SelfPiCo: Self-Guided Partial Code Execution with LLMs

Zhipeng Xue
Zhejiang University
Hangzhou, China
zhipengxue@zju.edu.cn

Zhipeng Gao
Zhejiang University
Hangzhou, China
zhipeng.gao@zju.edu.cn

Shaohua Wang
Central University of Finance and Economics
Beijing, China
davidshwang@ieee.org

Xing Hu
Zhejiang University
Hangzhou, China
xinghu@zju.edu.cn

Xin Xia
Zhejiang University
Hangzhou, China
xin.xia@acm.org

Shanping Li
Zhejiang University
Hangzhou, China
shan@zju.edu.cn

Abstract

Code executability plays a vital role in software debugging and testing (e.g., detecting runtime exceptions or assertion violations). However, code execution, especially partial or arbitrary code execution, is a non-trivial task due to missing definitions and complex third-party dependencies. To make partial code (such as code snippets posted on the web or code fragments deep inside complex software projects) executable, the existing study has proposed a machine learning model to predict the undefined element types and inject the pre-defined dummy values into execution. However, the performance of their tool is limited due to its simply designed dummy values and the inability to continue learning. In this paper, we design and implement a novel framework, named **SelfPiCo** (**Self**-Guided **P**arti**i**al **Co**de Executor), to dynamically guide partial code execution by incorporating the open-source LLM (i.e., Code Llama) within an interactive loop. Particularly, **SelfPiCo** leverages few-shot in-context learning and chain-of-thought reasoning to elicit human knowledge and logical reasoning based on fine-tuning the Code Llama model. **SelfPiCo** continuously learns from code execution results and refines its predictions step after step. Our evaluations demonstrate that **SelfPiCo** can execute 72.7% and 83.3% of all lines in the open-source code and Stack Overflow snippets, outperforming the most recent state-of-the-art Lexecutor by 37.9% and 33.5%, respectively. Moreover, **SelfPiCo** successfully detected 18 and 33 runtime type error issues by executing the partial code from eight GitHub software projects and 43 Stack Overflow posts, demonstrating the practical usage and potential application of our framework in practice.

CCS Concepts

• **Software and its engineering → Software testing and debugging**.

Keywords

Partial Code Execution, Dynamic Analysis, Large Language Model, Prompt Engineering

ACM Reference Format:
Zhipeng Xue, Zhipeng Gao, Shaohua Wang, Xing Hu, Xin Xia, and Shanping Li. 2024. SelfPiCo: Self-Guided Partial Code Execution with LLMs. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '24), September 16–20, 2024, Vienna, Austria.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3650212.3680368

ISSTA '24, September 16–20, 2024, Vienna, Austria

ACM ISBN 979-8-4007-0612-7/24/09
https://doi.org/10.1145/3650212.3680368

1 Introduction

To share ideas or programming techniques, developers write code snippets to illustrate specific task solutions and/or demonstrate programming concepts in the software development community, such as Stack Overflow or GitHub [19, 20, 23, 66, 67]. These arbitrary code snippets are often written for illustrative purposes and as quick ways to convey solutions, without implementation detail, which are widely used by developers [21, 22, 49]. Despite the wide adoption of code snippets among developers, 75% of the code snippets can not be directly executed [16, 18, 79, 80] and reused. This is because a significant number of code snippets are partial and incomplete (i.e., missing variable or function definitions, missing third-party dependencies). Therefore, executing arbitrary code snippets written by developers is essential for reusing these code snippets immediately and effectively.

The capability of executing partial code also facilitates diverse applications of dynamic program analysis, such as taint analysis [4, 9, 32, 60], vulnerability and bug detection [17, 29, 39, 41, 42, 44, 68, 74, 82], type inference [25, 43, 51, 52]. Dynamic analysis provides valuable insights into a program's runtime behavior, capturing information such as actual data inputs, execution traces, and system reactions. It has proven to be effective in unveiling various runtime bugs (e.g., memory leaks, buffer overflow, race conditions [34, 71]). However, for large-scale software projects, it is difficult, if not possible, to run the dynamic analysis tools on any arbitrary code area that is deep inside the project. Executing arbitrary code fragment enable us to run dynamic analysis tools on the key components and vulnerable code area (e.g., the newly updated code), without worrying about the complex building procedure and sophisticated third-party dependencies.

To achieve the goal of executing arbitrary code snippets, Souza et al. [63] first proposed Lexecutor, a neural network guided tool to

predict and inject missing values into program execution. In particular, when a missing element (e.g., variable, attributes, or function calls) is encountered, their approach queries a machine learning model (i.e., CodeT5[70]) to predict the element type and inject a pre-defined dummy value instead. However, the performance of Lexecutor is still relatively suboptimal in terms of the code coverage on open-source project functions (50.6%) and Stack Overflow code snippets (61.0%). After empirically investigating their experimental results, two main challenges are observed regarding their approach: (i) the pre-defined dummy values are too simple and inflexible to cover the practical scenarios in the real development environment. For instance in Listing 1, Lexecutor successfully predicts the correct type of `filter_cached`, i.e., *Callable*. Then Lexecutor will inject a pre-defined *DummyCall* for it. However, the program will crash during the execution, since the expected return of `filter_cached` includes two values, while the pre-defined *DummyCall* returns only a single value. (ii) the disability of interactive learning. The Lexecutor uses a machine learning model to predict the missing value types, the prediction results are constant when the input samples are fixed. It cannot continue learning from the program execution results, which can provide valuable information to guide the model to make more accurate predictions. A skilled developer can gain insights from failed execution results to refine predictions step by step. According to the error message in Listing 1, the skilled developer would rectify the *DummyCall* by returning either two values or an iterable object, e.g., *Tuple*. Thus the key question we ask in this work is: *can we design models that can continuously learn from code execution results and incrementally refine predictions, ultimately enabling non-executable code to become executable.*

Listing 1: A Failure Case of Lexecutor

```
# Original Code: black/src/black/concurrency.py:
sources, cached = filter_cached(cache, sources)
# Lexecutor Injection:
filter_cached = DummyCall(*args)
TypeError: cannot unpack non-iterable DummyObject object
# SelfPiCo Injection:
def filter_cached(*args):
    return (1, 2)
```

Inspired by the impressive capacities of LLMs (Large Language Models) for code comprehension and their great potential for interacting with humans [10, 15, 47–49, 70, 78, 81], in this work, we first investigate incorporating LLMs for the task of executing arbitrary code snippets. The key idea of this work is **LLM-in-the-loop**. Compared with human-in-the-loop (HITL) which uses human interaction to aid computers in making decisions, we first introduce the concept of LITL (LLM-in-the-loop), where the LLMs are engaged within an interactive loop for generating useful artifacts. In particular, we design and implement a novel LLMs-based framework, named **SelfPiCo**, to guide partial code execution. The **SelfPiCo** is constructed by following three components:

- **Interactive Value Predictor.** The interactive value predictor is the core module of **SelfPiCo**, which includes an *interactive value generator* and an *execution value checker*. The *interactive value generator* is responsible for generating likely values for the missing elements (e.g., undefined variables, return values, or missing functions). The *execution value checker* is responsible for ensuring the validity of the generated values. If the generated values provided by *interactive value generator* fail to execute the given arbitrary code snippet, the *execution value checker* will query back the *interactive value generator* with error execution messages for regenerating new likely values.
- **Complementary Type Predictor.** This component serves as a complement module to the interactive value predictor, addressing cases where the interactive value predictor exceeds maximum iterations. If the value predictor can not predict appropriate values, the complementary type predictor predicts the type of missing element and injects the pre-defined dummy value.
- **Runtime Engine.** The runtime engine instruments the partial code with execution hooks, which catch the exceptions during code execution, and inject values from interactive value predictor to guide partial code execution.

Automated program repair (APR) techniques aim to generate a patch that passes compilation and test execution and recent studies have leveraged LLMs for fixing bugs (e.g., compilation or execution bugs) [11, 30, 31, 38, 40, 75]. The goal of APR overlaps to some extent with our partial code execution. However, there are two significant distinctions between them: (i) **The goal is different.** APR aims to fully repair programs to pass all tests, while our task seeks to make partial code executable. Our work can be regarded as a base model to enable other dynamic analysis tools for checking partial code. Notably, our tool can also assist developers in fixing bugs or code errors (e.g., exposing runtime errors during execution), but fixing bugs is not the final goal of this research. (ii) **The way of interacting with code is different.** APR generates correct patches to fix bugs in buggy code, which need to modify and update the original buggy code. In contrast, our tool injects missing values to run partial code, we keep the original code untouched without changing any original code elements. Due to different goals and ways of generating code, APR methods are not applicable to our partial code execution task.

To evaluate the effectiveness of our **SelfPiCo**, we used the same dataset from Lexecutor containing two sets of code snippets: functions extracted from popular open-source projects and code snippets extracted from Stack Overflow posts. Our results indicate that the **SelfPiCo** enables the execution of 72.7% and 83.3% of all lines in the open-source code and Stack Overflow snippets, respectively, outperforming Lexecutor by 37.9% and 33.5%. Souza et al. [63] first propose the task of partial code execution, and they use Lexecutor to find the semantics-changing commits. In this paper, we attempt to validate the practical usage of **SelfPiCo** on a dynamic analysis task: runtime type error detection. Specifically, by running **SelfPiCo** on partial code fragment, our framework successfully detected 18 type error issues from eight popular Python GitHub repositories and 33 type error issues from Stack Overflow posts. In summary, this paper contributes the following:

- We design and implement a framework, named **SelfPiCo**, to engage the Code Llama model within LITL (LLM-in-the-loop) to guide the partial code execution. Our fine-tuned Code Llama model performs similarly to the close-source, commercial GPT-3.5 model in the task of guiding partial code execution. The richer

complimentary dummy types help SelfPiCo to guide more partial code execution.
- We extensively evaluate SelfPiCo on both functions extracted from popular open-source projects and code snippets extracted from Stack Overflow posts. The evaluation results show that SelfPiCo can significantly outperform Souza et al [63]'s method in both datasets (37.9% code coverage and 62.6% fully executed rate improvement on the Open-source projects dataset, 33.5% code coverage and 57.7% fully executed rate improvement on Open-source projects dataset), achieving the state-of-the-art performance.
- We validate SelfPiCo with a practical dynamic analysis application: runtime type error detection. From eight popular Python GitHub repositories and 43 Stack Overflow posts, we successfully detected 18 and 33 type error issues, respectively. To the best of our knowledge, our work is the first attempt to identify type errors at runtime, our tool can expose the runtime type error before compiling or running the entire software project, illustrating the effectiveness of our approach in practice.

2 Motivation

The ability to execute partial code is essential for various dynamic analysis applications. We demonstrate a motivating example of checking runtime type errors using our approach, however, we argue that our approach is not limited to this particular application. It can be used to incorporate dynamic analysis tools to support a wide range of applications, for example, detecting security vulnerabilities via taint analysis. Better combining our tool with advanced dynamic checking techniques is an interesting future direction, but it is beyond the scope of our current research.

Motivating Example. Python is one of the most popular programming languages nowadays. However, due to its dynamic type characteristics, variable types are determined and validated at runtime rather than compile time. Developers often suffer from runtime type errors when performing operations on inconsistent types of variables. Although Python static checkers (e.g., Pyre [2]) are designed to detect such type inconsistencies, however, they primarily rely on manually written type annotations which are unavailable most of the time. As a result, Python type errors are often hard to detect unless they are exposed at runtime. Figure 1 demonstrates an example of Python type error in *Luigi* project. Specifically, the method `replace` expects to be passed with two variables of the same type, in this case, both should be `bytes` objects. However, the developer wrongly passed a `string` object and thus introduced a type error. Due to complex internal dependencies, such type errors are difficult to trigger or reach out until bugs are eventually exposed. We manually checked the development history of the *Luigi* project, this runtime type error has existed for over two years until finally exposed by a bug issue report. During this time, any code refactorings associated with this buggy method could be influenced, posing significant risks to software quality and maintenance. It is thus beneficial to have a tool that can discover such type errors without worrying about complex code dependencies or writing extensive test cases.

SelfPiCo Usage Scenarios. SelfPiCo successfully detected this runtime type error without building/running the whole project. Based on the code snippet context, our framework correctly injects a `bytes` value object for the variable `d` and a `string` value object for the variable `module`, which successfully triggered the same runtime error reported by the bug issue report. Our framework can help developers expose this bug in an early stage (e.g., check-in time) and reduce the risks of introducing any unwanted problems or negative impacts. Suppose the developer who adopts our SelfPiCo during his/her development, when code change happens, our tool can be performed on the newly updated partial code snippets for checking runtime type errors and discovering potential type errors just-in-time. It is worth mentioning that the usage scenario of our SelfPiCo is not limited to runtime type error detection, our framework can be further extended to enable different dynamic analysis applications (e.g., assertion violation, taint analysis). In this work, we use runtime type error detection as a preliminary study to validate the practical usage of our framework.

```
# https://github.com/spotify/luigi/issues/1988
def _dump(self, fd):
    ...
    d = pickle.dumps(fd)
    module = os.path.basename(sys.argv[0])
        .rsplit('.', 1)[0]
    d = d.replace(b'(c__main__', "(c" + module)
    fd.write(d)
```

TypeError: a bytes-like object is required, not 'str'

Figure 1: A Type Error Detected From Partial Code

3 Our Approach

In this work, we design and implement an LLM-based framework, SelfPiCo, to interactively make predictions and execute partial code snippets. SelfPiCo includes three key components: the runtime engine, the interactive value predictor, and the complementary type predictor. As shown in Fig. 2, for a given non-executable arbitrary code snippet, the runtime engine first instruments it with execution hooks, and then executes the partial code and catches any exception that might be thrown when undefined code elements (e.g., variable, attribute) are met. The raised exception will trigger the execution hooks, which send the undefined element and its contextual information to the interactive value predictor for inferencing the valid values for the undefined element. The execution hooks inject the inference values to the undefined element and guide code execution. The interactive value predictor adaptively regenerates the likely values for the undefined elements and checks if these values can be executed by the runtime engine successfully. In certain cases, LLMs may fail to generate valid values even after multiple interactions, leading to the activation of the complementary type predictor. It queries the LLMs to predict the type of the undefined element and returns a pre-defined dummy value to the runtime engine. Details of each component are as follows.

3.1 Runtime Engine

The goal of the runtime engine is to catch the exception during partial code execution, query the interactive value generator, and inject the replied value to guide code execution.

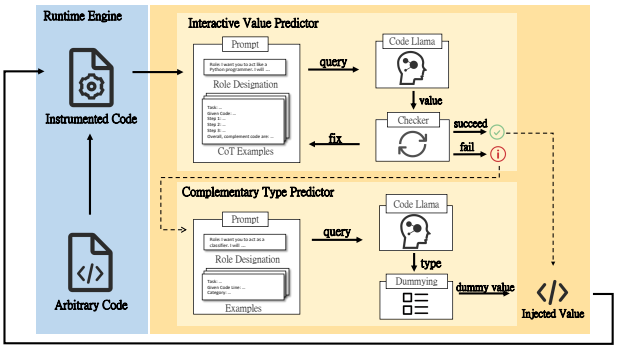

Figure 2: The Overview of SelfPiCo

The runtime engine initially instruments the arbitrary code with execution hooks from Lexecutor. It first visits the abstract syntax tree (AST) of the code and detects three types of AST nodes: variable reads, attribute reads, and calls of functions and methods. Then It instruments the detected three kinds of code by wrapping them with execution hooks. The original code and instrumented code of each kind are illustrated in Table 1. The iids refer to the instrument IDs, and the functions _n_, _a_, and _c_ are execution hooks for variable reads, attribute reads, and calls of functions and methods, respectively.

For the variable reads, the instrumented code calls the execution hook _n_, passing the name of the variable and a lambda function that tries to read the value of the variable. _n_ then returns the value from the lambda function. For the attribute reads, the instrumented code calls the execution hook _a_, passing the base object that has been assigned by _n_ and the name of the attribute. _a_ returns the value of the passing attribute of the base object. For the calls of functions and methods, the instrumented code calls the execution hook _c_, passing the callee function. Then the execution hook invokes the callee function and returns the result of it. During the execution of each hook, if it triggers some exception like *NameError*, *AttributeError*, the hook will query the interactive value predictor for a possible value. The query message of execution hooks combines the name of the code element, the kind of code element, the code line of the code element in the original code snippet located by the instrument IDs, and the error message during the execution.

3.2 Interactive Value Predictor

The interactive value predictor is to adaptively predict likely values of undefined code elements based on the contextual information and execution error message. The interactive value predictor combines a *value generator* and a *value checker*. The *value generator* generates the definition or assignment of the queried undefined code element. The *value checker* ensures the validity of the generated value by executing the definition or assignment and loading the value. If generated values are valid, the *value checker* sends the loaded value to the runtime engine, and the code execution continues. If generated values fail the validity checking, the *value checker* will query back the *value generator* again with a detailed execution error message. As a result, the interactive value predictor adaptively learns from the code execution results and progressively refines its predictions until executing the partial code successfully.

3.2.1 Interactive Value Generator. The underlying approach of the value generator is prompt engineering. Role designation, few-shot learning and chain-of-thought reasoning are incorporated to construct LLM prompts. An example of the prompts is shown in Table 2.

Role Designation. In prompt engineering, the role designation refers to designating LLMs with a specific role, providing them with a context that aids their understanding of the task context and leading to more accurate and relevant responses. In this study, since we aim to execute a Python code snippet, we designate the role of LLMs to act as a Python programmer. In addition, we also added output restriction and format restriction in the prompt. The prompt details are shown in Table 2.

In-context Few-shot Learning. Few-shot learning is utilized to augment the context with a few examples of desired inputs and outputs. In this work, to select representative examples for few-shot learning, we invited three developers with at least five years of Python programming experience. Each of them was asked to fill in the likely values for the undefined code element based on contextual information and error execution message. After manually examining 30 arbitrary code snippets by each developer, the developers then discuss the representation of the selection and refine the dataset until a consensus is reached. In total, we collect 6 examples as our representative dataset, with an example in Table 2.

Chain-of-thought Reasoning. The in-context few-shot learning has provided LLMs with a few examples to learn the expected inputs and outputs, but the LLMs still lack the logical thinking to address the complicated task. We introduce the method of chain-of-thought to elicit the ability of LLMs' reasoning and logical thinking for this study. It endows the LLMs to split a complex task into several relatively simple steps and generate a series of intermediate outputs that lead to a reasonable result. Following the previous studies [65, 69, 73], we design a three-step thinking process that leads to the prediction of likely values for the undefined code elements. An example of chain-of-thought reasoning is shown in Table 2. In particular, given a Python code snippet (e.g., `filepath = self.path`) which is non-executable: In step 1, LLMs are required to import necessary modules for the code snippet, in this case, the `os` module is imported which is relevant to `filepath` within the code snippet. In step 2, LLMs are required to define all the necessary class/method/variable undefined of the code snippet, in this case, the class `MyClass` is defined and instantiated. Two types of step 3 are designed for our task, regarding step 3 (assign), LLMs are required to learn from the runtime engine and infer the likely values for the undefined code elements. In particular, three types of information are sent to the interactive value predictor, namely the undefined code element and its type (in this case the undefined element is `path` and its type is `attribute`), and the error execution message (`Attribute Error:`self' objects has no attribute `path'` for this case), LLMs is required to inference the likely values for the missing `path` attribute, generating `self.path = os.path.abspath(__file__)`. Regarding step 3 (fix), LLMs are required to interactively fix the last round's predicted values based

Table 1: Execution Hooks

Ast Nodes	Original Code	Instrumented Code
Variable Reads	Var1 = Var2 + 1	Var1 = _n_(iid, "Var2", lamda: Var2) + 1
Attribute Reads	Opt1.Attr1 = Opt2.Attr2	Opt1.Attr1 = _a_(iid, _n_(iid, "Opt2", lambda: Opt2), "Attr2")
calls of functions & methods	Var = Foo()	Var = _c_(iid, _n_(iid, "Foo", lambda: Foo))

on the failed execution message (`NameError: name 'path' is not defined`) and last step output non-executable code (e.g., `self.path = path`). Finally, LLMs are required to summarize the aforementioned steps as outputs as shown Example Output in Table 2.

Prompt construction. We combine the aforementioned information, i.e., (<*Role Designation*> + 6 * (<*Chain-of-thought reasoning with Example Input*> + <*Example output*>)), to make two types of input prompt. Particularly, step 3 (assign) was used to construct the *initial assign prompt*, and step 3 (fix) was used to construct the *interactive fix prompt*. Two types of input prompts are constructed for the interactive value predictor, i.e., the *initial assign prompt* and *interactive fix prompt*. The *initial assign prompt* is used for initiating the interactive value predictor while the *interactive fix prompt* is used to fix undefined errors from the last round's predicted values. Due to the advantage of few-shot learning and chain-of-thought reasoning, the LLMs will consistently reply to a Python code to assign a target code element in the same format as our example output, which can be directly executed.

3.2.2 Execution Value Checker. After the value generator inference the likely values for the undefined code elements, the value checker executes the code with predicted values and queries back the value generator if necessary. In particular, the value checker first identifies and attempts to import or install the required third-party module. Then it invokes the `exec` function to execute the replied code from the value generator. if the code execution succeeds, the value checker loads the value of undefined code elements, and then it returns the loaded value and the replied code to the runtime engine. Otherwise, the value checker queries back the value generator with the previous predicted code and the error execution messages for refining.

3.2.3 LITL (LLMs-In-The-Loop) Algorithm. The key idea of incorporating LLMs in this work is to put LLMs-in-the-loop, we designate the LLMs as expert developers capable of interactively learning from execution results and finally guiding the partial code execution tasks. We demonstrate the details of the LITL (LLMs-In-The-Loop) algorithms in Algorithm 1. For a given arbitrary code snippet, LLMs interactively refine undefined code values and check these values by execution (lines 2 to 9). The *initial assign prompt* is constructed and queried to the LLMs to generate values for undefined code elements (lines 2 to 3). The algorithm then attempts to execute the generated code and catches any runtime exception (lines 4 to 7). It returns the value (i.e., R and V, the result code R refers to the predicted definition or assignment of the unknown code element from the value generator, and Loaded value V refers to the value of the unknown code element loaded from result code R execution.) if the code snippet executes successfully and no exception occurs. Otherwise, the *interactive fix prompt is constructed* and query back LLMs for refining its previous predictions (Line

Table 2: The Example of Prompt Engineering

Prompt Type	Instantiation
Role Designation	**Role:** *I want you to act like a Python programmer. I will give you Python code and comments, you should write Python code according to the comments step by step.* **Output Restriction:** *Only give reply with Python code and Do not write explanations.* **Format Restriction:** *Your reply is limited to only one code block and should wrap with backticks.*
Chain-of-thought Reasoning with Example Input	**Task:** *Complete and fix the given code to make it can be executed directly.* **Given code:** *Do not modify the given Python code or wrap it with function.* <filepath = self.path> **Step 1:** *Import needed module.* import os **Step 2:** *Define all the needed classes, methods, or variables here in detail.* class MyClass(): pass self = MyClass() **Step 3 (assign template):** *Define and assign* ⟨UNDEF ELE⟩ ⟨UNDEF ELE TYPE⟩ *to repair the error* ⟨ERR MSG⟩ **Step 3 (assign case):** *Define and assign* ⟨path⟩ ⟨attribute⟩ *to repair the error* ⟨Attribute Error: `self' object has no attribute `path' for this case⟩ self.path = os.path.abspath(__file__) **Step 3 (fix template):** *Fix the* ⟨LAST STEP CODE⟩ *since the* ⟨FAIL EXEC RES⟩ . **Step 3 (fix case):** *Fix the* ⟨self.path = path⟩ *since the* ⟨NameError: name `path' is not defined⟩ . path = os.path.abspath(__file__) self.path = path
Example Output	**Overall, complement code are:** import os class MyClass(): pass self = MyClass() path = os.path.abspath(__file__) self.path = path

8). Whenever Algorithm 1 reaches line 10, it has failed to generate valid values after t times iteration. It then throws an *InvalidValueError*, which triggers the complementary type predictor.

Algorithm 1: LITL Algorithm

Input: Kind k, name n, contextual information c of code and error message e
Output: Result code R and Loaded value V

```
1  Prompt ← InitializeAssignPrompt(k, n, c, e)
2  for i = 1 to t do
3      C ← query LLMs with Prompt
4      V ← Execute R + c and load value, or catch exception e
5      if no exception while executing and loading then
6          return V, R
7      end
8      Prompt ← BuildFixPrompt(c, e, V)
9  end
10 throw InvalidValueError
```

3.3 Complementary Type Predictor

The complementary type predictor acts as a backup component for the adaptive value predictor. In certain cases, arbitrary code elements exceed the maximum interactions and LLMs fail to predict the appropriate code value, then the complementary type predictor will be triggered to infer the type of target undefined code element and return a corresponding pre-defined dummy value. Different from Lexecutor using CodeT5 for training, SelfPiCo fine-tunes Code Llama by using the same training set of Lexecutor. We extend the predefined data types and leverage prompt engineering for type prediction. The implementation details are as follows.

3.3.1 Pre-defined Dummy Value. A pre-defined dummy value is a placeholder or default value that is used in place of a real value when the real value is not yet known or not applicable. As shown in Table 3, we reuse the built-in date type (including None, Boolean, Integer, Float, String, List, Tuple, Set, and Dictionary) and Function and Objects type (including Callable, Object, and Resource) defined by Lexecutor. Since Python is the primary programming language for deep learning and data analysis, we extend their pre-defined abstraction classes with three popular data types from third-party libraries (Tensor, Array, and DataFrame). When generated data values are not within the aforementioned abstraction classes, a dummy object value is injected by our approach.

3.3.2 Prompt Engineering. Similar to constructing a prompt for adaptive value predictor, we first design the role of LLMs as: *I want you to act as a classifier, I will give a line of Python code and a <word> in the code. You will classify the <word> into a category from None, Boolean, Integer, Float, String, List, Tuple, Set, Dictionary, Tensor, Array, DataFrame, Callable, Resource, Object.* Then we restrict the output of LLMs: *I want you to only reply with the classified category and nothing else. Do not explain the result.* Since the type predictor is a comparatively straightforward classification task, we randomly select an example of each abstraction class as a few-shot learning example. In the end, we input the undefined code element and its contextual information to LLMs for inference.

Table 3: Pre-defined Dummy Value

Type	Abstract Class	Dummy Value
Built-in Data Type	None	None
	Boolean	True
	Integer	1
	Float	1.0
	String	"a"
	List	[Dummy()]
	Tuple	(Dummy())
	Set	set(Dummy())
	Dictionary	"a": Dummy()
Third-party Data Type	Tensor	torch.tensor([[1.0]])
	Array	numpy.array([1])
	DataFrame	pandas.DataFrame({"a": 1})
Functions &objects	Callable	DummyCall
	Object	Dummy()
	Resource	DummyResource()
Others		Dummy()

3.4 Implementation

For the LLMs, we use the Code Llama instruct model with 34B parameters [58], which is the state-of-the-art open-source LLMs for coding applications. We fine-tuned it with the Lexecutor code element type dataset [63] to perform the missing type prediction. We employed the Parameter-Efficient Fine-Tuning strategy (PEFT)[45], and Low-Rank Adaptation (LoRA)[26] to accelerate the fine-tuning process. Based on 4*A800 Gpus, the fine-tuning process followed these hyperparameters: learning rate of $3e^{-4}$, batch size of 128, 2 epochs, and a warmup ratio of 100. After fine-tuning, the fine-tuned model is then used as our interactive value predictor and complementary type predictor by further leveraging prompt engineering and chain-of-thought reasoning techniques.

We set the number of few-shot learning examples for the adaptive value predictor and complementary type predictor as 6 and 15, respectively. The 6 examples in the adaptive value predictor include 2 examples for each kind of undefined code element and 15 examples in the complementary type predictor include an example for each kind of abstract class. According to a small-scale pilot study, we set the maximum threshold in the adaptive value predictor as 5.

4 Evaluation

4.1 Experimental Setup

Dataset. Following the experiment set in Lexecutor, we reused the same datasets which included two sets of code snippets: functions extracted from popular open-source projects and code snippets extracted from Stack Overflow posts. To avoid potential bias, as the raw data set is not provided, we carefully followed the data collection steps in Lexecutor. Particularly, we initially extracted all the functions from five popular projects evaluated in Lexecutor, and we randomly selected 200 samples from each project. The dataset is composed of 1,000 randomly selected functions, that amount to 7,225 non-empty, non-comment lines of code. To build the Stack Overflow code snippets dataset, we search for questions with the

tag Python, then we randomly select an answer and extract the code in the top 1,000 votes questions. After removing the code snippets with invalid syntax, we collected 586 code snippets involving 4,540 non-empty, non-comment lines of code. The detail of the Stack Overflow code snippets dataset is shown in Table 4.

Table 4: Detail of Experiment Datasets

Dataset		Count	Loc
Open-source Projects Functions	Black	200	2,162
	Flask	200	1,100
	Pandas	200	1,438
	Scrapy	200	1,150
	TensorFlow	200	1,375
Stack Overflow Code Snippets		586	4,540

Baseline. We set up Lexecutor [63] which achieves states-of-art performance in partial code execution guiding. Lexecutor fine-tuned the pre-trained models (i.e., CodeT5 [70] and CodeBert [15]) with collected <code, type> tuple in the training phase. During the execution phase, the runtime engine inputs the CodeT5 model with code and injects a dummy value according to the predicted type. Following the instructions in the replication package, we first collected training and validation sets from five popular open-source projects and fine-tuned CodeT5 with the set hyperparameters same as Lexecutor (denoted as **Lexecutor-CodeT5**), i.e., learning rate, epochs, and batch size. We achieved 79.1%, 86.9%, and 90.2% accuracy of the top-1,3,5 predictions, respectively. The evaluation results of it closely match the accuracy they reported, we thus are confident with our replication process for Lexeuctor. Since **SelfPiCo** is based on Code Llama, to conduct a more fair comparison with Lexecutor, we replaced the CodeT5 of Lexecutor with Code LLama, denoted as **Lexecutor-CodeLlama**. Following the methodology described in Sec 3.4, we fine-tuned the Code Llama model using the same strategy and hyperparameters as **SelfPiCo**. we also replaced Code Llama with ChatGPT [1] within our framework as a baseline, denoted as **SelfPiCo**-GPT-3.5, which is used without any fine-tuning.

Metrics. We evaluate the ability of **SelfPiCo** to guide code execution in average code coverage and fully executed rate. The **Code Coverage** refers to the ratio of the number of executed lines of code to the total number of lines of code in the program. If the entire line has been executed without crashing, we label this line as "covered". **The Branch Coverage** refers to the ratio of the number of executed branches to the total number of branches in the program. The **Fully Executed Rate** measures how many of all code snippets we achieve 100% line coverage. The temperature hyperparameter of pre-trained generative models (including both CodeT5 and CodeLlama) controls the randomness of generated outputs. To help code execution cover as many lines as possible, we set the temperature hyperparameter to 0.8. This introduces more randomness and diversity in the generated outputs, allowing for a wider range of possible responses. We calculate the above metrics by combining results from five independent executions. The higher the metrics score, the better the approach can guide the incomplete code execution.

4.2 RQ1: Effectiveness of SelfPiCo

To measure the effectiveness of **SelfPiCo** in covering and successfully executing non-executable code, we evaluated **SelfPiCo** on datasets constructed of popular open-source project functions and Stack Overflow code snippets. The evaluation results are shown in Table 5. Lexecutor-CodeLlama has its advantage over Lexecutor-CodeT5, this is reasonable because Code Llama is a more powerful LLM which is 523 times larger than CodeT5. The performance of **SelfPiCo** is significantly better than Lexecutor-based models (i.e., Lexecutor-CodeT5 and Lexecutor-CodeLlama) in both open-source project functions and Stack Overflow code snippets in terms of all metrics. We attribute this to the ability of **SelfPiCo** for interactive learning by refining its predictions from execution results. In addition, our extension of pre-defined dummy values also contributed to better results. By comparing **SelfPiCo** with **SelfPiCo**-GPT-3.5, we can see that they achieve a very close performance on both datasets, suggesting the generalizability of our approach for incorporating different LLMs. Compared with ChatGPT which contains 175B parameters, Code Llama is much smaller with only 34B parameters. Nonetheless, **SelfPiCo** can achieve a comparable or even better performance than GPT-3.5 after fine-tuning, verifying the effectiveness of the fine-tuning process.

Table 5: Overall Effectiveness Evaluation

Approach	Metrics	Open-source Projects Functions	Stack Overflow Code Snippets
Lexecutor-CodeT5	Code coverage	0.527	0.624
	Branch Coverage	0.312	0.474
	Fully Executed Rate	0.342	0.478
Lexecutor-CodeLlama	Code coverage	0.542	0.641
	Branch Coverage	0.371	0.511
	Fully Executed Rate	0.373	0.502
SelfPiCo-GPT-3.5	Code Coverage	0.730	0.819
	Branch Coverage	0.644	0.789
	Fully Executed Rate	0.556	0.746
SelfPiCo	Code Coverage	0.727	0.833
	Branch Coverage	0.643	0.795
	Fully Executed Rate	0.556	0.754

Fig. 3 demonstrates the Venn diagrams of fully executed code snippets reported by our approach and Lexecutor. We can find that, **most code snippets covered by Lexecutor are also covered by our approach, while our approach can cover far more cases Lexecutor can not handle.** For example, 228 project functions (40%) and 185 Stack Overflow code snippets (39.8%) are successfully executed by our approach but failed to be handled by Lexecutor. While only 14 project functions (2.5%) and 23 Stack Overflow code snippets (4.9%) are reported by Lexecutor but missed by ours. This further justifies the superiority of our proposed **SelfPiCo**. We also observed that several cases can not covered by our approach and/or Lexecutor, We detailed discussed why we work and why we fail in Section 4.6.

4.3 RQ2: Component Analysis

The performance of **SelfPiCo** mainly relies on two components: the interactive value predictor and the complementary type predictor. We evaluate the performance of each component respectively. In particular, we compare **SelfPiCo** with two incomplete versions:

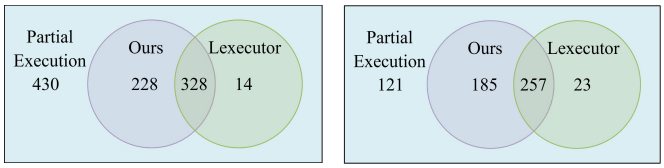

Figure 3: Venn Graph for Fully Executed Code Snippets Reported by SelfPiCo and Lexecutor, Open-Source Project Functions (left) and Stack Overflow Code Snippets (right)

- Interactive value predictor. In this version, we only keep the interactive value predictor and remove the complementary type predictor, the values are directly injected from the interactive value predictor.
- Complementary type predictor. In this version, we only use the complementary type predictor to infer the type of queried element and inject the pre-defined value.

Table 6: Effectiveness of two components in SelfPiCo

Approach	Metrics	Open-source Projects Functions	Stack Overflow Code Snippets
Interactive Value Predictor	Code coverage	0.636	0.744
	Branch Coverage	0.552	0.704
	Fully Executed Rate	0.477	0.679
Complementary Type Predictor	Code Coverage	0.558	0.656
	Branch Coverage	0.390	0.521
	Fully Executed Rate	0.382	0.514
SelfPiCo	Code Coverage	0.727	0.833
	Branch Coverage	0.643	0.795
	Fully Executed Rate	0.556	0.754

The results are shown in Table 6. From the tables, several points stand out: (i) No matter which component we removed, it reduces the performance of our approach in guiding partial code execution. This verifies the importance and usefulness of our interactive value predictor and complementary type predictor. (ii) The interactive value predictor and complementary type predictor can complement and enhance the performance of each other. For example, the interactive value predictor performs better on Stack Overflow code snippets while the complementary type predictor achieves better performance on open-source project functions. Although both components use our fine-tuned Code Llama model for inference, the interactive value predictor focuses on generating likely values for undefined code elements, while the complementary type predictor focuses on predicting the types, the different learning objectives of these two sub-components make them a suitable pair to enhance each other's capabilities. As a result, after combining these two modules, the performance of **SelfPiCo** is significantly boosted and achieved state-of-the-art performance.

4.4 RQ3: Sensitivity Analysis

The interaction learning and chain-of-thought reasoning are the core mechanisms of our approach. To explore the effectiveness of the above mechanisms, we construct a sensitivity analysis. To demonstrate the effectiveness of interactive learning from code execution results, we evaluate the performance of **SelfPiCo** after each iteration, to demonstrate the effectiveness of chain-of-thought reasoning, we evaluate **SelfPiCo** using the prompt without chain-of-thought settings. It is worth mentioning that to better present the performance contributed by each mechanism alone, we drop the complementary type predictor for this RQ setting.

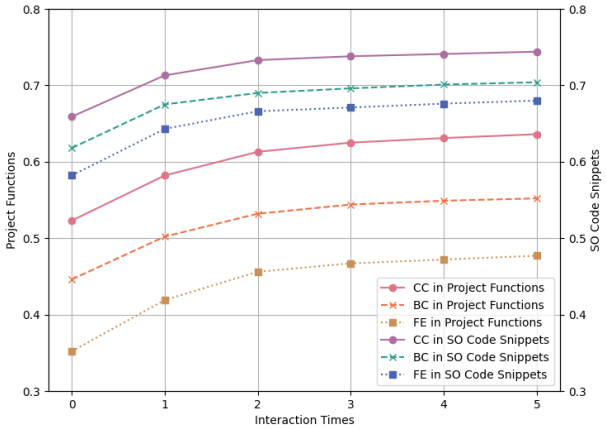

Figure 4: Performance of Interactive Value Predictor

Fig. 4 illustrates the performance of our approach under different interaction times. We found that the performance of our approach rapidly increased after the initial two interactions. Regarding simple problems such as introducing unknown variables or parameters, can be easily resolved after one or two iterations of interactively learning. This further confirms the self-guided interactive learning ability of our fine-tuning Code Llama model. Then the improvement ratio slows down after 3 interactions, the reason for this can be the insufficient code context information and/or unclear code execution error message, which indicates that the fine-tuning Code Llama model is helpful but not a 'silver bullet' for value prediction. The 5 interaction times we used in our settings are reasonable for achieving optimal results.

Table 7 illustrated the performance of our approach (drop complementary type predictor) with and without chain-of-thought reasoning. The code coverage, branch coverage and fully executed rate decreased by 24.7%, 61.4% and 45.9% on the open-source project functions dataset and those decreased by 15.3%, 31.1% and 27.5% on the Stack Overflow code snippets dataset. We found that, without the chain-of-thought reasoning, the logic reasoning capability of our approach drops significantly, for example, the fine-tuning Code Llama model often uses a module before importing it (as defined in step 1), query undefined classes or variables (as defined in step 2), this further justifies the effectiveness of chain-of-thought reasoning for prompt engineering.

Table 7: Effectiveness of Chain-of-thought Reasoning

Approach	Metrics	Open-source Projects Functions	Stack Overflow Code Snippets
Interactive Value Predictor w/o CoT	Code Coverage	0.479	0.630
	Branch Coverage	0.213	0.485
	Fully Executed Rate	0.258	0.493
Interactive Value Predictor	Code Coverage	0.636	0.744
	Branch Coverage	0.552	0.704
	Fully Executed Rate	0.477	0.680

4.5 RQ4: Time Cost Analysis

In this RQ, to evaluate the efficiency of **SelfPiCo**, we conduct the time cost analysis regarding two aspects: (i) We compare the time cost of each prediction taken by **SelfPiCo** and Lexecutor; (ii) We compare the performance of Lexecutor and **SelfPiCo** by allocating them with the same time budget.

For the first aspect of time cost analysis, the prediction time of Lexecutor costs from 0.18s to 0.48s, while SelfPiCo takes 6.25s to perform a single prediction. We found that the time cost of **SelfPiCo** is largely due to the interactive learning process (i.e., 2.28s for a single round of interactive learning in open-source project functions). Besides, the model in Lexecutor only needs to output a predicted type, while **SelfPiCo** must generate a concrete code snippet. The time cost of **SelfPiCo** can be reduced with parallelization and more advanced hardware. Moreover, we argue that **SelfPiCo** is a general framework that can easily be incorporated with other smaller-size LLMs, further decreasing the time costs.

Regarding the second part analysis, we equitably compared the performance of Lexecutor by allocating it with the same time spent by **SelfPiCo**. Specifically, we set different temperature values to run multiple-rounds of execution until reaching the time limit. The experimental results remained the same (e.g., 0.527 code coverage on the Open-source dataset and 0.624 code coverage on the StackOverflow dataset) between the basic five-round executions in the previous experiment and subsequent multiple-round executions, suggesting allocating extra time to Lexecutor can't bring performance gains.

4.6 Result Discussion

Why SelfPiCo works. As shown in Fig. 3, there are 86 functions and 125 code snippets that can be fully executed by our approach but failed by Lexecutor. We summarize three advantages of our approach over Lexecutor, including valid value injection, accurate type prediction, and comprehensive data types.

In particular, compared with Lexecutor: **First, we generate more accurate values** for the undefined code elements by self-guided interactive learning. The Lexecutor uses the pre-defined dummy values to fill the code, to compare, the interactive learning of **SelfPiCo** can dynamically assign and refine the required values based on code context and execution results. As the Listing 1 shows, Lexecutor successfully predicts the correct type (i.e., *Callable*) for the undefined `filter_cached`, but the pre-defined value *DummyObject* conflicts the expected return value. According to the error message, our approach guides the model to assign the value as a tuple, which can be unpacked into two values and successfully address the issue. **Secondly, we predict more accurate types with the fine-tuned Code Llama model.** Compared with CodeT5, LLMs are trained on ultra-large-scale datasets and exhibit promising performance in code understanding and logical reasoning, which have achieved great accuracy on type prediction [13]. Listing 2 - Ex.1 illustrates an example where the queried code element `declarations` is a list, but Lexecutor predicts it as *DummyObject*, resulting in a type error. In contrast, our approach accurately identifies the type of `declarations` and assigns it with a tuple, enabling successful value retrieval later on. **Thirdly, we apply more comprehensive data types with third-party libraries.** In Listing 2 - Ex.2, the unknown code element `df` is actually a `DataFrame` provided by the pandas module. None of the pre-defined dummy values in Lexecutor could be injected appropriately. However, our approach overcomes this limitation by importing the pandas module and defining the `DataFrame` as an extended third-party data type, allowing code execution successfully continues.

Listing 2: Successful & Failed Cases of SelfPiCo

```
# Ex1: Accurate type prediction
# Original Code: black/src/black/concurrency.py:
for prop, value in declarations:
    prop = prop.lower()
    value = value.lower()
# Lexecutor Injection:
declarations = DummyObject
TypeError: cannot unpack non-iterable DummyObject object
# SelfPiCo Injection:
declarations = [("color", "red")]
-------------------------------------------------

# Ex2: Comprehensive data types
# Original Code: pandas/tests/groupby/transform/
    test_transform.py:
expected = df[-df.a.isin(drop_idx.index)]
# Lexecutor Injection:
df = DummyObject
TypeError: bad operand type for unary -: 'DummyObject'
# SelfPiCo Injection:
df = pd.DataFrame({'a': 1})
-------------------------------------------------

# Ex3:Insufficient code Instrumentation &
    Inadequate contextual information
# Original Code: black/src/black/trans.py:
LL = line.leaves
…
if LL[comma_idx].type == token.COMMA:
# SelfPiCo Injection:
class Line:
    def __init__(self, leaves):
        self.leaves = leaves
line = Line([])
IndexError: list index out of range
```

Limitations of SelfPiCo. We also investigate why our **SelfPiCo** fails to execute certain partial codes, two main reasons are identified, as shown in Listing 2 - Ex.3: **Inadequate contextual information.** In our approach, we only input the line where the undefined code element is located as the contextual information. The generated value may satisfy the current line execution requirement but conflict with the subsequent code. For example in Ex.3, without the following contextual information about `LL`, **SelfPiCo** assigns an empty list for it. **Insufficient code instrumentation.** In the code instrumentation phase, we utilize the Lexecutor and instrument three kinds of execution hooks: variable reads, attribute reads, and calls of functions. However, these hooks are insufficient and miss some important operations, such as indexing, and binary operation. For example, the empty list assignment to `LL` results in an *IndexError* when indexing operation `LL[comma_idx]` is performed. However, none of the exception hooks can catch the *IndexError*, leading to the termination of code execution.

5 Practical Applications

5.1 Runtime Type Error Detection

In this section, we apply **SelfPiCo** in a real dynamic program analysis task: runtime error detection. As discussed in the Motivation Section, Python runtime errors are often hard to discover and/or trigger until the bugs are eventually exposed. To verify the practical usage of our framework, we apply **SelfPiCo** in real GitHub projects and Stack Overflow posts to assess its effectiveness.

Particularly, to collect the real type errors from GitHub, we first selected eight popular GitHub open-source projects that have more than 1000 stars (i.e., Pandas, Airflow, Luigi, Ansible, Core, Keras, Requests, and Salt). Then for each project, we searched the pull requests containing the keyword *Type Error* to find those reporting and fixing type error issues. We excluded type errors whose messages included project-specific domain knowledge that could not be generated. Finally, we collected 42 type errors from the above eight projects for our evaluation. For each collected type error, we check out the code snippet that introduced type error as our partial code input. To collect real type errors from Stack Overflow posts, we randomly selected 200 posts with the keyword "Type Error" in the title or body. After filtering out code snippets containing invalid syntax, we collected 47 unique type errors for analysis. We arranged each partial code into a separate file and then leveraged **SelfPiCo** to run the partial code to see if the target type error could be successfully triggered. If and only if our approach terminates at the same fault localization and reports the same error messages with the issues, we consider this type error as successfully detected.

SelfPiCo successfully detected 18 Python type errors from 42 collected ones from the GitHub projects and 33 Python type errors from 47 collected ones from the Stack Overflow forum. Lexecutor-CodeLlama can only detect 8 and 21 Python type errors from GitHub projects and Stack Overflow forum, respectively. Figure 5 shows a detected type error. First, **SelfPiCo** injects the value of `True` for `get_logs`, and the variable `last_log_time` is assigned with `None`. When executing the buggy line, **SelfPiCo** predicted the undefined variable `pendulum` as an object which includes a method `now` to return the current time. Then **SelfPiCo** successfully detected the runtime type error: *TypeError: unsupported operand type(s) for -: 'DateTime' and 'NoneType'*. The runtime type error detection shows the practical value of our tool to facilitate the dynamic program analysis of applications. However, there are still cases **SelfPiCo** can not handle correctly, the failed cases primarily stem from cases where the partial codes are too complicated to perfectly handle, and/or insufficient context to infer the error triggered types (e.g., `last_log_time` may also be assigned with `DateTime` object). Furthermore, to estimate the false positive rate of SelfPiCo, we randomly sampled 50 function bodies from the 8 open-source projects and 50 partial codes without error from the Stack Overflow forum. Following that, we run SelfPiCo to execute these 100 partial codes to see if any potential type errors will be triggered. The experimental results show that SelfPiCo reported 11 type errors for 100 partial codes, resulting in a false positive rate of 11%, while Lexecutor-CodeLlama performed a 28% false positive rate. The relatively low false positive ratio further confirms the practical usage of our approach. We manually checked false positive cases, these

```
# https://github.com/apache/airflow/pull/14513
if get_logs:
    read_logs_since_sec = None
    last_log_time = None
...
delta = pendulum.now() - last_log_time
read_logs_since_sec = math.ceil(delta.
    total_seconds())
```

TypeError: unsupported operand type(s) for -: 'DateTime' and 'NoneType'

Figure 5: Type error detected by SelfPiCo

failed cases are primarily caused by imprecise value predictions (7 cases due to lack of code context, 4 cases due to complex variable value), it would be interesting to address these limitations in future work.

5.2 Discussion

Unit test generation (UTG) technology is widely used to detect runtime errors, recent research also leveraged LLMs to generate unit test [36, 59, 84]. For example, Schafer et al. [59] introduced the LLM-based model TestPilot to generate tests by re-prompting the model with error messages, Yuan et al. [84] proposed a ChatGPT-based model ChatTester to leverage ChatGPT to improve unit test generation. UTG methods differ from our research as follows: (i) **UTG methods are incapable of handling partial code.** UTG methods, such as TestPilot and ChatTester, require the method under test (i.e., focal method) can be invoked and executed properly. The underlying assumption is that focal methods should be complete and compilable, while either our open-source functions (uncompilable) or Stack Overflow code snippets (incomplete and uncompilable) fail to satisfy such conditions. In other words, the partial code can not be directly invoked and executed, making UTG tools unable to generate tests for them. For example, TestPilot and ChatTester can't generate unit tests for SO code snippets, because 75% SO code snippets can not be executed. Moreover, SO code snippets are often code lines and lack method signatures, rendering UTG tools ineffective; (ii) **SelfPiCo can discover different runtime errors that UTG can not detect**. Based on the focal method, UTG tools (e.g., TestPilot and ChatTester) generate a unit test that invokes the target method with reasonable input parameter values and checks the output with corresponding assertions. UTG injects values only at well-defined interfaces, such as function entry points. While **SelfPiCo** can inject valid runtime values in arbitrary points of the code during execution on-demand, enabling the discovery of bugs that are not triggered by changing input value changes. Such as the state-dependent issues and boundary condition problems, these bugs can't be essentially checked by UTG methods, **SelfPiCo** can assist developers to find these hidden bugs in runtime.

6 Threats to Validity

In our experiments evaluating our model, threats to internal validity may arise from the randomness of LLMs generation, which may generate different results for different runs. It means LLMs may reply with different outputs based on the same input. To mitigate this threat, we calculated the metrics by combining results from five independent executions.

The main external threat to the validity of our work is the representative of the testing dataset selected to evaluate our approach. To mitigate this threat, we followed the same strategy as the baseline method, representing an unbiased testing dataset for our study. Moreover, Our practical evaluation is based on known issues confirmed or reported by developers, these data samples can be regarded as ground truth and we can easily measure the effectiveness of our **SelfPiCo** on these samples. It would be interesting future research direction to use our tool to detect more runtime-type errors in the wild.

7 Related Work

Incompletion code execution. micro-execution [24] builds a runtime Virtual Machine that allows for executing arbitrary x86 code by injecting binary values into memory on demand. X-force [55] executes arbitrary binary code and fixes the invalid memory by setting the offending pointers to the allocated memory. UC-KLEE [57] extends the symbolic execution (KLEE) for an incompletion code snippet. J-Force [33] forced to execute the uncovered path and inject the value candidates from data flow for missing objects. JS-Force [64], Dual-Force [27] and Oyama et al. [53] explored the execution paths of arbitrary malware code by switching between different execution paths when encountering exceptions. LExecutor [63] predicted the type of missing code element and injected the corresponding injected pre-defined dummy value. Our work fundamentally differs from the above approach by predicting the definition and assignment of missing code elements and injecting realistic value.

Execution behaviour analysis Since several tasks require the behavior of code execution, some research focuses on predicting the behavior of code execution. Bieber et al. [7] proposed an instruction pointer attention graph neural network (IPA-GNN) to infer the runtime value of each variable. Some research aimed to predict the type of dynamic Language [50, 56] or binaries [35, 54]. TRACED [13] fine-tuning the large language model by the execution trace of code and predicting the execution branch of code without execution. Moreover, Bieber et al. [6] predicted whether a program has runtime errors or an exception raised. The approach mentioned above illustrated the feasibility of predicting the runtime behavior of the program. Compared to all the above work, our approach not only predicts the runtime value of the code element but also practically executes the program.

Automated Program Repair. APR tries to modify a program to achieve successful compilation or execution, which overlaps to some extent with our partial code execution. A common approach to APR regards it as a code transformation task, which transforms the buggy program into a bug-fixing program [28, 37, 83, 86]. Recent research has explored the potential of large language models for program repair [76]. Several studies have demonstrated that LLMs display a basic level of program comprehension that can be for APR [11, 30, 31, 75]. Different from APR, our study focuses on partial code execution and will not modify the target program or its semantics.

Dynamic analysis for Python. Dynamic analysis is crucial in program analysis, and there is some research work for Python dynamic analysis. For the runtime dynamic analysis, Xu et al. [77] collected the execution trace of the Python program and leveraged the SMT solver to detect bugs. Chen et al [8] instrumented the bytecode of the Python program and executed instrumented bytecode to capture the data and control flow and sliced the file. SCALENE [5] is a high-performance CPU, GPU, and memory profiler for Python, which monitors memory usage during Python program execution. DynaPyt [14] is a dynamic analysis framework that instruments the code with the analysis hooks and supports customized dynamic analysis tasks. Fuzzing technology is widely used for bug identifying during execution[12, 61, 72]. The above approaches need to execute the Python code, while our approach can support them to analyze non-executable Python code. For compile-time dynamic analysis, angr [62] translated the binary code into an intermediate representation (IR) and performed symbolic execution. Triton is a dynamic analysis applied taint analysis and symbolic execution on the instrumented IR. Since the non-executable code can not be translated to valid IR, our approach can complement the code to be successfully compiled and executed.

Unit Test Generation. UTG is widely used to detect errors dynamically, which can also detect the runtime time errors. Based on the target function, these tools produce a unit test that invokes the target function with reasonable input parameter values and checks the output with corresponding assertions [46, 85]. Recently there are also some tools leveraging LLMs to generate unit test [36, 59, 84]. A key difference to our work is that UTG assumes that the target method is complete and compilable, which can be invoked and executed directly, while our **SelfPiCo** aims to detect runtime type errors from un-executable partial code.

8 Conclusion and Future Work

Aim to dynamically analyze arbitrary code, e.g., non-executable partial code snippets, we introduce **SelfPiCo** leveraging the powerful learning capabilities of LLMs to interactively fill in the partial code to make it executable. The experiments demonstrate the effectiveness of our approach in guiding partial code execution. The exploratory study of **SelfPiCo** on the practical usage shows that **SelfPiCo** successfully detects 51 type errors, illustrating the usefulness of our approach in arbitrary code dynamic analysis. In this study, we only use error messages for applying LLMs, we will explore other domain information during code execution (such as partial AST and API sequence) in our future work.

9 Data Availability

Our replication package is available at [3].

Acknowledgments

This research is supported by the Starry Night Science Fund of Zhejiang University Shanghai Institute for Advanced Study, Grant No. SN-ZJU-SIAS-001. This research is partially supported by the Shanghai Sailing Program (23YF1446900) and the National Science Foundation of China (No. 62202341). This research is partially supported by the Ningbo Natural Science Foundation (No. 2023J292). This research was also supported by the advanced computing resources provided by the Supercomputing Center of Hangzhou City University. The authors would like to thank the reviewers for their insightful and constructive feedback.

References

[1] 2023. Introducing ChatGPT. https://chat.openai.com/.

[2] 2023. Pyre. https://pyre-check.org/.

[3] 2024. Our replication package. https://zenodo.org/records/10401593.

[4] Mark W. Aldrich, Alexi Turcotte, Matthew Blanco, and Frank Tip. 2022. Augur: Dynamic Taint Analysis for Asynchronous JavaScript. *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering* (2022). https://api.semanticscholar.org/CorpusID:255441495

[5] E. Berger. 2020. Scalene: Scripting-Language Aware Profiling for Python. *ArXiv* abs/2006.03879 (2020).

[6] David Bieber, Rishab Goel, Daniel Zheng, H. Larochelle, and Daniel Tarlow. 2022. Static Prediction of Runtime Errors by Learning to Execute Programs with External Resource Descriptions. *ArXiv* abs/2203.03771 (2022).

[7] David Bieber, Charles Sutton, H. Larochelle, and Daniel Tarlow. 2020. Learning to Execute Programs with Instruction Pointer Attention Graph Neural Networks. *ArXiv* abs/2010.12621 (2020).

[8] Zhifei Chen, Lin Chen, Yuming Zhou, Zhaogui Xu, William Cheng-Chung Chu, and Baowen Xu. 2014. Dynamic Slicing of Python Programs. *2014 IEEE 38th Annual Computer Software and Applications Conference* (2014), 219–228.

[9] James A. Clause, Wanchun Li, and Alessandro Orso. 2007. Dytan: a generic dynamic taint analysis framework. In *International Symposium on Software Testing and Analysis*. https://api.semanticscholar.org/CorpusID:11142970

[10] Zhenlong Dai, Chang Yao, WenKang Han, Ying Yuan, Zhipeng Gao, and Jingyuan Chen. 2024. MPCODER: Multi-user Personalized Code Generator with Explicit and Implicit Style Representation Learning. *arXiv preprint arXiv:2406.17255* (2024).

[11] Pantazis Deligiannis, Akash Lal, Nikita Mehrotra, and Aseem Rastogi. 2023. Fixing rust compilation errors using llms. *arXiv preprint arXiv:2308.05177* (2023).

[12] Yinlin Deng, Chun Xia, Haoran Peng, Chenyuan Yang, and Lingming Zhang. 2022. Large Language Models Are Zero-Shot Fuzzers: Fuzzing Deep-Learning Libraries via Large Language Models. *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis* (2022).

[13] Yangruibo Ding, Benjamin Steenhoek, Kexin Pei, Gail E. Kaiser, Wei Le, and Baishakhi Ray. 2023. TRACED: Execution-aware Pre-training for Source Code. *ArXiv* abs/2306.07487 (2023).

[14] Aryaz Eghbali and Michael Pradel. 2022. DynaPyt: A Dynamic Analysis Framework for Python. In *ESEC/FSE '22: 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM.

[15] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. 2020. CodeBERT: A Pre-Trained Model for Programming and Natural Languages. In *Findings of the Association for Computational Linguistics: EMNLP 2020*. 1536–1547.

[16] Ehsan Firouzi, Ashkan Sami, Foutse Khomh, and Gias Uddin. 2020. On the use of C# Unsafe Code Context: An Empirical Study of Stack Overflow. *Proceedings of the 14th ACM / IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)* (2020). https://api.semanticscholar.org/CorpusID:225047199

[17] Pedro Fonseca, Cheng Li, and Rodrigo Seromenho Miragaia Rodrigues. 2011. Finding complex concurrency bugs in large multi-threaded applications. In *European Conference on Computer Systems*. https://api.semanticscholar.org/CorpusID:1510847

[18] Akalanka Galappaththi, Sarah Nadi, and Christoph Treude. 2022. Does This Apply to Me? An Empirical Study of Technical Context in Stack Overflow. *2022 IEEE/ACM 19th International Conference on Mining Software Repositories (MSR)* (2022), 23–34. https://api.semanticscholar.org/CorpusID:247922603

[19] Zhipeng Gao, Xin Xia, John Grundy, David Lo, and Yuan-Fang Li. 2020. Generating question titles for stack overflow from mined code snippets. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 29, 4 (2020), 1–37.

[20] Zhipeng Gao, Xin Xia, David Lo, and John Grundy. 2020. Technical Q8A site answer recommendation via question boosting. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 30, 1 (2020), 1–34.

[21] Zhipeng Gao, Xin Xia, David Lo, John Grundy, and Yuan-Fang Li. 2021. Code2que: A tool for improving question titles from mined code snippets in stack overflow. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1525–1529.

[22] Zhipeng Gao, Xin Xia, David Lo, John Grundy, Xindong Zhang, and Zhenchang Xing. 2023. I know what you are searching for: Code snippet recommendation from stack overflow posts. *ACM Transactions on Software Engineering and Methodology* 32, 3 (2023), 1–42.

[23] Zhipeng Gao, Xin Xia, David Lo, John Grundy, and Thomas Zimmermann. 2021. Automating the removal of obsolete TODO comments. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 218–229.

[24] Patrice Godefroid. 2014. Micro execution. *Proceedings of the 36th International Conference on Software Engineering* (2014).

[25] Momoko Hattori, Shimpei Sawada, Shinichiro Hamaji, Masahiro Sakai, and Shunsuke Shimizu. 2020. Semi-static type, shape, and symbolic shape inference for dynamic computation graphs. *Proceedings of the 4th ACM SIGPLAN International Workshop on Machine Learning and Programming Languages* (2020). https://api.semanticscholar.org/CorpusID:219167608

[26] Edward J Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. 2021. Lora: Low-rank adaptation of large language models. *arXiv preprint arXiv:2106.09685* (2021).

[27] Xunchao Hu, Yao Cheng, Yue Duan, Andrew Henderson, and Heng Yin. 2018. Jsforce: A forced execution engine for malicious javascript detection. In *Security and Privacy in Communication Networks: 13th International Conference, SecureComm 2017, Niagara Falls, ON, Canada, October 22–25, 2017, Proceedings 13*. Springer, 704–720.

[28] Nan Jiang, Thibaud Lutellier, and Lin Tan. 2021. CURE: Code-Aware Neural Machine Translation for Automatic Program Repair. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. 1161–1173. https://doi.org/10.1109/ICSE43902.2021.00107

[29] Zu-Ming Jiang, Jia-Ju Bai, Kangjie Lu, and Shih-Min Hu. 2022. Context-Sensitive and Directional Concurrency Fuzzing for Data-Race Detection. *Proceedings 2022 Network and Distributed System Security Symposium* (2022). https://api.semanticscholar.org/CorpusID:248222066

[30] Matthew Jin, Syed Shahriar, Michele Tufano, Xin Shi, Shuai Lu, Neel Sundaresan, and Alexey Svyatkovskiy. 2023. Inferfix: End-to-end program repair with llms. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1646–1656.

[31] Harshit Joshi, José Cambronero Sanchez, Sumit Gulwani, Vu Le, Gust Verbruggen, and Ivan Radiček. 2023. Repair is nearly generation: Multilingual program repair with llms. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 37. 5131–5140.

[32] Rezwana Karim, Frank Tip, Alena Sochurkova, and Koushik Sen. 2020. Platform-Independent Dynamic Taint Analysis for JavaScript. *IEEE Transactions on Software Engineering* 46 (2020), 1364–1379. https://api.semanticscholar.org/CorpusID:69361376

[33] Kyungtae Kim, I Luk Kim, Chung Hwan Kim, Yonghwi Kwon, Yunhui Zheng, X. Zhang, and Dongyan Xu. 2017. J-Force: Forced Execution on JavaScript. *Proceedings of the 26th International Conference on World Wide Web* (2017).

[34] Owolabi Legunsen, Nader Al Awar, Xinyue Xu, Wajih Ul Hassan, Grigore Roşu, and Darko Marinov. 2019. How effective are existing Java API specifications for finding bugs during runtime verification? *Automated Software Engineering* 26 (2019), 795 – 837. https://api.semanticscholar.org/CorpusID:208190648

[35] Daniel Lehmann and Michael Pradel. 2022. Finding the Dwarf: Recovering Precise Types from WebAssembly Binaries. *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (2022).

[36] Caroline Lemieux, Jeevana Priya Inala, Shuvendu K Lahiri, and Siddhartha Sen. 2023. Codamosa: Escaping coverage plateaus in test generation with pre-trained large language models. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 919–931.

[37] Yi Li, Shaohua Wang, and Tien N. Nguyen. 2020. DLFix: context-based code transformation learning for automated program repair. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering* (Seoul, South Korea) *(ICSE '20)*. Association for Computing Machinery, New York, NY, USA, 602–614. https://doi.org/10.1145/3377811.3380345

[38] Yi Li, Shaohua Wang, and Tien N Nguyen. 2020. Improving automated program repair using two-layer tree-based neural networks. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: Companion Proceedings*. 316–317.

[39] Yi Li, Shaohua Wang, and Tien N Nguyen. 2021. Vulnerability detection with fine-grained interpretations. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 292–303.

[40] Yi Li, Shaohua Wang, and Tien N Nguyen. 2022. Dear: A novel deep learning-based approach for automated program repair. In *Proceedings of the 44th international conference on software engineering*. 511–523.

[41] Yi Li, Shaohua Wang, Tien N Nguyen, and Son Van Nguyen. 2019. Improving bug detection via context-based code representation learning and attention-based neural networks. *Proceedings of the ACM on Programming Languages* 3, OOPSLA (2019), 1–30.

[42] Yi Li, Aashish Yadavally, Jiaxing Zhang, Shaohua Wang, and Tien N Nguyen. 2023. Commit-Level, Neural Vulnerability Detection and Assessment. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1024–1036.

[43] Yi Li, Aashish Yadavally, Jiaxing Zhang, Shaohua Wang, and Tien N Nguyen. 2023. DeMinify: Neural Variable Name Recovery and Type Inference. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 758–770.

[44] Zhong Li, Minxue Pan, Yu Pei, Tian Zhang, Linzhang Wang, and Xuandong Li. 2024. Empirically revisiting and enhancing automatic classification of bug and non-bug issues. *Frontiers of Computer Science* 18, 5 (2024), 185207.

[45] Haokun Liu, Derek Tam, Mohammed Muqeeth, Jay Mohta, Tenghao Huang, Mohit Bansal, and Colin A Raffel. 2022. Few-shot parameter-efficient fine-tuning is better and cheaper than in-context learning. *Advances in Neural Information Processing Systems* 35 (2022), 1950–1965.

[46] Ping Ma, Hangyuan Cheng, Jingxuan Zhang, and Jifeng Xuan. 2020. Can this fault be detected: A study on fault detection via automated test generation. *Journal of Systems and Software* 170 (2020), 110769.

[47] Yingwei Ma, Yue Liu, Yue Yu, Yuanliang Zhang, Yu Jiang, Changjian Wang, and Shanshan Li. 2023. At Which Training Stage Does Code Data Help LLMs Reasoning? *arXiv preprint arXiv:2309.16298* (2023).

[48] Yingwei Ma, Qingping Yang, Rongyu Cao, Binhua Li, Fei Huang, and Yongbin Li. 2024. How to Understand Whole Software Repository? *arXiv preprint arXiv:2406.01422* (2024).

[49] Yubo Mai, Zhipeng Gao, Xing Hu, Lingfeng Bao, Yu Liu, and JianLing Sun. 2024. Are Human Rules Necessary? Generating Reusable APIs with CoT Reasoning and In-Context Learning. *Proceedings of the ACM on Software Engineering* 1, FSE (2024), 2355–2377.

[50] Amir M Mir, Evaldas Latoškinas, Sebastian Proksch, and Georgios Gousios. 2022. Type4Py: practical deep similarity learning-based type inference for python. In *Proceedings of the 44th International Conference on Software Engineering*. 2241–2252.

[51] Yusuke Miyazaki, Taro Sekiyama, and Atsushi Igarashi. 2018. Dynamic type inference for gradual Hindley–Milner typing. *Proceedings of the ACM on Programming Languages* 3 (2018), 1 – 29. https://api.semanticscholar.org/CorpusID:53113736

[52] Jens Nicolay, Carlos Noguera, Coen De Roover, and Wolfgang De Meuter. 2013. Determining dynamic coupling in JavaScript using object type inference. *2013 IEEE 13th International Working Conference on Source Code Analysis and Manipulation (SCAM)* (2013), 126–135. https://api.semanticscholar.org/CorpusID:609446

[53] Yoshihiro Oyama and Hirotaka Kokubo. 2023. Forced continuation of malware execution beyond exceptions. *Journal of Computer Virology and Hacking Techniques* 19, 4 (2023), 483–501.

[54] Kexin Pei, Jonas Guan, Matthew Broughton, Zhongtian Chen, Songchen Yao, David Williams-King, Vikas Ummadisetty, Junfeng Yang, Baishakhi Ray, and Suman Sekhar Jana. 2021. StateFormer: fine-grained type recovery from binaries using generative state modeling. *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (2021).

[55] Fei Peng, Zhui Deng, Xiangyu Zhang, Dongyan Xu, Zhiqiang Lin, and Zhendong Su. 2014. {X-Force}:{Force-Executing} binary programs for security applications. In *23rd USENIX Security Symposium (USENIX Security 14)*. 829–844.

[56] Yun Peng, Cuiyun Gao, Zongjie Li, Bowei Gao, David Lo, Qi reng Zhang, and Michael R. Lyu. 2021. Static Inference Meets Deep learning: A Hybrid Type Inference Approach for Python. *2022 IEEE/ACM 44th International Conference on Software Engineering (ICSE)* (2021), 2019–2030.

[57] David A. Ramos and Dawson R. Engler. 2015. Under-Constrained Symbolic Execution: Correctness Checking for Real Code. In *USENIX Annual Technical Conference*.

[58] Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, et al. 2023. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950* (2023).

[59] Max Schäfer, Sarah Nadi, Aryaz Eghbali, and Frank Tip. 2023. An empirical evaluation of using large language models for automated unit test generation. *IEEE Transactions on Software Engineering* (2023).

[60] Koushik Sen, Swaroop Kalasapur, Tasneem G. Brutch, and Simon J. Gibbs. 2013. Jalangi: a selective record-replay and dynamic analysis framework for JavaScript. In *ESEC/FSE 2013*. https://api.semanticscholar.org/CorpusID:18240724

[61] Kostya Serebryany. 2017. OSS-Fuzz - Google's continuous fuzzing service for open source software. USENIX Association, Vancouver, BC.

[62] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Audrey Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. 2016. SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis. In *IEEE Symposium on Security and Privacy*.

[63] Beatriz Souza and Michael Pradel. 2023. LExecutor: Learning-Guided Execution. *ArXiv* abs/2302.02343 (2023).

[64] Zhenhao Tang, Juan Zhai, Minxue Pan, Yousra Aafer, Shiqing Ma, Xiangyu Zhang, and Jianhua Zhao. 2018. Dual-force: Understanding webview malware via cross-language forced execution. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. 714–725.

[65] H. Trivedi, Niranjan Balasubramanian, Tushar Khot, and Ashish Sabharwal. 2022. Interleaving Retrieval with Chain-of-Thought Reasoning for Knowledge-Intensive Multi-Step Questions. In *Annual Meeting of the Association for Computational Linguistics*. https://api.semanticscholar.org/CorpusID:254877499

[66] Haoye Wang, Zhipeng Gao, Xing Hu, David Lo, John Grundy, and Xinyu Wang. 2024. Just-In-Time TODO-Missed Commits Detection. *IEEE Transactions on Software Engineering* (2024).

[67] Shaohua Wang, NhatHai Phan, Yan Wang, and Yong Zhao. 2019. Extracting API tips from developer question and answer websites. In *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*. IEEE, 321–332.

[68] Wenbo Wang, Tien N Nguyen, Shaohua Wang, Yi Li, Jiyuan Zhang, and Aashish Yadavally. 2023. DeepVD: Toward Class-Separation Features for Neural Network Vulnerability Detection. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 2249–2261.

[69] Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc V. Le, Ed H. Chi, Sharan Narang, Aakanksha Chowdhery, and Denny Zhou. 2023. Self-Consistency Improves Chain of Thought Reasoning in Language Models. In *ICLR 2023*. https://arxiv.org/abs/2203.11171

[70] Yue Wang, Weishi Wang, Shafiq R. Joty, and Steven C. H. Hoi. 2021. CodeT5: Identifier-aware Unified Pre-trained Encoder-Decoder Models for Code Understanding and Generation. *ArXiv* abs/2109.00859 (2021). https://api.semanticscholar.org/CorpusID:237386541

[71] Yue Wang, Zhide Zhou, Zhilei Ren, Dong Liu, and He Jiang. 2023. A Comprehensive Study of WebAssembly Runtime Bugs. *2023 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)* (2023), 355–366. https://api.semanticscholar.org/CorpusID:258725989

[72] Anjiang Wei, Y. Deng, Chenyuan Yang, and Lingming Zhang. 2022. Free Lunch for Testing: Fuzzing Deep-Learning Libraries from Open Source. *2022 IEEE/ACM 44th International Conference on Software Engineering (ICSE)* (2022), 995–1007.

[73] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Ed Huai hsin Chi, F. Xia, Quoc Le, and Denny Zhou. 2022. Chain of Thought Prompting Elicits Reasoning in Large Language Models. *ArXiv* abs/2201.11903 (2022). https://api.semanticscholar.org/CorpusID:246411621

[74] Xin-Cheng Wen, Xinchen Wang, Cuiyun Gao, Shaohua Wang, Yang Liu, and Zhaoquan Gu. 2023. When less is enough: Positive and unlabeled learning model for vulnerability detection. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 345–357.

[75] Chunqiu Steven Xia, Yuxiang Wei, and Lingming Zhang. 2023. Automated program repair in the era of large pre-trained language models. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 1482–1494.

[76] Chunqiu Steven Xia and Lingming Zhang. 2023. Keep the Conversation Going: Fixing 162 out of 337 bugs for $0.42 each using ChatGPT. *arXiv preprint arXiv:2304.00385* (2023).

[77] Zhaogui Xu, Peng Liu, X. Zhang, and Baowen Xu. 2016. Python predictive analysis for bug detection. *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering* (2016).

[78] Zhipeng Xue, Zhipeng Gao, Xing Hu, and Shanping Li. 2023. ACWRecommender: A Tool for Validating Actionable Warnings with Weak Supervision. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 1876–1880.

[79] Aashish Yadavally, Yi Li, Shaohua Wang, and Tien N Nguyen. 2024. A Learning-Based Approach to Static Program Slicing. *Proceedings of the ACM on Programming Languages* 8, OOPSLA1 (2024), 83–109.

[80] Aashish Yadavally, Tien N Nguyen, Wenbo Wang, and Shaohua Wang. 2023. (Partial) Program Dependence Learning. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 2501–2513.

[81] Dapeng Yan, Zhipeng Gao, and Zhiming Liu. 2023. A Closer Look at Different Difficulty Levels Code Generation Abilities of ChatGPT. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 1887–1898.

[82] Xu Yang, Shaowei Wang, Yi Li, and Shaohua Wang. 2023. Does data sampling improve deep learning-based vulnerability detection? Yeas! and Nays!. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 2287–2298.

[83] He Ye, Matias Martinez, Xiapu Luo, Tao Zhang, and Martin Monperrus. 2022. Selfapr: Self-supervised program repair with test execution diagnostics. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*. 1–13.

[84] Zhiqiang Yuan, Yiling Lou, Mingwei Liu, Shiji Ding, Kaixin Wang, Yixuan Chen, and Xin Peng. 2023. No more manual tests? evaluating and improving chatgpt for unit test generation. *arXiv preprint arXiv:2305.04207* (2023).

[85] Fengji Zhang, Bei Chen, Yue Zhang, Jacky Keung, Jin Liu, Daoguang Zan, Yi Mao, Jian-Guang Lou, and Weizhu Chen. 2023. Repocoder: Repository-level code completion through iterative retrieval and generation. *arXiv preprint arXiv:2303.12570* (2023).

[86] Qihao Zhu, Zeyu Sun, Yuan-an Xiao, Wenjie Zhang, Kang Yuan, Yingfei Xiong, and Lu Zhang. 2021. A syntax-guided edit decoder for neural program repair. In *Proceedings of the 29th ACM joint meeting on European software engineering conference and symposium on the foundations of software engineering*. 341–353.

Received 2024-04-12; accepted 2024-07-03