -float('inf'), meaning that there is no valid path to the last node.

We need to create a test case where it's impossible to reach the last node (i.e., there is no valid path), which will cause the function to return an empty list.
Input : A = [1, 2, 4, -1, 2], B = 2
Output : [1, 3, 5]
Input : A = [1, -1, 3, 2, 1], B = 3
Output : [1, 3, 5]
Input : A = [1, 2, -1, -1, 2], B = 2
Output : []

This test case creates a scenario where no valid path to the last node is possible due to the -1 values at positions 2 and 3, ensuring that line 16 is covered and the function correctly returns an empty list.

Figure 9: One instance of Unit test generation through MPLSandbox.

to generate additional unit test samples to verify these uncovered snippets. Through MPLSandbox, unit test generation is simple and effective. Users can modify the system prompt to achieve this task easily. An instance is illustrated in Figure 9.

5.2.2 Code translation

Second, we showcase translating code from one programming language to another. Large language models have been widely used in code translation. Prior work (Tao et al., 2024; Luo et al., 2024) has demonstrated that including structural information of the code, such as unit test cases and the control flow graph, into the system prompt can help LLMs better understand the code, significantly increasing the success rate of code translation.

With MPLSandbox, users can effortlessly integrate various code-related information from traditional analysis tools into the system prompt, to enhance the performance of code translation. An instance of translating code from Python into Java is shown in Figure 10.

5.2.3 Vulnerability location

LLMs can assist developers in identifying potential vulnerabilities in their code. When combined with traditional vulnerability analysis tools, LLMs can significantly enhance the performance of vulnerability detection and localization (Lu et al., 2024; Akuthota et al., 2023). MPLSandbox can also efficiently deploy LLM-based vulnerability detection workflows, minimizing development costs. An instance of detecting vulnerabilities in the code through MPLSandbox is illustrated in Figure 11.

These results all indicate that PLSandbox can enhance the efficiency of deploying LLMs in various code-related tasks.

6 Prompts in Detail

The system prompt for multi-programming language is as follows:

You are a master programmer. Now given a programming problem, you need to analyze it carefully and answer this programming problem directly in ***{lang}*** *programming language.*
Note that generated code will accept a unit test case through standard input (stdin) from the terminal. And after printing the output result, the program will terminate
### Prompt:
[Programming problem]
{programming problem}
Solve this problem directly in ***{lang}****. The code needs to be surrounded by backquotes (i.e., "'code"')*
### Response:

The system prompts for correcting and refining code are as follows:

You are a master of program correction. Now given a programming problem, an incorrect solution to this problem, and the compiler error message. You need to carefully analyze and correct the incorrect code
### Prompt:
[Programming problem]
{programming problem}
[Incorrect code]
{incorrect code}
[Compiler error message]
{error message}
Please correct this problem, and the new code needs to be surrounded by backquotes (i.e., "'code"')
### Response:

You are a master of program refinement. Now given a programming problem, the correct solution, and some suggestions for improvement including readability, complexity and code Specification. You need to refine the code according to these suggestions
### Prompt:
[Programming problem]
{programming problem}
[Original code]
{code}
[Suggestions through code smell]
{analysis results} # including code style and specifications (*e.g.,* long method, long parameter list and tight coupling)
Please first summarize recommendations for refinement from these analysis results, and then refine this code. The new code needs to be surrounded by backquotes (i.e., "'code"')
### Response:

The prompts for unit test generation, code translation, and vulnerability location are shown in Figure 9, 10, and 11.

References

2023. gpt-3.5-turbo. https://platform.openai.com/docs/models/gpt-3-5.

2023. Llama-2. https://ai.meta.com/research/publications/llama-2-open-foundation-and-fine-tuned-chat-models.

2024. astpretty. https://pypi.org/project/astpretty/.

2024. clang. https://clang.llvm.org/.

2024. coverage. https://coverage.readthedocs.io/en/7.6.4/.

2024. javalang. https://github.com/c2nes/javalang.

2024. joern. https://github.com/joernio/joern.

2024. jprofiler. https://www.ej-technologies.com/jprofiler.

2024. lineprofiler. https://github.com/pyutils/line_profiler.

2024. Llama-3. https://ai.meta.com/blog/meta-llama-3.

2024. Llmsandbox. https://hackernoon.com/introducing-llm-sandbox-securely-execute-llm-generated-code-with-ease.

2024. mistralai.

2024. pmd. https://pmd.github.io/.

2024. Promptfoo. https://www.promptfoo.dev/docs/guides/sandboxed-code-evals/.

2024. radon. https://pypi.org/project/radon/.

2024. Terrarium. https://github.com/cohere-ai/cohere-terrarium.

Marah Abdin, Sam Ade Jacobs, Ammar Ahmad Awan, Jyoti Aneja, Ahmed Awadallah, Hany Awadalla, Nguyen Bach, Amit Bahree, Arash Bakhtiari, Harkirat Behl, et al. 2024. Phi-3 technical report: A highly capable language model locally on your phone. *arXiv preprint arXiv:2404.14219*.

Wasi Uddin Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2020. A transformer-based approach for source code summarization. *arXiv preprint arXiv:2005.00653*.

Vishwanath Akuthota, Raghunandan Kasula, Sabiha Tasnim Sumona, Masud Mohiuddin, Md Tanzim Reza, and Md Mizanur Rahman. 2023. Vulnerability detection and monitoring using llm. In *2023 IEEE 9th International Women in Engineering (WIE) Conference on Electrical and Computer Engineering (WIECON-ECE)*, pages 309–314. IEEE.

Loubna Ben Allal, Raymond Li, Denis Kocetkov, Chenghao Mou, Christopher Akiki, Carlos Munoz Ferrandis, Niklas Muennighoff, Mayank Mishra, Alex Gu, Manan Dey, et al. 2023. Santacoder: don't reach for the stars! *arXiv preprint arXiv:2301.03988*.

AI Anthropic. 2024. The claude 3 model family: Opus, sonnet, haiku. *Claude-3 Model Card*, 1.

### System Prompt:
You are a master of code translation. Now, given a programming problem, source Python code, unit test cases for the code, and the control flow graph for code. Based on these information, you need to translate the source code from Python to Java

### Prompt:
[Programming problem]
Design a class to find the k-th largest element in a stream. You should implement the class KthLargest which adds numbers to the stream and finds the k-th largest element.

[Source Python code]
```
import heapq
def __init__(self, k: int, nums: list[int]):
    self.k = k
    self.min_heap = nums
    heapq.heapify(self.min_heap)
    while len(self.min_heap) > k:
        heapq.heappop(self.min_heap)
def add(self, val: int) -> int:
    heapq.heappush(self.min_heap, val)
    if len(self.min_heap) > self.k:
        heapq.heappop(self.min_heap)
    return self.min_heap[0]
```

[Unit test cases]
Input: ["KthLargest", "add", "add", "add", "add", "add"]
[[3, [4, 5, 8, 2]], [3], [5], [10], [9], [4]]
Output: [null, 4, 5, 5, 8, 8]

[Control flow graph]
```
op2=>operation: import heapq
st5=>start: start __init__
io7=>inputoutput: input: self, k, nums
op10=>operation: self.k = k
op12=>operation: self.min_heap = nums
sub14=>subroutine: heapq.heapify(self.min_heap)
cond17=>operation: heapq.heappop(self.min_heap)
while  (len(self.min_heap) > k)
e29=>end: end __init__
st33=>start: start add
io35=>inputoutput: input: self, val
sub38=>subroutine: heapq.heappush(self.min_heap,
val)
cond41=>operation: heapq.heappop(self.min_heap)
if  (len(self.min_heap) > self.k)
io54=>inputoutput: output:  self.min_heap[0]
e52=>end: end function return
op2->st5
st5->io7
io7->op10
op10->op12
op12->sub14
sub14->cond17
cond17->e29
e29->st33
st33->io35
io35->sub38
sub38->cond41
cond41->io54
io54->e52
```

Please translate the code from Python to Java, and the new code needs to be surrounded by backquotes (i.e., ```code```)

### Response:
…
```java
import java.util.PriorityQueue;
public class KthLargest {
   private int k;
   private PriorityQueue<Integer> minHeap;
   public KthLargest(int k, int[] nums) {
      this.k = k;
      minHeap = new PriorityQueue<>();
      // Add all elements from nums to the heap
      for (int num : nums) {
         minHeap.offer(num);
      }
      // Maintain only the k largest elements in the heap
      while (minHeap.size() > k) {
         minHeap.poll();
      }
   }
   public int add(int val) {
      minHeap.offer(val);
      if (minHeap.size() > k) {
         minHeap.poll();  // Remove the smallest element to
keep only k largest
      }
      return minHeap.peek();  // The root of the heap is the
k-th largest element
   }
}
```
…

Figure 10: One instance of Code translation through MPLSandbox.

### System Prompt:
You are a master of vulnerability localization. Now, given a programming problem, the code, the analysis report based on traditional vulnerability detection tools. Based on these information, you need to detect and localize the vulnerability in the code

### Prompt:
[Programming problem]
You are given a directory path where several text files are stored. Each text file contains a list of integers, one per line. Your task is to write a function that reads all the files in the directory, extracts the integers from each file, and returns the maximum integer value found across all files.

You can assume that the directory and files are valid and accessible, and all files contain valid integers.

[Code]
```
import os
import subprocess

class Solution:
    def find_max_in_directory(self,
directory_path: str) -> int:
        max_value = -float('inf')

        command = f"ls {directory_path}"
        result = subprocess.run(command,
shell=True, capture_output=True, text=True)
 # Get list of files in the directory
        files = result.stdout.splitlines()

        for file in files:
            file_path =
os.path.join(directory_path, file)

            # Another subprocess command to
read the contents of each file
            command = f"cat {file_path}"
            result = subprocess.run(command,
shell=True, capture_output=True, text=True)

            # Process file contents and find
the maximum value
            for line in
result.stdout.splitlines():
                num = int(line)
                max_value = max(max_value, num)
        return max_value
```

[Analysis report]
>>Issue: [B404:blacklist] Consider possible security implications associated with the subprocess module.
Severity: Low   Confidence: High
CWE: CWE-78
Location: ./solution.py:2:0
>> Issue: [B602:subprocess_popen_with_shell_equals_true] subprocess call with shell=True identified, security issue.
Severity: High   Confidence: High
CWE: CWE-78
Location: ./solution.py:9:17
Location: ./solution.py:18:21

Please detect and localize the vulnerability in the code

### Response:
Based on the analysis report and the code provided, there are two high-severity vulnerabilities identified, specifically in the use of subprocess.run() with shell=True. This opens the code to potential command injection attacks (CWE-78), where an attacker could inject malicious shell commands through user input.
1.Vulnerability 1: Command Injection in ls Command (Line 9)
Issue: The directory_path parameter is directly embedded in the shell command without any sanitization. If an attacker controls the value of directory_path, they can inject arbitrary shell commands. For example, setting directory_path = "valid_dir; rm -rf /" could cause dangerous commands to be executed on the system.
Location: Line 9, subprocess.run(command, shell=True, ...)
2. Vulnerability 2: Command Injection in cat Command (Line 18)
Issue: Similarly, file_path is passed directly into a shell command without proper escaping or validation. If the file name contains malicious content (like file.txt; rm -rf /), this could also lead to arbitrary command execution.
Location: Line 18, subprocess.run(command, shell=True, ...)

Figure 11: One instance of vulnerability location through MPLSandbox.

S Antoy, Paola Forcheri, and Maria Teresa Molfino. 1990. Specification-based code generation. In *Twenty-Third Annual Hawaii International Conference on System Sciences*, volume 2, pages 165–173. IEEE Computer Society.

Jinze Bai, Shuai Bai, Yunfei Chu, Zeyu Cui, Kai Dang, Xiaodong Deng, Yang Fan, Wenbin Ge, Yu Han, Fei Huang, et al. 2023. Qwen technical report. *arXiv preprint arXiv:2309.16609*.

Manish Bhattarai, Javier E Santos, Shawn Jones, Ayan Biswas, Boian Alexandrov, and Daniel O'Malley. 2024. Enhancing code translation in language models with few-shot learning via retrieval-augmented generation. *arXiv preprint arXiv:2407.19619*.

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

Yinghao Chen, Zehao Hu, Chen Zhi, Junxiao Han, Shuiguang Deng, and Jianwei Yin. 2024. Chatunitest: A framework for llm-based test generation. In *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering*, pages 572–576.

Yiran Cheng, Lwin Khin Shar, Ting Zhang, Shouguo Yang, Chaopeng Dong, David Lo, Shichao Lv, Zhiqiang Shi, and Limin Sun. 2024. Llm-enhanced static analysis for precise identification of vulnerable oss versions. *arXiv preprint arXiv:2408.07321*.

DeepSeek-AI. 2024. Deepseek llm: Scaling open-source language models with longtermism. *arXiv preprint arXiv:2401.02954*.

Paul Denny, Viraj Kumar, and Nasser Giacaman. 2023. Conversing with copilot: Exploring prompt engineering for solving cs1 problems using natural language. In *Proceedings of the 54th ACM Technical Symposium on Computer Science Education V. 1*, pages 1136–1142.

Shihan Dou, Haoxiang Jia, Shenxi Wu, Huiyuan Zheng, Weikang Zhou, Muling Wu, Mingxu Chai, Jessica Fan, Caishuang Huang, Yunbo Tao, et al. 2024a. What's wrong with your code generated by large language models? an extensive study. *arXiv preprint arXiv:2407.06153*.

Shihan Dou, Yan Liu, Haoxiang Jia, Limao Xiong, Enyu Zhou, Junjie Shan, Caishuang Huang, Wei Shen, Xiaoran Fan, Zhiheng Xi, et al. 2024b. Stepcoder: Improve code generation with reinforcement learning from compiler feedback. *arXiv preprint arXiv:2402.01391*.

Mingzhe Du, Anh Tuan Luu, Bin Ji, and See-Kiong Ng. 2024a. Mercury: An efficiency benchmark for llm code synthesis. *arXiv preprint arXiv:2402.07844*.

Xiaohu Du, Ming Wen, Jiahao Zhu, Zifan Xie, Bin Ji, Huijun Liu, Xuanhua Shi, and Hai Jin. 2024b. Generalization-enhanced code vulnerability detection via multi-task instruction fine-tuning. *arXiv preprint arXiv:2406.03718*.

Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Amy Yang, Angela Fan, et al. 2024. The llama 3 herd of models. *arXiv preprint arXiv:2407.21783*.

Markus Engelberth, Jan Göbel, Christian Schönbein, and Felix C Freiling. 2012. Pybox-a python sandbox.

Shuzheng Gao, Cuiyun Gao, Wenchao Gu, and Michael Lyu. 2024. Search-based llms for code optimization. *arXiv preprint arXiv:2408.12159*.

Tal Garfinkel, Mendel Rosenblum, et al. 2003. A virtual machine introspection based architecture for intrusion detection. In *Ndss*, volume 3, pages 191–206. San Diega, CA.

Siqi Gu, Chunrong Fang, Quanjun Zhang, Fangyuan Tian, and Zhenyu Chen. 2024. Testart: Improving llm-based unit test via co-evolution of automated generation and repair iteration. *arXiv preprint arXiv:2408.03095*.

Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, Y. Wu, Y. K. Li, Fuli Luo, Yingfei Xiong, and Wenfeng Liang. 2024. Deepseek-coder: When the large language model meets programming – the rise of code intelligence.

Dan Hendrycks, Steven Basart, Saurav Kadavath, Mantas Mazeika, Akul Arora, Ethan Guo, Collin Burns, Samir Puranik, Horace He, Dawn Song, and Jacob Steinhardt. 2021. Measuring coding challenge competence with APPS. In *Proceedings of the Neural Information Processing Systems Track on Datasets and Benchmarks 1, NeurIPS Datasets and Benchmarks 2021, December 2021, virtual*.

Albert Q Jiang, Alexandre Sablayrolles, Arthur Mensch, Chris Bamford, Devendra Singh Chaplot, Diego de las Casas, Florian Bressand, Gianna Lengyel, Guillaume Lample, Lucile Saulnier, et al. 2023. Mistral 7b. *arXiv preprint arXiv:2310.06825*.

Nan Jiang, Xiaopeng Li, Shiqi Wang, Qiang Zhou, Soneya Binta Hossain, Baishakhi Ray, Varun Kumar, Xiaofei Ma, and Anoop Deoras. 2024a. Training llms to better self-debug and explain code. *arXiv preprint arXiv:2405.18649*.

Zongze Jiang, Ming Wen, Jialun Cao, Xuanhua Shi, and Hai Jin. 2024b. Towards understanding the effectiveness of large language models on directed test input generation. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, pages 1408–1420.

Haolin Jin, Linghan Huang, Haipeng Cai, Jun Yan, Bo Li, and Huaming Chen. 2024. From llms to llm-based agents for software engineering: A survey of current, challenges and future. *arXiv preprint arXiv:2408.02479*.

Bogdan Korel. 1990. Automated software test data generation. *IEEE Transactions on software engineering*, 16(8):870–879.

Jahnavi Kumar and Sridhar Chimalakonda. 2024. Code summarization without direct access to code-towards exploring federated llms for software engineering. In *Proceedings of the 28th International Conference on Evaluation and Assessment in Software Engineering*, pages 100–109.

Hung Le, Hailin Chen, Amrita Saha, Akash Gokul, Doyen Sahoo, and Shafiq Joty. 2023. Codechain: Towards modular code generation through chain of self-revisions with representative sub-modules. In *The Twelfth International Conference on Learning Representations*.

Hung Le, Yue Wang, Akhilesh Deepak Gotmare, Silvio Savarese, and Steven Chu Hong Hoi. 2022. Coderl: Mastering code generation through pretrained models and deep reinforcement learning. *Advances in Neural Information Processing Systems*, 35:21314–21328.

Rongao Li, Jie Fu, Bo-Wen Zhang, Tao Huang, Zhihong Sun, Chen Lyu, Guang Liu, Zhi Jin, and Ge Li. 2023. Taco: Topics in algorithmic code generation dataset. *arXiv preprint arXiv:2312.14852*.

Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, et al. 2022. Competition-level code generation with alphacode. *Science*, 378(6624):1092–1097.

Zhenkai Liang, VN Venkatakrishnan, and R Sekar. 2003. Isolated program execution: An application transparent approach for executing untrusted programs. In *19th Annual Computer Security Applications Conference, 2003. Proceedings.*, pages 182–191. IEEE.

Guanjun Lin, Sheng Wen, Qing-Long Han, Jun Zhang, and Yang Xiang. 2020. Software vulnerability detection using deep neural networks: a survey. *Proceedings of the IEEE*, 108(10):1825–1848.

Jiate Liu, Yiqin Zhu, Kaiwen Xiao, Qiang Fu, Xiao Han, Wei Yang, and Deheng Ye. 2023. Rltf: Reinforcement learning from unit test feedback. *arXiv preprint arXiv:2307.04349*.

Edward S Lowry and Cleburne W Medlock. 1969. Object code optimization. *Communications of the ACM*, 12(1):13–22.

Guilong Lu, Xiaolin Ju, Xiang Chen, Wenlong Pei, and Zhilong Cai. 2024. Grace: Empowering llm-based software vulnerability detection with graph structure and in-context learning. *Journal of Systems and Software*, 212:112031.

Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin Clement, Dawn Drain, Daxin Jiang, Duyu Tang, et al. 2021. Codexglue: A machine learning benchmark dataset for code understanding and generation. *arXiv preprint arXiv:2102.04664*.

Yang Luo, Richard Yu, Fajun Zhang, Ling Liang, and Yongqiang Xiong. 2024. Bridging gaps in llm code translation: Reducing errors with call graphs and bridged debuggers. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, pages 2448–2449.

Lezhi Ma, Shangqing Liu, Yi Li, Xiaofei Xie, and Lei Bu. 2024. Specgen: Automated generation of formal program specifications via large language models. *arXiv preprint arXiv:2401.08807*.

Daye Nam, Andrew Macvean, Vincent Hellendoorn, Bogdan Vasilescu, and Brad Myers. 2024. Using an llm to help with code understanding. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, pages 1–13.

OpenAI. 2023. Gpt-4 technical report. *Preprint*, arXiv:2303.08774.

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

Houxing Ren, Mingjie Zhan, Zhongyuan Wu, Aojun Zhou, Junting Pan, and Hongsheng Li. 2024. Reflectioncoder: Learning from reflection sequence for enhanced one-off code generation. *arXiv preprint arXiv:2405.17057*.

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

Gabriel Ryan, Siddhartha Jain, Mingyue Shang, Shiqi Wang, Xiaofei Ma, Murali Krishna Ramanathan, and Baishakhi Ray. 2024. Code-aware prompting: A study of coverage-guided test generation in regression setting using llm. *Proceedings of the ACM on Software Engineering*, 1(FSE):951–971.

John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. 2017. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*.

Jiho Shin, Clark Tang, Tahmineh Mohati, Maleknaz Nayebi, Song Wang, and Hadi Hemmati. 2023. Prompt engineering or fine tuning: An empirical assessment of large language models in automated software engineering tasks. *arXiv preprint arXiv:2310.10508*.

Parshin Shojaee, Aneesh Jain, Sindhu Tipirneni, and Chandan K Reddy. 2023. Execution-based code generation using deep reinforcement learning. *arXiv preprint arXiv:2301.13816*.

Qingxiao Tao, Tingrui Yu, Xiaodong Gu, and Beijun Shen. 2024. Unraveling the potential of large language models in code translation: How far are we? *arXiv preprint arXiv:2410.09812*.

Qwen Team. 2024. Qwen2.5: A party of foundation models.

Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, et al. 2023. Llama 2: Open foundation and fine-tuned chat models. *arXiv preprint arXiv:2307.09288*.

Yuvraj Virk, Premkumar Devanbu, and Toufique Ahmed. 2024. Enhancing trust in llm-generated code summaries with calibrated confidence scores. *arXiv preprint arXiv:2404.19318*.

Wenhan Wang, Kaibo Liu, An Ran Chen, Ge Li, Zhi Jin, Gang Huang, and Lei Ma. 2024. Python symbolic execution with llm-powered code generation. *arXiv preprint arXiv:2409.09271*.

Chunqiu Steven Xia, Matteo Paltenghi, Jia Le Tian, Michael Pradel, and Lingming Zhang. 2024. Fuzz4all: Universal fuzzing with large language models. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, pages 1–13.

HanXiang Xu, ShenAo Wang, Ningke Li, Yanjie Zhao, Kai Chen, Kailong Wang, Yang Liu, Ting Yu, and HaoYu Wang. 2024. Large language models for cyber security: A systematic literature review. *arXiv preprint arXiv:2405.04760*.

Xin Yin, Chao Ni, Tien N Nguyen, Shaohua Wang, and Xiaohu Yang. 2024. Rectifier: Code translation with corrector via llms. *arXiv preprint arXiv:2407.07472*.

Zishun Yu, Yunzhe Tao, Liyu Chen, Tao Sun, and Hongxia Yang. 2023. Beta-coder: On value-based deep reinforcement learning for program synthesis. In *The Twelfth International Conference on Learning Representations*.

Zibin Zheng, Kaiwen Ning, Jiachi Chen, Yanlin Wang, Wenqing Chen, Lianghong Guo, and Weicheng Wang. 2023. Towards an understanding of large language models in software engineering tasks. *arXiv preprint arXiv:2308.11396*.

Qihao Zhu, Daya Guo, Zhihong Shao, Dejian Yang, Peiyi Wang, Runxin Xu, Y Wu, Yukun Li, Huazuo Gao, Shirong Ma, et al. 2024. Deepseek-coder-v2: Breaking the barrier of closed-source models in code intelligence. *arXiv preprint arXiv:2406.11931*.