Understanding Programs by Exploiting (Fuzzing) Test Cases

Jianyu Zhao[*1], **Yuyang Rong**[*2], **Yiwen Guo**[†3], **Yifeng He**[2], **Hao Chen**[2]
[1]Tencent Security Big Data Lab, [2]UC Davis, [3]Independent Researcher
yjjyzhao@tencent.com, {PeterRong96, guoyiwen89}@gmal.com
{yfhe, chen}@ucdavis.edu

Abstract

Semantic understanding of programs has attracted great attention in the community. Inspired by recent successes of large language models (LLMs) in natural language understanding, tremendous progress has been made by treating programming language as another sort of natural language and training LLMs on corpora of program code. However, programs are essentially different from texts after all, in a sense that they are normally heavily structured and syntax-strict. In particular, programs and their basic units (i.e., functions and subroutines) are designed to demonstrate a variety of behaviors and/or provide possible outputs, given different inputs. The relationship between inputs and possible outputs/behaviors represents the functions/subroutines and profiles the program as a whole. Therefore, we propose to incorporate such a relationship into learning, for achieving a deeper semantic understanding of programs. To obtain inputs that are representative enough to trigger the execution of most part of the code, we resort to fuzz testing and propose fuzz tuning to boost the performance of program understanding and code representation learning, given a pre-trained LLM. The effectiveness of the proposed method is verified on two program understanding tasks including code clone detection and code classification, and it outperforms current state-of-the-arts by large margins. Code is available at https://github.com/rabbitjy/FuzzTuning.

1 Introduction

Code intelligence powered by machine learning has attracted considerable attention in both the AI and software engineering community. Particularly, code representation learning, which aims to encode functional semantics of source code, lays the foundation for achieving the intelligence and is of great interest. The learned representation can be applied to various downstream tasks, including code classification (Mou et al., 2016), code summarization (Iyer et al., 2016), clone detection (Svajlenko et al., 2014; Mou et al., 2016), etc.

Many efforts inspired by the developments of natural language understanding have been devoted to learning code representations, among which it has been increasingly popular to adopt large language models (LLMs) that are capable of learning contextual information from data at scale (Feng et al., 2020; Li et al., 2022). The LLMs can then be fine-tuned on domain-specific code to achieve superior performance compared with tradition models.

Despite being effective, these natural language processing methods do not fit perfectly for handling programs. Specifically, programs are heavily structured and syntax-strict (to be understood by compilers or interpreters), while natural language corpus is not. As basic units of programs, functions and subroutines can take a variety of argument values to demonstrate different logical behaviors or return different results. That being said, the relationship between inputs and possible outputs/behaviors essentially represents the functions/subroutines and further the whole programs.

In this paper, we propose to incorporate such a relationship into learning for a deeper understanding of programs. In fact, given enough inputs to execute all pieces of the code, then the outputs would include enough runtime information we need to profile and understand the program. However, it is nontrivial to generate a limited number of inputs that are representative enough to execute every part of the code. Without a proper strategy, we may end up with a large number of inputs that execute similar parts of codes. To address the issue, we opt to utilize fuzz testing (also known as fuzzing) (Sutton et al., 2007), which is a common software testing practice and dynamic analysis tool whose original goal is to find software bugs by executing as

[*]Equal contribution
[†]Corresponding author

Findings of the Association for Computational Linguistics: ACL 2023, pages 10667–10679
July 9-14, 2023

much code as possible. More specifically, we repurpose fuzz testing to generate input and output data for assisting code representation learning, and we demonstrate how the input and output data (i.e., test cases) can be appropriately incorporated into existing LLMs to achieve superior program understanding performance.

The contributions of this paper are three-fold. First, by recognizing the essence code representation learning, we propose to take advantage of the relationship between inputs and possible outputs for achieving a deeper understanding of programs. Second, we, for the first time, repurpose fuzz testing to assist code presentation learning, marrying these two concepts from different communities for achieving more powerful AI. Third, we obtain state-of-the-art results on typical program understanding tasks including clone detection and code classification, in comparison to prior arts.

2 Related Work

In this section, we introduce related work on code understanding (from the natural language understanding community) and fuzzing (from the software engineering community).

2.1 Code Representation Learning

Inspired by the success of LLMs in natural language processing (Raffel et al., 2020; Liu et al., 2019; Devlin et al., 2018), LLMs trained on programming languages have also been widely used to drive code intelligence. For instance, Kanade et al. (2020) proposed cuBERT to train BERT models on a curated and deduplicated corpus of 7.4M Python files from GitHub, and adapt the pre-trained models to various code classification tasks and a program repair task. Thereafter, a bunch of methods have been developed and a variety of LLMs have been trained on code data, including CodeBERT (Feng et al., 2020), CodeT5 (Wang et al., 2021c), and CodeGPT (Lu et al., 2021).

The importance of comprehending syntax and structures for learning code representations has also been pointed out by several prior arts, and methods that incorporate programming-language-specified features, including abstract syntax tree (Tipirneni et al., 2022; Guo et al., 2022), control or data flow graphs (Guo et al., 2020), and intermediate representation of code (Peng et al., 2021) have been developed. These methods only utilize information available for static analysis. It is generally difficult for static analysis to be both safe and sound (Taneja et al., 2020) when analyzing the behavior of programs. For instance, a path that exists on the control flow graph may never be executed due to data-flow limitations. Our work is the first to take dynamic program information (by generating and exploiting test cases with inputs and outputs) into account for code representation learning. An input will lead to execution of part of the program and an output (or some behaviors if no output is required), which would reflect the functionality of that part of the program. We hypothesize that if we have enough inputs to execute the code sufficiently, the outputs would also include enough runtime information that we need to profile the program.

2.2 Fuzzing

Fuzz testing, or fuzzing, is a process that tests the correctness of programs. Fuzzing can be roughly considered as a four-stage loop. First the program is executed with a given input. Second, the behavior of the program is monitored to determine if any new behavior is triggered. Third, if a new behavior is present, the corresponding input will be saved into a store, otherwise, the input is discarded as not interesting. Finally, a mutator takes a saved input in the store, mutates it in different fashions and sends the input for another round of execution.

American Fuzzy Lop (AFL)[1] is the first fuzzer to implement behavior monitoring using branch coverage. It tracks which edges of the control flow graph have been executed. Since the invention of AFL, many innovations have been made to improve the overall fuzzing performance. Wang et al. (2021a); Gan et al. (2018) modified branch coverage to lower the overhead while improving tracking sensitivity. Böhme et al. (2019) proposed that power scheduling is better than a first-in-first-out queue for input store and improved the fuzzing performance by a magnitude. Chen and Chen (2018) introduced new mutation algorithms and showed superior performance than random mutation. Many of the changes have been incorporated into a more modern tool called AFL++ (Fioraldi et al., 2020).

Unfortunately, current use of fuzzers only focuses on the bugs in the software (Chen et al., 2018; Rong et al., 2020; Aschermann et al., 2019) and did not show possibility of adopting fuzzing results in code representation learning for AI. We identify that these results can be used to profile programs

[1] https://lcamtuf.coredump.cx/afl/

and improve the performance of code representation learning and program understanding.

3 Method

As mentioned in Section 1, programs show strict syntax. To inspire deeper understanding of the syntax and logical behaviors of a program or functions/subroutines (which are the building blocks of the program), we attempt to exploit the relationship between their inputs and possible outputs/behaviors for achieving improved understanding of programs and code, akin to how engineers understands third-party code.

However, with existing learning techniques, it seems nontrivial to generate inputs that could lead to execution of sufficient part of the code, thus we resort to fuzzing to achieve this goal.

3.1 Fuzzing for Obtaining Inputs (and Outputs)

Despite being widely adopted for testing software, fuzzing has rarely been adopted in machine learning tasks. In general, fuzzing is a software testing practice, whose goal is to find software bugs by executing as much code as possible. To achieve this, it executes the program with different inputs and monitors the behavior of each execution. Therefore, as byproducts of fuzzing, a large number of inputs may be produced by a fuzzer, each triggering a new behavior of the program under test.

Fuzzing is programming language agnostic in general. However, with only source code, we have to compile the programs into executable files for fuzzing. We mainly describe details for four mainstream languages (C, C++, Java, and Python), and a tool was specifically designed to build the programs for fuzzing. This tool interacts with the compiler or interpreter to automatically fix some problems that prevent it from being fuzzed. Since the main aim of this work is to assist models to better understand programs, we fix problems that do not affect the semantics and functionality of code but prevent fuzzing.

For C and C++, we treat them as C++ files. Some semantics-irrelevant errors in the program would prevent the code from compiling and fuzzing. For example, missing headers, absent return of a `main` function that is defined to have one, and misuse of reserved keyword. In order to fix these compilation errors, we designed a compiler plugin that can automatically fix these. First we run the lexer and parser on the program to gain abstract syntax tree, which would make code transformation much easier. Then we designed a parser to parse error message from the compiler. We introduce several fixes to correct the program for different errors.

1. **Missing headers.** We included most commonly used headers in the C++ library at the head of each program.

2. **Incorrect return type and/or arguments.** For instance, if a `main` function is defined as "`int main()`" but provides no return, we fixed it by added "`return 0;`" to the end of the program.

3. **Misuse of keywords in the standard library.** Reserved keywords might be misused as variables and we added a "`fixed_`" prefix to each of such variables to avoid keyword violation.

4. **Incorrect `struct` definition.** Many structures were defined without a semicolon after it, we will append that semicolon.

5. **Undeclared identifier.** We notice that many programs use static values as a constant value, yet the value is sometimes missing. We would analyze the usage of the constants and insert definitions for them.

For Java programs, we compiled them into bytecodes using Kelinci (Kersten et al., 2017) to instrument them for fuzzing. Not all programs we tested were named `Main.java` but they all defined a `Main` class. In order to compile them, we changed the `Main` class to its file name in order to compile it. For each program, a TCP server was added to communicate with a fork server which then sends data to the fuzzer.

Python is the most difficult language. First many lexical errors are not as easy to fix as C/C++ and Java. For example, if the program mixed tabs and spaces, it is hard to infer what is the intended indention. To solve this, we used `autopep8`[2] to transform the program. The next challenge is that Python2 and Python3 can't be easily distinguished, therefore, it is unclear which interpreter should be used for fuzzing. To detect the version, also to verify the correctness of the transformation, we treated all code as Python3 in the first place and try to compile python program to bytecode using

[2] https://pypi.org/project/autopep8/

py_compile. If the compilation failed, then it was probably a Python2 implementation and we tried to convert it to Python3 using 2to3[3]. Finally, we had to instrument the behavior monitoring and reporting to communicate with the fuzzer. We used py-afl-fuzz[4] to achieve this.

We want to point out that all the changes made in this section are for fuzzing only. When training models in the following sections, the programs remain unchanged.

We selected AFL++ (Fioraldi et al., 2020) as our fuzzer and fuzzed all experimental data on a server with 2 20-core 40-thread x86_64 CPUs and 692GB of memory. Each fuzzer only has one thread and ran until it exhausts all paths or a K-minute timeout is triggered. The stored inputs that are of interest to AFL++ can then be utilized to execute the program and obtain outputs, and they constitute the fuzzing test cases. The test cases (i.e., pairs of inputs and outputs) were produced in bytes and we may decode it into human readable strings.

3.2 Model

Although it is possible to train representation learning models from scratch using the obtained fuzzing test cases, it can be more effective to take advantage of previous pre-training effort. In particular, given a pre-trained LLM, we attempt to take these test cases as model inputs somehow. Considering that the LLM was mostly trained on programming language and natural language corpus (Feng et al., 2020; Wang et al., 2021c; Guo et al., 2022), the source code of the program is fed into the model together with fuzzing test cases, by concatenating the two parts.

3.3 Prompting

The fuzzing test cases in their raw format are a series of bytes, and, by decoding, we can obtain a series of Unicode strings which are unorganized. To help LLMs better understand these test cases, we introduce cloze prompts (Petroni et al., 2019). Prompt have shown significant power in natural language processing since the invention of LLMs. Considering that LLMs can be pre-trained on both natural language corpora and programming language corpora, we design both natural-language-based prompts and programming-language-based prompts for each pair of input (denoted by [INPUT]) and output (denoted by [OUTPUT]) as follows:

[3] https://docs.python.org/3/library/2to3.html
[4] https://pypi.org/project/python-afl/

Dataset	# of problems	# of programs
POJ104	104	52K
C++1000	1000	500K
Python800	800	240K
Java250	250	75K

Table 1: Dataset statistics.

1. **Natural-language-based prompt:**

   (a) [SEP] + "input: " + [INPUT] + "," + "output: " + [OUTPUT];

   (b) [SEP] + "input is " + [INPUT] + "and" + "output is " + [OUTPUT];

2. **Programming-language-based prompt:**

   (a) [SEP] + "cin>>" + [INPUT] + ";" + "cout <<" + [OUTPUT]; (For C/C++)

   (b) [SEP] + "System.in " + [INPUT] + ";" + "System.out" + [OUTPUT]; (For Java)

   (c) [SEP] + "input()" + [INPUT] + "\n" + "print" + [OUTPUT]; (For Python)

In experiments, we found that the programming-language-based prompts are more effective and we will stick with it in the sequel of this paper, if not specified. This is unsurprising since the fuzzing test cases can stay in harmony with the source code with such a prompt.

Each prompted pair of input and output can be concatenated together before being further concatenated with the source code. Pre-trained LLMs can be tuned on downstream datasets with their inputs being modified to consider both the source code and fuzzing test cases. We call this method *fuzz tuning* in the paper.

4 Experimental Results

In this section, we report experimental results to verify the effectiveness of our fuzz tuning. We consider popular tasks (i.e., clone detection and code classification) and datasets involving mainstream languages including C, C++, Java, and Python. Experiments were performed on NVIDIA V100 GPUs using PyTorch 1.7.0 (Paszke et al., 2019) implementations.

Datasets. Our experiments were performed mainly on two datasets, i.e., POJ104 (Mou et al., 2016) and CodeNet (Puri et al., 2021). POJ104 has been incorporated into CodeXGlue (Lu et al., 2021) and is widely used. It consists of 104 problems, each containing 500 C/C++ implementations. CodeNet is a recently proposed large-scale dataset

Method	Java250	Python800	C++1000*
Rule-based w/SPT(AROMA) (Puri et al., 2021)	19.00	19.00	-
GNN w/SPT(MISIM) (Puri et al., 2021)	64.00	65.00	-
CodeBERT+FineT (Feng et al., 2020)	81.47	83.23	44.94
UniXcoder+FineT (Guo et al., 2022)	84.35	85.00	49.75
CodeBERT+FuzzT (ours)	**83.39**	**85.64**	**54.92**
UniXcoder+FuzzT (ours)	**86.74**	**86.01**	**60.21**

Table 2: Clone detection results on CodeNet. Compared with normal fine-tuning (FineT), our fuzz tuning (FuzzT) leads to significant improvements and new state-of-the-arts. C++1000* contains 16% of all problems, which is a roughly 6.3x downsample of the original dataset (see Table 8 for results on other scales). Bold stats are better.

Method	MAP@R
CodeBERT+FineT (Feng et al., 2020)	84.29
GraphCodeBERT+FineT (Guo et al., 2020)	85.16
PLBART+FineT (Ahmad et al., 2021)	86.27
SYNCOBERT+FineT (Wang et al., 2021b)	88.24
CodeT5+FineT (Wang et al., 2021c)	88.65
ContraBERT+FineT (Liu et al., 2023)	90.46
UniXcoder+FineT (Guo et al., 2022)	90.52
CodeBERT+FuzzT (ours)	**92.01**
UniXcoder+FuzzT (ours)	**93.40**

Table 3: Clone detection results on POJ104. Our fuzz tuning (FuzzT) leads to state-of-the-art results. Bold stats are better.

for AI for code applications, and it contains programs written in C++, Java, and Python. In particular, it has four subsets for these languages: Java250, Python800, C++1000, and C++1400. We chose Java250, Python800, and C++1000 for experiments, which cover all the three languages in CodeNet. Java250 consists of 250 problems where each includes 300 Java programs, and Python800 consists of 800 problems where each includes 300 Python programs. C++1000 consists of 1000 problems where each includes 500 C++ programs, and it is mainly used to verify the effectiveness of our method over various training scales (i.e., our fuzz tuning will be performed on various subsampling ratios of the set) in this paper. See Table 1 for a summarization of key information of all datasets.

Pre-trained LLMs. To make our experiments more comprehensive, our fuzz tuning (FuzzT) was tested on two different LLMs: CodeBERT (Feng et al., 2020) and UniXcoder (Guo et al., 2022) that were pre-trained on both natural languages and programming languages.

Obtaining test cases. We set $K = 5$ for the fuzzer. In POJ104, 90.3% of all fuzzed programs quits before timeout, which justifies our decision. Taking advantage of our effort in Section 3.1, all datasets have more than 95% of the programs compiled / validated, and all of them have more than 90% of programs fuzzed.

4.1 Clone Detection Results

The clone detection task aims to measure the similarity between two code snippets or two programs, which can help reduce bugs and prevent the loss of intellectual property.

Given a code snippet or the source code of a program as a query, models should detect semantically similar implementations on the test set. POJ104 is adopted as the default dataset for code clone detection on CodeXGlue, thus we did experiment on it first. We followed previous work (Lu et al., 2021) and used a 64/16/24 split. That said, training was performed on 64 problems while validation and test were performed on other 16 and 24 problems, respectively. Besides, We further experimented on CodeNet which shows a larger data scale and variety. Java250, Python800, and C++1000 were used, and we followed Puri et al. (2021) which used a 50%/25%/25% split for training/validation/test for all these concerned sets. C++1000 is mainly used to test our method over various training scales in Section 4.4 with the full test set, and we will only discuss results on the smallest subsampling ratio (which is roughly 6.3x, achieving by randomly selecting 16% of the problems for experiments) in this subsection. We denoted such a subsampled set as C++1000*. All models tested here were tuned on a single V100, for no longer than 8 GPU hours.

Results were evaluated using the mean average precision@R (MAP@R) (Musgrave et al., 2020). To train our models, we followed previous work (Lu et al., 2021) and directly set the learning rate, batch size, and maximal sequence length (for code tokens) to 2e-5, 8, and 400, respectively. We used the Adam optimizer (Kingma and Ba, 2014) to fine-tune each pre-trained model for 2 epochs. The best model on the validation set is selected to test. We adopted the same hyper-parameters on

Method	Java250	Python800	C++1000†
GIN (Puri et al., 2021)	6.74%	5.83%	-
CodeGraph+GCN (Puri et al., 2021)	5.90%	12.2%	-
C-BERT+FineT (Puri et al., 2021)	2.60%	2.91%	-
CodeBERT+FineT (Feng et al., 2020)	2.37%	2.19%	16.34%
UniXcoder+FineT (Guo et al., 2022)	2.02%	1.95%	14.57%
CodeBERT+FuzzT (ours)	**1.77%**	**1.61%**	**7.63%**
UniXcoder+FuzzT (ours)	**1.56%**	**1.28%**	**6.18%**

Table 4: Code classification results on CodeNet. Our fuzz tuning (FuzzT) leads to new state-of-the-arts. C++1000† contains 40% of all problems, which is a 2.5x downsample of the original dataset (see Table 9 for results on other data scales). Bold stats are better.

Method	Error Rate
TBCNN (Mou et al., 2016)	6.00%
ProGraML (Cummins et al., 2020)	3.38%
OSCAR (Peng et al., 2021)	1.92%
CodeBERT+FineT (Feng et al., 2020)	1.61%
UniXcoder+FineT (Guo et al., 2022)	1.61%
CodeBERT+FuzzT (ours)	**1.40%**
UniXcoder+FuzzT (ours)	**1.38%**

Table 5: Code classification results on POJ104. Our fuzz tuning (FuzzT) leads to new state-of-the-arts. Bold stats are better.

both POJ104 and CodeNet.

Table 3 provides the results on POJ104. It is obvious that, when the proposed fuzz tuning is applied, we obtain significant performance gains with both CodeBERT and UniXcoder. More specifically, comparing with the normal fine-turning (FineT) method, we obtained **+7.72%** and **+2.88%** absolute gains in MAP@R with CodeBERT and UniXcoder, respectively. Such practical improvements clearly demonstrate the benefits of incorporating fuzzing test cases into program understanding. In addition, fuzz tuning obtained models (i.e., CodeBERT+FuzzT and UniXcoder+FuzzT) outperform all other state-of-the-art models significantly, leading to obvious empirical superiority (i.e., **+2.94%**) comparing even with a very recent winning solution on the CodeXGLUE benchmark [5] called ContraBERT.

Table 2 demonstrates the results on CodeNet. Apparently, on CodeNet, our fuzz tuning also leads to significant performance gains on CodeBERT and UniXcoder, when compared with FineT.

[5] https://microsoft.github.io/CodeXGLUE/

4.2 Code Classification Results

The concerned code classification task (Mou et al., 2016) requires that we assign the same label to programs that were implemented to solve the same problem and achieve the same goal. The experiments were also performed on POJ104 and CodeNet, where each unique problem is considered as a single class. For POJ104, we adopted the same dataset split as in Peng et al. (2021)'s work, and, for CodeNet, we kept the official data split (Puri et al., 2021). As previously mentioned, C++1000 in CodeNet is mainly used to test our method over various training scales in Section 4.4, and we will only discuss results on the smallest subsampling ratio (which is 2.5x, achieving by randomly selecting 40% of the programs for experiments) here. We denoted such a subsampled tuning set as C++1000†. We followed the hyper-parameter setting of CodeBERT in defect detection to set a learning rate of 2e-5, a training batch size of 32, and a maximal sequence length (for code tokens) of 400. We tuned pre-trained LLMs for 10 epochs and selected the models that performed the best on the validation set and report their results on the test sets. Error rate of different methods are reported for comparison. All our code classification models were were tuned on two V100, for no longer than 8 hours.

Table 5 and Table 4 summarize the results. Similarly, We observe that our fuzz tuning bring significant improvement, comparing with the normal fine-tuning (FineT) method, it leads to **+0.21%** and **+0.23%** absolute performance gain in reducing the error rate with CodeBERT and UniXcoder, respectively, on POJ104. Fuzz tuning obtained models (i.e, CodeBERT+FuzzT and UniXcoder+FuzzT) also outperform all previous models on this task on POJ104, leading to new state-of-the-arts. The same conclusion can also be drawn on CodeNet,

Decoding	CD	CC
CodeBERT+FineT	84.29	1.61%
CodeBERT+FuzzT		
-in bytes	84.21	1.55%
-in UTF-8 string	**92.01**	**1.40%**

Table 6: Comparing using raw and decoded fuzzing test cases in tuning clone detection (CD) and code classification (CC) models on POJ104. MAP@R and the error rate are evaluated for the two tasks, respectively. Bold stats are better.

Prompt Type	CD	CC
CodeBERT+FineT	84.29	1.61%
CodeBERT+FuzzT		
-w/o prompt	89.36	1.51%
-NL prompt, type (a)	91.14	1.54%
-NL prompt, type (b)	91.59	1.58%
-PL prompt, for C/C++	**92.01**	**1.40%**

Table 7: Comparing different prompts for our fuzz tuning on the clone detection (CD) and code classification (CC) tasks on POJ104. Bold stats are better.

showing that the effectiveness of our method hold on various programming languages.

The results on both clone detection and code classification demonstrate the effectiveness of our fuzz tuning. Both tasks requires the model to understand not only the structure of the code, but further the semantics, which is hard to acquire by simply looking at the code. Yet, provided with inputs and outputs, the model can excel. We contribute this accuracy gain to program profiling provided through fuzzing. These profiles include essential dynamic information that isn't used by any other models.

4.3 Ablation Study

In this subsection, we investigate impacting factors in our method: including the quality of test cases, decoding, and prompting.

Random cases vs fuzzing cases. Given the success of fuzz tuning in clone detection and code classification, the effectiveness of incorporating test cases can be recognized. One may expect that random input generator can work to some extent, for providing test cases. Unfortunately, our evaluation shows otherwise. We tried following this idea and crafted around 2000 inputs for each program, yet none of them is valid and understandable to the program. This result is expected, since the chance of a byte being a digit is only 10/256, there is less than 1% change of generating a 3-digit number. Thus, it is reasonable to conclude that random input generator is prone to generating invalid inputs, which lead to crash and hang of the program and cannot be used to profile it. By contrast, our fuzzer provides behavior monitoring, all these ineffective inputs are filtered and not reported in the first place.

Method	4% (↓25x)	8% (↓12.5x)	16% (↓∼6.3x)
CodeBERT+FineT	26.92	36.79	44.94
CodeBERT+FuzzT	**30.66**	**40.57**	**54.92**
UniXcoder+FineT	34.71	43.06	49.75
UniXcoder+FuzzT	**42.53**	**51.83**	**60.21**

Table 8: How different methods scale with the size of training/tuning dataset on the C++1000 *clone detection* task. Bold stats are better.

Decoding. As mentioned in Section 3.1, the fuzzer processes obtained inputs as a series of bytes. We argue that reading test cases as bytes will cause severe performance degradation, since LLMs are pre-trained using human-readable codes and natural languages, which explains why we decode the obtained bytes before feeding them to LLMs. To verify the effectiveness of decoding, we compare using human-readable UTF-8 strings and those raw bytes, both with cloze prompts, for program understanding. The experiment was conducted on the POJ104 clone detection task and the POJ104 code classification task. Table 6 shows the results. Apparently, human-readable test cases perform much better than bytes-format ones on both two tasks.

Prompting. We then compare the performance of fuzz tuning with and without prompting. Table 7 demonstrates the results. For prompting, two types of natural-language-based (NL-based) prompts and the advocated programming-language-based (PL-based) prompt are tested. Apparently, prompting is beneficial. As has been mentioned in Section 3, the PL-based prompt outperforms the two types of NL-based prompts. It shows a **+2.65%** absolute gain on the POJ104 clone detection task and a **+0.11%** absolute gain on code classification, compared with an implementation of fuzz tuning without prompts. For clone detection, prompting is always effective, no matter it is NL-based or PL-based, while, for code classification, the NL-based prompts fail.

4.4 Data Scale

We then investigate whether our fuzz tuning is effective on various training data scales. To achieve this, we subsampled from C++1000 in CodeNet to construct data sets of various scales to perform fuzz tuning. The official split of C++1000 was considered to construct test sets (Puri et al., 2021), and

Method	10% (↓10x)	20% (↓5x)	40% (↓2.5x)
CodeBERT+FineT	34.38%	19.15%	16.34%
CodeBERT+FuzzT	**30.90%**	**12.53%**	**7.63%**
UniXcoder+FineT	21.66%	16.24%	14.57%
UniXcoder+FuzzT	**14.48%**	**8.39%**	**6.18%**

Table 9: How different methods scale with the size of training/tuning dataset on the C++1000 *code classification* task. Bold stats are better.

the same test sets were adopted for testing models obtained on all these training scales. For clone detection, we sampled 4%, 8%, and 16% of the training and validation problems (i.e., subsampled the training and validation set by 25x, 12.5x, and roughly 6.3x). For the code classification task, we sampled 10%, 20% and 40% (i.e., subsampled by 10x, 5x, and 2.5x) of the training and validation and keep the sample ratio between the two sets as 4:1. We adopted the same experimental settings as in Section 4.1 and Section 4.2.

Results of different fuzz tuning scales are provided in Table 8 and Table 9. Apparently, our fuzz tuning is effective on all these training scales. In particular, for clone detection, when only 4% of the data is used for training, normal fine-tuning of CodeBERT and UniXcoder shows an MAP@R of only 26.92 and 34.71, respectively, while, by introducing fuzz tuning, we can achieve 30.66 and 42.53, respectively, showing even more obvious superiority than with 16% of the data. It is also possible for both fine-tuning and fuzz tuning to scale to more than 16% of the data, yet it requires more than 10 epochs to reach their performance plateaus and weaken the necessity of pre-training, thus we will leave it to future work for exploration. The same conclusion can be drawn for code classification.

4.5 Case Study

In this section, we extract some real cases in the concerned dataset (i.e., POJ104) to show how our fuzz tuning works. Figure 1 reports the achieved per-program MAP@R and the performance gap between FuzzT and FineT on the POJ104 test set, *with CodeBERT*. We see that FuzzT outperforms FineT on 17 out of the 24 test problems.

Figure 1 demonstrates that using the normal fine-tuning leads to very low MAP@R on Problem 103 of POJ104 [6], yet our fuzz tuning more than doubled the score. Although POJ104 does not describe each problem in detail, we did some investigations and

[6]Note that Problem 1-80 are training and validation problems, and Problem 81-104 are test problems.

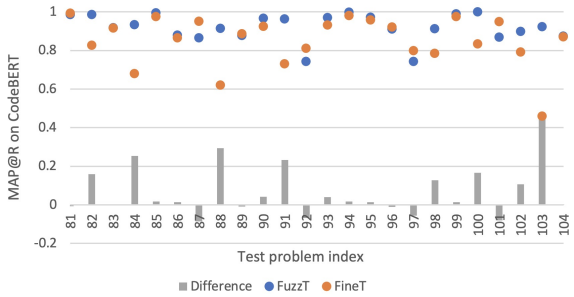

Figure 1: Per-problem clone detection performance on the POJ104 test set, using *CoderBERT+FineT* or *CoderBERT+FuzzT*. The horizontal axis shows the ID of the POJ104 problems, and the vertical axis is the MAP@R.

conjecture that this particular problem is asking how many identical consecutive letters are there in a given string, if letter case is ignored. Our investigations show that many programmers all unifies the letter case in the string first, but they disagree on whether to use uppercase letters or lowercase letter sand disagree on how to achieve this, leading to different implementations including utilizing standard library calls (i.e., to convert each character "c" using "`toupper(c)`"), calculating offset by casting (i.e., implementing something like `c-'A'+'a'`), and static mapping (i.e., using "`caseMap[c]`"). This will pose challenges to models for understanding their functionality, if fine-tuning on source code only. One may expect this particular problem to be addressed by pre-training on relevant data or by taking more advantage of static information of programs. This is possible, since for other pre-trained LLMs, Problem 103 may not be the most challenging one. However, other issues similarly exist, e.g., UniXcoder+FineT shows its worst performance on Problem 88, as can be seen in Figure 2.

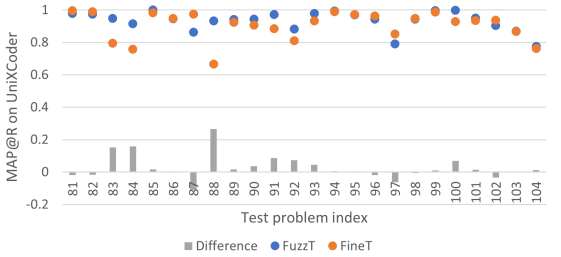

Figure 2: Per-problem clone detection performance on the POJ104 test set, using *UniXcoder+FineT* or *UniXcoder+FuzzT*. The horizontal axis shows the ID of the POJ104 problems, and the vertical axis is the MAP@R.

By contrast, since the programs achieve the same goal, the test cases can help convey that information to the model. This further demonstrates our idea and explains the effectiveness of our fuzz tuning.

5 Conclusion

In this paper, we have pointed out that exploiting informative test cases helps to understanding of programs. We have developed fuzz tuning, a novel method which takes advantage of fuzzing together with prior large-scale pre-training effort to achieve this goal. Our fuzz tuning repurposes traditional fuzzers to generate informative test cases that well-represent the functionality of programs and it introduces appropriate cloze prompts to incorporate the test cases into being processed. By performing comprehensive experiments on two datasets and two program understanding tasks, we have verified the effectiveness of the method and achieved new state-of-the-arts.

Limitations

Fuzzers are designed to reach deep and complex control flow in large software. Many programs for current AI for code datasets do not have complex control flow. As a result, AFL++ can quickly cover all program branches before generating many inputs for us to feed to the model. We plan to try data-flow coverage as a more accurate coverage metric in the future.

AFL++ uses branch coverage to track fuzzing progress. Although it works well on C/C++ programs, it may be ineffective on languages with exceptions, which are implicit control flow. For example, AFL++ cannot distinguish different exceptions thrown in the same block, which sometimes leads to low coverage in Python programs. To overcome this issue, one possible way is to change from branch coverage to line coverage.

Although our current implementation requires a fuzzer, our approach can also work on tasks with only functions or code snippets as long as we can acquire adequate input/output pairs of the functions or code snippets, which may have some engineering challenges but is not infeasible. For example, in recent years, the software engineering community has proposed various ways to fuzz bare functions (Serebryany, 2016; Ispoglou et al., 2020).

Ethical Consideration

Our method exploits fuzzing test cases for program understanding. Improved semantic understanding of programs facilitates various tasks, e.g., code generation and code completion, which might further be used to patch vulnerabilities or fix defects of softwares and systems. Nevertheless, considerable effort has to be further devoted to apply the method to these applications, for which we encourage to take special care in advance. In addition, a number of crashes and hangs have been observed on programs in the adopted datasets, since fuzz testing is utilized. We do not demonstrate test cases that lead to these crashes and hangs to avoid misuse of this information.

Acknowledgment

This material is partially based upon work supported by the National Science Foundation under Grant No. 1801751 and 1956364.

References

Wasi Uddin Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2021. Unified pre-training for program understanding and generation. *arXiv preprint arXiv:2103.06333*.

Cornelius Aschermann, Tommaso Frassetto, Thorsten Holz, Patrick Jauernig, Ahmad-Reza Sadeghi, and Daniel Teuchert. 2019. Nautilus: Fishing for deep bugs with grammars. In *NDSS*.

M. Böhme, V. Pham, and A. Roychoudhury. 2019. Coverage-based greybox fuzzing as markov chain. *IEEE Transactions on Software Engineering*, 45(5):489–506.

Jiongyi Chen, Wenrui Diao, Qingchuan Zhao, Chaoshun Zuo, Zhiqiang Lin, XiaoFeng Wang, Wing Cheong Lau, Menghan Sun, Ronghai Yang, and Kehuan Zhang. 2018. Iotfuzzer: Discovering memory corruptions in iot through app-based fuzzing. In *NDSS*.

Peng Chen and Hao Chen. 2018. Angora: Efficient fuzzing by principled search. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 711–725.

Chris Cummins, Zacharias V Fisches, Tal Ben-Nun, Torsten Hoefler, and Hugh Leather. 2020. Programl: Graph-based deep learning for program optimization and analysis. *arXiv preprint arXiv:2003.10536*.

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*.

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. *arXiv preprint arXiv:2002.08155*.

Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. 2020. AFL++ : Combining incremental steps of fuzzing research. In *14th USENIX Workshop*

on Offensive Technologies (WOOT 20). USENIX Association.

Shuitao Gan, Chao Zhang, Xiaojun Qin, Xuwen Tu, Kang Li, Zhongyu Pei, and Zuoning Chen. 2018. Collafl: Path sensitive fuzzing. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 679–696.

Daya Guo, Shuai Lu, Nan Duan, Yanlin Wang, Ming Zhou, and Jian Yin. 2022. Unixcoder: Unified cross-modal pre-training for code representation. *arXiv preprint arXiv:2203.03850*.

Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, et al. 2020. Graphcode-bert: Pre-training code representations with data flow. *arXiv preprint arXiv:2009.08366*.

Kyriakos Ispoglou, Daniel Austin, Vishwath Mohan, and Mathias Payer. 2020. {FuzzGen}: Automatic fuzzer generation. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 2271–2287.

Srinivasan Iyer, Ioannis Konstas, Alvin Cheung, and Luke Zettlemoyer. 2016. Summarizing source code using a neural attention model. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 2073–2083.

Aditya Kanade, Petros Maniatis, Gogul Balakrishnan, and Kensen Shi. 2020. Learning and evaluating contextual embedding of source code. In *International Conference on Machine Learning*, pages 5110–5121. PMLR.

Rody Kersten, Kasper Luckow, and Corina S. Păsăreanu. 2017. Poster: Afl-based fuzzing for java with kelinci. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, page 2511–2513, New York, NY, USA. Association for Computing Machinery.

Diederik P Kingma and Jimmy Ba. 2014. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*.

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.

Shangqing Liu, Bozhi Wu, Xiaofei Xie, Guozhu Meng, and Yang Liu. 2023. Contrabert: Enhancing code pre-trained models via contrastive learning. *arXiv preprint arXiv:2301.09072*.

Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. 2019. Roberta: A robustly optimized bert pretraining approach. *arXiv preprint arXiv:1907.11692*.

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*.

Lili Mou, Ge Li, Lu Zhang, Tao Wang, and Zhi Jin. 2016. Convolutional neural networks over tree structures for programming language processing. In *Thirtieth AAAI conference on artificial intelligence*.

Kevin Musgrave, Serge Belongie, and Ser-Nam Lim. 2020. A metric learning reality check. In *European Conference on Computer Vision*, pages 681–699. Springer.

Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. 2019. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems*, 32.

Dinglan Peng, Shuxin Zheng, Yatao Li, Guolin Ke, Di He, and Tie-Yan Liu. 2021. How could neural networks understand programs? In *International Conference on Machine Learning*, pages 8476–8486. PMLR.

Fabio Petroni, Tim Rocktäschel, Patrick Lewis, Anton Bakhtin, Yuxiang Wu, Alexander H Miller, and Sebastian Riedel. 2019. Language models as knowledge bases? *arXiv preprint arXiv:1909.01066*.

Ruchir Puri, David S Kung, Geert Janssen, Wei Zhang, Giacomo Domeniconi, Vladmir Zolotov, Julian Dolby, Jie Chen, Mihir Choudhury, Lindsey Decker, et al. 2021. Project codenet: A large-scale ai for code dataset for learning a diversity of coding tasks. *arXiv preprint arXiv:2105.12655*, 1035.

Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, Peter J Liu, et al. 2020. Exploring the limits of transfer learning with a unified text-to-text transformer. *Journal of Machine Learning Research*, 21(140):1–67.

Yuyang Rong, Peng Chen, and Hao Chen. 2020. Integrity: Finding integer errors by targeted fuzzing. In *International Conference on Security and Privacy in Communication Systems*, pages 360–380. Springer.

Kosta Serebryany. 2016. Continuous fuzzing with libfuzzer and addresssanitizer. In *2016 IEEE Cybersecurity Development (SecDev)*, pages 157–157. IEEE.

Michael Sutton, Adam Greene, and Pedram Amini. 2007. *Fuzzing: brute force vulnerability discovery*. Pearson Education.

Jeffrey Svajlenko, Judith F Islam, Iman Keivanloo, Chanchal K Roy, and Mohammad Mamun Mia. 2014. Towards a big data curated benchmark of inter-project code clones. In *2014 IEEE International Conference*

on Software Maintenance and Evolution, pages 476–480. IEEE.

Jubi Taneja, Zhengyang Liu, and John Regehr. 2020. Testing static analyses for precision and soundness. In *Proceedings of the 18th ACM/IEEE International Symposium on Code Generation and Optimization*, CGO 2020, page 81–93, New York, NY, USA. Association for Computing Machinery.

Sindhu Tipirneni, Ming Zhu, and Chandan K Reddy. 2022. Structcoder: Structure-aware transformer for code generation. *arXiv preprint arXiv:2206.05239*.

Jinghan Wang, Chengyu Song, and Heng Yin. 2021a. Reinforcement learning-based hierarchical seed scheduling for greybox fuzzing. In *Network and Distributed System Security Symposium*.

Xin Wang, Yasheng Wang, Fei Mi, Pingyi Zhou, Yao Wan, Xiao Liu, Li Li, Hao Wu, Jin Liu, and Xin Jiang. 2021b. Syncobert: Syntax-guided multi-modal contrastive pre-training for code representation. *arXiv preprint arXiv:2108.04556*.

Yue Wang, Weishi Wang, Shafiq Joty, and Steven CH Hoi. 2021c. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. *arXiv preprint arXiv:2109.00859*.

ACL 2023 Responsible NLP Checklist

A For every submission:

☑ A1. Did you describe the limitations of your work?
section Limitations

☑ A2. Did you discuss any potential risks of your work?
section Ethical consideration

☑ A3. Do the abstract and introduction summarize the paper's main claims?
Section abstract and section 1

☒ A4. Have you used AI writing assistants when working on this paper?
Left blank.

B ☒ Did you use or create scientific artifacts?

Left blank.

☐ B1. Did you cite the creators of artifacts you used?
No response.

☐ B2. Did you discuss the license or terms for use and / or distribution of any artifacts?
No response.

☐ B3. Did you discuss if your use of existing artifact(s) was consistent with their intended use, provided that it was specified? For the artifacts you create, do you specify intended use and whether that is compatible with the original access conditions (in particular, derivatives of data accessed for research purposes should not be used outside of research contexts)?
No response.

☐ B4. Did you discuss the steps taken to check whether the data that was collected / used contains any information that names or uniquely identifies individual people or offensive content, and the steps taken to protect / anonymize it?
No response.

☐ B5. Did you provide documentation of the artifacts, e.g., coverage of domains, languages, and linguistic phenomena, demographic groups represented, etc.?
No response.

☐ B6. Did you report relevant statistics like the number of examples, details of train / test / dev splits, etc. for the data that you used / created? Even for commonly-used benchmark datasets, include the number of examples in train / validation / test splits, as these provide necessary context for a reader to understand experimental results. For example, small differences in accuracy on large test sets may be significant, while on small test sets they may not be.
No response.

C ☑ Did you run computational experiments?

section 4

☑ C1. Did you report the number of parameters in the models used, the total computational budget (e.g., GPU hours), and computing infrastructure used?
section 4

The Responsible NLP Checklist used at ACL 2023 is adopted from NAACL 2022, with the addition of a question on AI writing assistance.

☑ C2. Did you discuss the experimental setup, including hyperparameter search and best-found hyperparameter values?
section 4

☒ C3. Did you report descriptive statistics about your results (e.g., error bars around results, summary statistics from sets of experiments), and is it transparent whether you are reporting the max, mean, etc. or just a single run?
The benchmark has detailed instructions about experimental settings and even random seeds, thus, in order to compare fairly with previous methods, we strictly followed its setting and performed a single run to evaluate each model numerically.

☑ C4. If you used existing packages (e.g., for preprocessing, for normalization, or for evaluation), did you report the implementation, model, and parameter settings used (e.g., NLTK, Spacy, ROUGE, etc.)?
section 4

D ☒ Did you use human annotators (e.g., crowdworkers) or research with human participants?

Left blank.

☐ D1. Did you report the full text of instructions given to participants, including e.g., screenshots, disclaimers of any risks to participants or annotators, etc.?
No response.

☐ D2. Did you report information about how you recruited (e.g., crowdsourcing platform, students) and paid participants, and discuss if such payment is adequate given the participants' demographic (e.g., country of residence)?
No response.

☐ D3. Did you discuss whether and how consent was obtained from people whose data you're using/curating? For example, if you collected data via crowdsourcing, did your instructions to crowdworkers explain how the data would be used?
No response.

☐ D4. Was the data collection protocol approved (or determined exempt) by an ethics review board?
No response.

☐ D5. Did you report the basic demographic and geographic characteristics of the annotator population that is the source of the data?
No response.