Code Representation Pre-training with Complements from Program Executions

Jiabo Huang[1], **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
{jiabohuang, yjjyzhao}@tencent.com, {PeterRong96, guoyiwen89}@gmail.com, {yfhe, chen}@ucdavis.edu

Abstract

Language models for natural language processing have been grafted onto programming language modeling for advancing code intelligence. Although it can be represented in the text format, code is syntactically more rigorous, as it is designed to be properly compiled or interpreted to perform a set of behaviors given any inputs. In this case, existing works benefit from syntactic representations to learn from code less ambiguously in forms of abstract syntax tree, control-flow graph, *etc*. However, programs with the same purpose can be implemented in various ways showing different syntactic representations, while the ones with similar implementations can have distinct behaviors. Though trivially demonstrated during executions, such semantics about functionality are challenging to be learned directly from code, especially in an unsupervised manner. Hence, in this paper, we propose FuzzPretrain to explore the dynamic information of programs revealed by their test cases and embed it into the feature representations of code as complements. The test cases are obtained with the assistance of a customized fuzzer and are only required during pre-training. FuzzPretrain yielded more than $6\%/19\%$ mAP improvements on code search over its masked language modeling counterparts trained with only source code and source code coupled with abstract syntax trees (ASTs), respectively. Our experiments show the benefits of learning discriminative code representations from FuzzPretrain.

1 Introduction

Code representation learning is drawing growing attention across the community of artificial intelligence (AI) and software engineering (SE) (Husain et al., 2019; Deng et al., 2023; Liu et al., 2023a; Xiong et al., 2023; Lin et al., 2024; He et al., 2024). The pre-training recipes (Devlin et al., 2019; Liu et al., 2019) for natural languages have been shown effective in code representation learning (Feng et al., 2020; Radford et al., 2019). These methods leverage source code and code structures, such as abstract syntax tree (AST) (Guo et al., 2022; Tipirneni et al., 2022) and control-flow graph (CFG) (Allamanis et al., 2018), to learn code representation. However, these structures are not sufficient for code representation, as they neglect the dynamic behavior of code, which is reflected in program execution (Liu et al., 2023a). Therefore, some works (Wang and Su, 2020; Zhao et al., 2023; Wang et al., 2024) proposed to learn program embedding from the combination of symbolic and concrete execution behaviors. Specially, Zhao et al. (2023) proposed FuzzTuning that utilized fuzzing (Zeller et al., 2019) to generate input-output pairs of programs for code-related downstream tasks through fine-tuning with these test cases. The underlying motivation is that the relationship between inputs and their corresponding outputs essentially represents the functions or subroutines, and ultimately, the entire program. Although effective, these methods necessitate the use of input-output pairs during the inference process. Yet, obtaining high-quality input-output pairs during inference can be time-consuming and requires intricate engineering, thus posing challenges for practical implementation.

In this work, we aim to embed the input-output relationships (represented by test cases) of code into its feature representations pre-training instead of fine-tuning, to address the dependency on real-time fuzzing during inference. To accomplish this goal, we follow Zhao et al. (2023) to take advantage of fuzzing to produce test cases that cover the logic paths of code as comprehensively as possible. However, in this paper, these test cases are used in pre-training instead of fine-tuning. We propose a novel method called **FuzzPretrain** for joint static

[*] Corresponding author

Project page: `https://github.com/Raymond-sci/FuzzPretrain`.

and dynamic information modeling. Particularly, in addition to exploring code structure by masked language modeling (Devlin et al., 2021), it formulates a dynamic information matching (DIM) pretext task to tell test cases of different programs apart according to their correspondence to code. By doing so, the model learns holistic feature representations of code and its test cases, encoding both the structure and functionality. FuzzPretrain also involves a self-distillation objective to accomplish a dynamic information distillation (DID) objective. Thereby, the dynamic information is not only properly modelled but distilled from the holistic representations to code features, so to benefit in practice where the test cases are not available.

We make three contributions in this paper: **(1)** We propose to leverage the test cases of programs obtained with the help of fuzzing as explicit indications of functionality to complement their code and syntactic representations. To the best of our knowledge, this is the first attempt to unveil the benefits of fuzzing to code representation *pre-training*. **(2)** We introduce a novel code pre-training method named FuzzPretrain. It simultaneously models the structure and functionality of code while distilling from such holistic information to represent code in its feature space. It is ready to benefit downstream tasks without extra cost on test case generations. **(3)** Extensive experiments on four code understanding downstream tasks demonstrate the effectiveness of FuzzPretrain on complementing both source code and its syntactic representations, *e.g.* AST, by test cases for learning discriminative feature representations.

2 Related Work

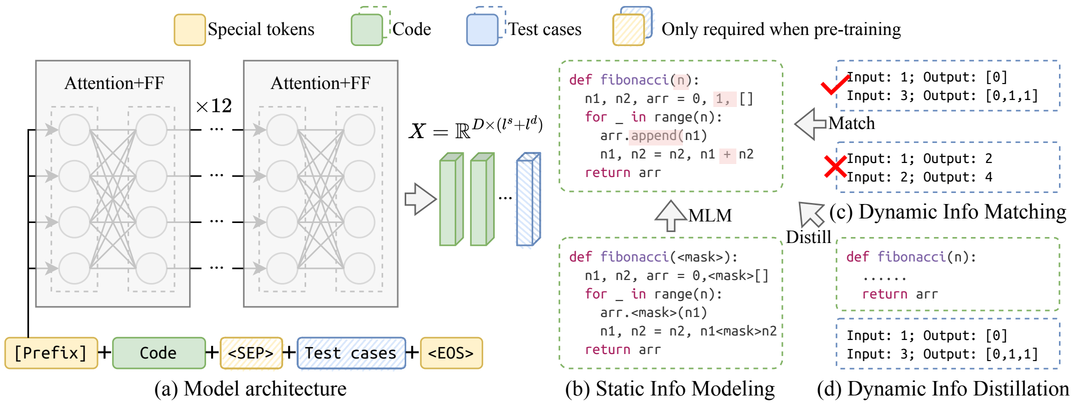
Code representation learning. Language models (Raffel et al., 2020; Liu et al., 2019; Devlin et al., 2021) have achieved unprecedented breakthroughs in natural language processing in recent years (Vaswani et al., 2017; Devlin et al., 2021; Radford et al., 2019). Such successes of language models have been consistently transferred to code representation learning and advance code intelligence. These works leverage plain code (Feng et al., 2020; Chen et al., 2021a; Lu et al., 2021) and code structures, such as abstract syntax tree (AST) (Guo et al., 2022; Tipirneni et al., 2022) and control-flow graph (CFG) (Allamanis et al., 2018) for code representation learning.

However, as illustrated by Wang and Christodorescu (2019), due to the inherent gap between program syntax and runtime semantics, models learned from source code and code structure (i.e., the static models) can be imprecise and not deep at capturing semantic properties. More recent approaches (Wang et al., 2024; Zhao et al., 2023; Liu et al., 2023a) attempted to use dynamic execution traces to learn program representation. By considering dynamic execution paths, symbolic traces provide precise information about dynamic program behavior and reduce false-positive rates in code related tasks.

Language models meet software testing. There are recent efforts on automated bugs mining by language models (Schäfer et al., 2023; Kang et al., 2023), which hold an opposite objective to ours on benefiting software testing by code generation. On the other hand, harnessing program execution traces for comprehensive code representation learning has been widely studied (Wang et al., 2017; Wang and Su, 2020; Henkel et al., 2018; Liu et al., 2023a; Ding et al., 2023). As execution traces are challenging for the end user to specify, they are more difficult to obtain than more basic forms of specification such as input/output pairs (Shin et al., 2018). Fuzzing is supported out-of-the-box for most mainstream programming languages to generate the test cases (Serebryany, 2017; Ding and Le Goues, 2021), which is crucial for constructing multilingual code understanding models. Whilst Shin et al. (2018); Chen et al. (2021b) explores the benefits of test cases to program synthesis, Zhao et al. (2023) share the same insight with us to leverage auto-generated test cases for discriminative code representation learning. Zhao et al. (2023) assume the availability of test cases on every downstream task, however, collecting additional information about the structure or functionality of code requires sufficient expertise in SE and this undoubtedly hampers the model's applicability. By contrast, we aim to explore program executions only in pre-training to preserve the benefits of dynamic information to code understanding in practice where test cases are not mandatory.

3 Code Representation Pre-training

Given a piece of source code S and a sequence encoder f_θ parameterized by θ, our objective is to explore the underlying semantics of the code and encode them in a latent representational space $X^s = f_\theta(S) = \{\boldsymbol{x}_1^s, \boldsymbol{x}_2^s, \cdots, \boldsymbol{x}_{|S|}^s\} \in \mathbb{R}^{k\times|S|}$ in



```
def fibonacci(n):
  n1, n2, arr = 0, 1, []
  for _ in range(n):
    arr.append(n1)
    n1, n2 = n2, n1 + n2
  return arr
```

```
Input: 1; Output: [0]
Input: 3; Output: [0,1,1]
```

```
Input: 1; Output: 2
Input: 2; Output: 4
```

```
def fibonacci(<mask>):
  n1, n2, arr = 0,<mask>[]
  for _ in range(n):
    arr.<mask>(n1)
    n1, n2 = n2, n1<mask>n2
  return arr
```

```
def fibonacci(n):
  ......
  return arr
```

```
Input: 1; Output: [0]
Input: 3; Output: [0,1,1]
```

Figure 1: An overview of FuzzPretrain. **(a)** The input(code and test cases) is encoded by a transformer. FuzzPretrain learns code feature representations by **(b)** static information modeling (SIM) through masked tokens predictions, **(c)** dynamic information matching (DIM) to match test cases to code, and **(d)** dynamic information distillation (DID) to summarize the holistic information about code structure and functionality.

k-dimensions. This is to provide a general understanding of code, which enables efficient fine-tuning on downstream tasks.

In this paper, we propose to explore the dynamic information obtained from fuzzing process, to complement the static information learned from code structure, such that we can embed both in feature representations of code. We present **FuzzPretrain** whose overview is depicted in Fig. 1. We first collect a large-scale code corpus based on CodeNet (Puri et al., 2021) and pair each code snippet with multiple test cases synthesized with the assistance of the same fuzzer as in Zhao et al. (2023)'s work. We denote the test cases corresponding to S as D and concatenate it with the code as its joint static and dynamic representation $H = S \oplus D$. By feeding S (or H) into f_θ, the features X^s (or X^h) are trained by masked tokens predictions (Fig. 1 (b)) and test cases to code matching (Fig. 1 (c)). Besides, FuzzPretrain distills from the holistic features X^h of code and test cases and embed it into X^s, in order to adapt to downstream tasks where test cases D are not available.

3.1 Fuzzing Code Corpus

Fuzzing is a software verification technique that plays an important role in identifying vulnerabilities and enhancing software reliability. A fuzzer verifies the software by repeatedly generating inputs for the software to execute. For each execution, the fuzzer monitors the internal state of the software to determine if the input triggers any new behaviors, and a new behavior is deemed triggered if an input explores at least one new edge of the program. These inputs will be stored for future input generation. Input generation and behavior monitoring together allow the fuzzer to effectively focus on exploring new program behaviors. By running programs with these inputs, we obtain test cases (*i.e.*, program inputs and the corresponding outputs) of each program. The test cases embed runtime information that cannot be easily inferred by static analysis or learned by language models that solely learn from static information. Therefore, using them should supply extra dynamic information to the language models. In practice, we employed exactly the same methods as outlined in FuzzTuning (Zhao et al., 2023) to carry out preprocessing, compilation, and fuzzing of the code. Details can be found in Zhao et al. (2023)'s paper.

3.2 Static and Dynamic Information Modeling

Taking CodeBERT as a base model, we show how to derive *FuzzCodeBERT*, a more powerful code representation model obtained following the spirit of our FuzzPretrain in this section. The design with other base models should be similar.

Input/Output representations. As illustrated in Fig. 1 (a), we follow Feng et al. (2020) to concatenate different parts of inputs together with an <SEP> token and put an end-of-sentence <EOS> token to the end of the concatenation. For the code part, we follow Feng et al. (2020) to obtain S. For the test cases, we follow Zhao et al. (2023) to decode them from a series of bytes to Unicode strings and then prompt them in the form of natural language: "Input is: INPUT; Output is: OUTPUT", and we concatenate multiple test cases of a program also with the <SEP> token. CodeBERT adopts the output feature of the <BOS> token as its sequence-level representation, thus our FuzzCodeBERT feeds the concatenation $H = S \oplus D$ into the encoder f_θ and use the output feature of <BOS> as the sequence-level representation $\boldsymbol{x}^h$. The representation $\boldsymbol{x}^s$ can be similarly obtained when only S is used as the input.

Static information modeling. To learn from the structure of code S, we adopt the conventional masked language modeling (MLM) which has been shown simple yet effective on context understanding (Devlin et al., 2021). We follow the common practices to randomly choose 15% of the tokens in S and replace 80% of the selections with a special <MASK> token, 10% with random tokens and the remaining are left unchanged. Formally, given the code S, a subset $M \subset S$ of it is masked out and leaving a sequence $\tilde{S}$ with replaced tokens. Then, the learning objective is:

$$\mathcal{L}_{\text{SIM}}(S) = -\sum_{m \in M} \log p(m|\tilde{X}^s), \qquad (1)$$

where m is one of the masked tokens and $\tilde{X}^s$ is the features of $\tilde{S}$ produced by f_θ. The term $p(m|\tilde{X}^s)$ denotes the probability that m is correctly reconstructed given the incomplete context $\tilde{X}^s$.

Dynamic information modeling. To learn from the dynamic program information, we propose to match the input-output mappings derived from test cases. Given a code sequence S, we randomly sample an unmatched list of test cases D^- and decide whether to concatenate S with its own test cases D or the negative one D^- to form an input sequence H at each training step. We then pair H with a binary label $y \in \{0, 1\}$ indicating whether the mapping relationships embedded in it are consistent. After that, H is encoded by f_θ to compute its sequence-level representation $\boldsymbol{x}^h$, which is further fed into an additional linear projection layer FC followed by a binary classifier f_ϕ:

$$\mathcal{L}_{\text{DIM}}(S, D) = \text{BCE}(y, f_\phi(\text{FC}(\boldsymbol{x}^h))). \qquad (2)$$

In Eq. (2), the feature $\boldsymbol{x}^h$ of H is linearly transformed the fed into the classifier f_ϕ to predict how likely the code and test cases in H are matched.

Dynamic information distillation. Eq. (1) and Eq. (2) require distinct model inputs. Furthermore, it remains uncertain whether extracting dynamic information from H in Eq. (2) can enhance the representation of S. which is crucial considering that test cases are not available in many downstream tasks. Therefore, we further devise a dynamic information distillation (DID) objective to simultaneously learn the holistic information from both code and test cases $H = S \oplus D$ and enforce encoding such information in the features of code S. Inspired by Tian et al. (2020), we formulate DID in the contrastive learning paradigm to identify the holistic representation H from a list of random samples H^- according to the corresponding source code S. To be concrete, we follow He et al. (2020) to maintain a stale copy $f_{\hat{\theta}}$ of the backbone encoder, which shares the identical architecture with f_θ and is updated accordingly by exponential moving average (EMA) (Lucas and Saccucci, 1990). We then compute the sequence-level feature representation $\boldsymbol{x}^s$ of S and $\hat{\boldsymbol{x}}^h$ of H by f_θ and $f_{\hat{\theta}}$, respectively. Given the holistic features X^- of a set of random samples H^- computed by $f_{\hat{\theta}}$, which are likely with different semantics from H, we train f_θ to optimize:

$$\begin{aligned}&\mathcal{L}_{\text{DID}}(S, S \oplus D) = \\ &- \log \frac{g(\hat{\boldsymbol{x}}^h, \boldsymbol{x}^s)}{g(\hat{\boldsymbol{x}}^h, \boldsymbol{x}^s) + \sum_{\boldsymbol{x}^- \in X^-} g(\boldsymbol{x}^-, \boldsymbol{x}^s)}.\end{aligned} \qquad (3)$$

The function $g(x, y) = \exp(\cos(x, y)/\tau)$ in Eq. (3) computes the exponential cosine similarity between two vectors where τ is a temperature hyperparameter controlling the concentration degree of the similarity distribution. In contrast to $\mathcal{L}_{\text{DIM}}$, we always compute the holistic feature $\hat{\boldsymbol{x}}^h$ of code and its own (matching) test cases to avoid the distractions from inconsistent structure and functionality.

3.3 Model Training and Inference

The FuzzCodeBERT model is optimized alternatively according to the above three objectives on each mini-batch of data following (Lample and Conneau, 2019; Guo et al., 2022). At each training step, the stale encoder $f_{\hat{\theta}}$ is updated according to f_θ by EMA: $\hat{\theta} = \lambda\hat{\theta} + (1 - \lambda)\theta$ with a momentum factor λ, and the holistic representations $\hat{\boldsymbol{x}}^h$ obtained from code and its corresponding test cases will be fed into the queue X^- with the oldest ones inside being removed in a first-in-first-out manner. After pre-training, we keep only the transformer encoder f_θ which is able to yield discriminative feature representations of code $X^s = f_\theta(S)$ when only it is available but not the test cases D at inference or on downstream tasks.

4 Experiments

We adopted the benchmark datasets introduced by Puri et al. (2021) to be fuzzed as the training data of our experiments, which are composed of 1.2M code snippets implemented in C++/Python/Java.

We want to emphasize that our FuzzPretrain is a generic method that can be integrated into many other static-based models more than CodeBERT. To verify this, we perform experiments with one more base model called UniXcoder (Guo et al., 2022), which was trained with code and AST. Correspondingly, we denote the variant of FuzzPretrain built upon UniXcoder as *FuzzUniXcoder*. We followed the base models, *i.e.*, CodeBERT (Feng et al., 2020) and UniXcoder (Guo et al., 2022) to take a 12-layer transformer with 125M learnable parameters for sequence encoding. We trained FuzzPretrain for 10K steps by the Adam optimizer (Kingma and Ba, 2014), which took around 12/20 hours on 8 Nvidia V100 GPUs for code and AST, respectively. For hyperparameter selections, we carefully aligned with our base models as well as He et al. (2020) regarding $\mathcal{L}_{\text{DID}}$ (Eq. (3)). We evaluated FuzzPretrain on four standard code understanding benchmarks adopted by Guo et al. (2022) including code-to-code search (abbreviated as code search) on CodeNet, clone detection on POJ-104 (Mou et al., 2016), defect detection on Devign (Zhou et al., 2019) and text-to-code search (abbreviated as text search) on CosQA (Huang et al., 2021). We adopted mean average precision (mAP) as the evaluation metric for code search and clone detection, accuracy for defect detection and mean reciprocal rank (MRR) for text search. More details about our implementation and evaluation protocols can be found in Appendix A. Note that test cases are only used in our unsupervised pre-training phase and never used in any downstream tasks in experiments.

4.1 Code Representation Learning

Learning with modality discrepancy. To study whether the inconsistency between pre-training and deployment will refrain FuzzPretrain from benefiting general code understanding, we first adopted the code search task to identify equivalent functions without fine-tuning. Considering that FuzzPretrain was trained on different data from its base models (CodeBERT and UniXcoder), to derive reliable conclusions from fair comparisons, we built several fairer baselines. The baselines were trained under the exact same settings as FuzzPretrain but learning from only code or AST without test cases. We presented CodeBERT-MLM/UniXcoder-MLM to train by MLM solely as our baselines following Liu et al. (2023b), and CodeBERT-MLM+RTD/UniXcoder-MLM+Contrast to adopt all the losses dedicated to code understanding in their papers for comprehensive exploration on static information modeling.

As shown in Table 1, the superior performances attained by FuzzCodeBERT and FuzzUniXcoder over their static baselines demonstrate that FuzzPretrain is able to yield discriminative code representations that are beneficial to downstream tasks where test cases are not given. We attribute the performance superiority obtained by FuzzPretrain to the designs of not only modeling the dynamic information jointly from code and test cases but also distilling such knowledge to be encoded into the feature representations of code. This is evident by the degradation of FuzzPretrain when training without either of the proposed components. Such performance drops further verify the effectiveness of our delicate designs and demonstrate that it is non-trivial to benefit code representation learning by dynamic program information. We further make qualitative studies to show the superiority of FuzzPretrain in Appendix B.

Code understanding in novel domains. We investigated whether our learned code features are transferable and beneficial to downstream tasks in unseen data domains (Lu et al., 2021) in Table 2. We see non-negligible performance advantages obtained by FuzzPretrain over CodeBERT-MLM and UniXcoder-MLM. Although introducing contrastive learning by feeding the same code inputs to the encoder twice (Gao et al., 2021) (*i.e.*, "Contrast" in Table 2) is helpful to UniXcoder-MLM on defect detection, it leads to subtle performance degradation on the other two tasks. In fact, FuzzPretrain can obtain a similar improvement (from 64.5% to 65.6%) by integrating such a code-to-code contrast into our FuzzUniXcoder reported in Table 2. This also implies the potential of our dynamic information modeling on more advanced base models.

Comparisons with more state-of-the-arts. Although FuzzPretrain adopted different pre-training data from the popular bi-modal dataset (Husain et al., 2019) to enable compilation and fuzzing, we compared it with the state-of-the-art models regardless to demonstrate its competitiveness on code understanding. Specifically, we compared FuzzPretrain with three types of methods. RoBERTa (Liu et al., 2019) learns at the natural language conventions. DISCO (Ding et al., 2022), CodeRetriever (Li et al., 2022a), and ContraBERT (Liu et al., 2023b) benefit from contrastive learning as in our solution. GraphCodeBERT (Guo et al., 2021), CodeExecutor (Liu et al., 2023a) and

Model	DYN	Ruby			Python			Java			Overall
		Ruby	Python	Java	Ruby	Python	Java	Ruby	Python	Java	
CodeBERT	✗	13.55	3.18	0.71	3.12	14.39	0.96	0.55	0.42	7.62	4.94
CodeBERT-MLM	✗	22.45	5.67	1.95	6.74	25.70	5.01	3.61	5.84	13.45	10.05
CodeBERT-MLM+RTD	✗	13.22	1.00	0.10	1.24	14.35	1.20	0.20	0.18	6.34	4.20
FuzzCodeBERT	✓	**27.92**	**14.88**	**7.92**	**15.39**	**30.47**	**10.26**	**9.94**	**10.65**	**17.75**	**16.13**
FuzzCodeBERT w/o DIM	✓	24.05	14.08	6.96	16.32	27.51	9.54	8.66	9.76	13.49	14.49
FuzzCodeBERT w/o DID	✓	18.21	2.92	0.72	2.88	25.67	3.13	0.80	1.98	17.98	8.25
UniXcoder	✗	29.05	26.36	15.16	23.96	30.15	15.07	13.61	14.53	16.12	20.45
UniXcoder-MLM	✗	20.49	13.54	3.25	10.40	19.49	3.69	4.13	5.14	12.29	10.27
UniXcoder-MLM+Contrast	✗	30.83	25.73	16.46	25.44	30.50	16.80	16.01	17.26	18.86	21.99
FuzzUniXcoder	✓	**42.84**	**29.83**	**17.70**	**33.73**	**47.77**	**21.94**	**20.83**	**23.52**	**33.78**	**30.22**
FuzzUniXcoder w/o DIM	✓	22.50	13.52	6.66	15.31	22.99	6.81	7.54	6.84	12.94	12.79
FuzzUniXcoder w/o DID	✓	12.92	5.10	1.36	5.56	14.86	0.87	0.96	0.50	6.81	5.44

Table 1: Evaluations on code search. Results of our base models (CodeBERT and UniXcoder) are from Guo et al. (2022)'s paper, which are marked in grey because of different training data. The first and second rows in the header indicate the programming language of the query and the target code snippets, respectively. The column "DYN" indicates whether a model was trained using the test cases or not. mAP scores (%) are reported.

Model	DYN	Clone	Defect	Text
CodeBERT	✗	82.7	62.1	65.7
CodeBERT-MLM	✗	88.7	63.5	67.4
CodeBERT-MLM+RTD	✗	84.7	62.0	66.3
FuzzCodeBERT	✓	**93.0**	**64.1**	**69.1**
UniXcoder	✗	90.5	64.5*	70.1
UniXcoder-MLM	✗	91.2	63.8	69.8
UniXcoder-MLM+Contrast	✗	91.1	**65.2**	69.7
FuzzUniXcoder	✓	**92.2**	64.5	**70.7**

Table 2: Evaluations in novel data domains. Results of the base models are marked in grey as training on different data from ours. Results marked with * are reproduced using the checkpoints from the authors.

Model (Year)	Clone	Defect	Text
RoBERTa (2019)	76.7	61.0	60.3
GraphCodeBERT (2021)	85.2	62.9	68.4
DISCO (2022)	82.8	63.8	-
CodeRetriever (2022a)	88.8	-	69.7
ContraBERT (2023b)	90.5	64.2	66.7*
CodeExecutor (2023a)	70.5*	59.0*	13.1*
TRACED (2023)	91.2	**65.9**	-
FuzzCodeBERT	**93.0**	64.1	69.1
FuzzUniXcoder	92.2	64.5	**70.7**

Table 3: Comparisons with the state-of-the-art that adopt the same backbone network as ours with 125M parameters. Results marked with * are reproduced using the checkpoints from the authors.

TRACED (Ding et al., 2023) explore program functionality from DFG or execution traces. Note that, we evaluated CodeExecutor without re-ranking by execution traces to be more practical. As shown in Table 3, the performance advantages of FuzzPretrain over GraphCodeBERT implies that mining the functionality of programs from the intricate dependencies among variables is more challenging than modeling from the concrete input-output behavior represented by test cases. Besides, TRACED is good at code understanding in finer granularity (*e.g.* defect detection) by learning from the detailed internal status of programs in execution traces while our FuzzPretrain is superior on global understanding of code snippets (*e.g.* clone detection) as the test cases we adopted is invariant to implementation variations that are agnostic to functionality. Whilst the methods that are based on contrastive learning of source code yielded promising results, FuzzPretrain's competitiveness shows the effectiveness of pre-training with complements from dynamic information and fuzzing test cases. More importantly, FuzzPretrain can be integrated into those methods to further benefit from more advanced modeling of static information.

4.2 Ablation study

Effects of dynamic information modeling. To study the independent contributions of DIM (Eq. (2)) and DID (Eq. (3)) to dynamic information modeling, we constructed and compared three variants of FuzzPretrain by removing either or both of them. As shown in Fig. 2, the variant of FuzzPretrain trained with only DID (w/o DIM) often out-performed the baselines (MLM) trained with neither DIM nor DID. This indicates that the test cases concatenated after the source code or its syntactic representations potentially play the roles of data augmentation to perturb the distributions of code by supplementing the dynamic information from test cases. Although adopting either DIM or DID is slightly better than FuzzPretrain occasion-

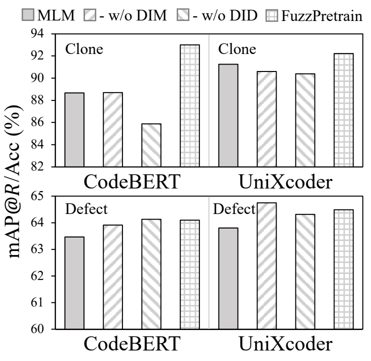


Figure 2: Effects of different components for dynamic information modeling. We constructed three variants of FuzzPretrain with either DIM or DID or both being removed to be compared.

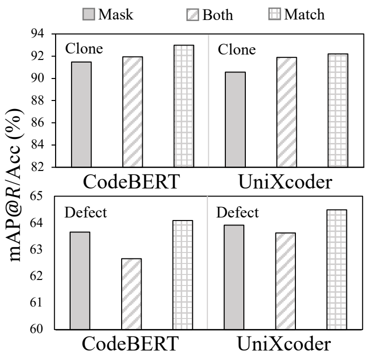


Figure 3: Dynamic information modeling by MLM. The "Mask" variant replaces DIM by MLM for both code and test cases while "Match" is the design we adopted and "Both" is the combination of the two.

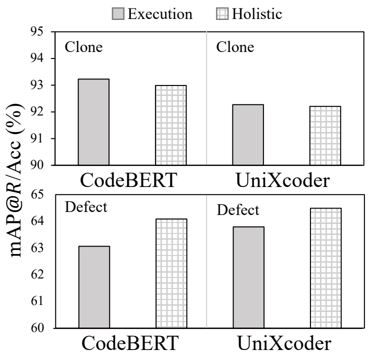


Figure 4: Positive pairs in DID. The "Execution" variant constructs the positive pairs in DID using code T^s and its test cases T^d, and our "Holistic" design contrasts code to its concatenation with test cases $T^s \oplus T^d$.

ally, the consistent improvements we brought to different base models on both the retrieval (clone detection) and classification (defect detection) tasks demonstrate the generality of combining the two designs, which is critical for a pre-training method.

Dynamic information modeling using MLM. To justify our DIM's effectiveness on dynamic modeling over the conventional MLM, we replace or combine it with MLM on both code and test cases to form two variants of FuzzPretrain as "Mask" and "Both" in Fig. 3, respectively. The performance superiority of "Match" to the two variants indicates that applying MLM in test cases is sub-optimal. From our training logs, we observe that the encoder could accurately reconstruct the masked tokens in test cases (or code) regardless of whether the code (or the test cases) is available in the model input. This implies that syntactic and functional representations are both very informative and can be well reconstructed independently, which makes it less straightforward to associate them by MLM. On the contrary, the labels for our DIM is defined only by the relationships between code and test cases, hence, it is infeasible to predict such labels without learning their correlations. Besides, the "Both" alternative tends to associate code with arbitrary patterns in test cases, which are explored by MLM. The resulted correlations can be distracting to code understanding considering the randomness in test cases introduced by fuzzing.

Positive pairs in DID. To justify our design of DID, we built a variant of FuzzPretrain which formulates the $\mathcal{L}_{\text{DID}}$ to identify test cases D according to their corresponding code S or AST by constructing the positive pairs in Eq. (3) to be (S, D) instead of $(S, S \oplus D)$ in FuzzPretrain. We denote this variant as "Execution" and FuzzPretrain as "Holistic" to be compared in Fig. 4. Although the performances of the "Execution" variant on clone detection are on par with that of the "Holistic" counterpart, its inferiority on defect detection is non-negligible. We believe that this is due to the distribution discrepancies between code and test cases (*e.g.* test cases are likely to involve an exhaustive list of random numbers as inputs which are barely seen in code). It is more reasonable to jointly learn from test cases and source code to simultaneously benefit from dynamic information and mitigate the negative impacts from distribution discrepancies.

5 Conclusion

In this paper, we have made the first attempt to use (fuzzing) test cases to facilitate effective code representation pre-training. To benefit from such a "new modality" of data that is often not available in downstream tasks, we have proposed FuzzPretrain for joint static and dynamic information modeling. Specifically, FuzzPretrain is trained not only to accomplish the conventional masked tokens predictions objective but also to learn the input-output relationships from test cases encoding the program-specific runtime behaviors, as well as enforcing the model to infer such dynamic knowledge from code structures solely. We have shown how FuzzPretrain can be used to enhance CodeBERT and UniXocder. Extensive experiments on various code understanding downstream tasks demonstrate the benefits of our FuzzPretrain.

References

Miltiadis Allamanis, Marc Brockschmidt, and Mahmoud Khademi. 2018. Learning to represent programs with graphs. In *The International Conference on Learning Representations*.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Pondé de Oliveira Pinto, Jared Kaplan, Harrison Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Joshua Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. 2021a. Evaluating large language models trained on code. *CoRR*, abs/2107.03374.

Xinyun Chen, Dawn Song, and Yuandong Tian. 2021b. Latent execution for neural program synthesis beyond domain-specific languages. *Advances in Neural Information Processing Systems*, 34:22196–22208.

Yinlin Deng, Chunqiu Steven Xia, Haoran Peng, Chenyuan Yang, and Lingming Zhang. 2023. Large language models are zero-shot fuzzers: Fuzzing deep-learning libraries via large language models.

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, pages 4171–4186, Minneapolis, Minnesota. Association for Computational Linguistics.

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2021. Bert: Pre-training of deep bidirectional transformers for language understanding. In *Association for Computational Linguistics*.

Yangruibo Ding, Luca Buratti, Saurabh Pujar, Alessandro Morari, Baishakhi Ray, and Saikat Chakraborty. 2022. Towards learning (dis)-similarity of source code from program contrasts. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 6300–6312, Dublin, Ireland. Association for Computational Linguistics.

Yangruibo Ding, Ben Steenhoek, Kexin Pei, Gail Kaiser, Wei Le, and Baishakhi Ray. 2023. Traced: Execution-aware pre-training for source code. In *International Conference on Software Engineering*.

Zhen Yu Ding and Claire Le Goues. 2021. An empirical study of oss-fuzz bugs. In *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*, pages 131–142. IEEE.

Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. 2020. CodeBERT: A pre-trained model for programming and natural languages. In *Findings of the Association for Computational Linguistics: EMNLP 2020*, pages 1536–1547, Online. Association for Computational Linguistics.

Tianyu Gao, Xingcheng Yao, and Danqi Chen. 2021. SimCSE: Simple contrastive learning of sentence embeddings. In *Conference on Empirical Methods in Natural Language Processing*, pages 6894–6910, Online and Punta Cana, Dominican Republic. Association for Computational Linguistics.

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

Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, et al. 2021. Graphcodebert: Pre-training code representations with data flow. In *The International Conference on Learning Representations*.

Kaiming He, Haoqi Fan, Yuxin Wu, Saining Xie, and Ross Girshick. 2020. Momentum contrast for unsupervised visual representation learning. In *The IEEE conference on computer vision and pattern recognition*, pages 9729–9738.

Yifeng He, Jiabo Huang, Yuyang Rong, Yiwen Guo, Ethan Wang, and Hao Chen. 2024. Unitsyn: A large-scale dataset capable of enhancing the prowess of large language models for program testing. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 1061–1072.

Jordan Henkel, Shuvendu K Lahiri, Ben Liblit, and Thomas Reps. 2018. Code vectors: Understanding programs through embedded abstracted symbolic traces. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 163–174.

Junjie Huang, Duyu Tang, Linjun Shou, Ming Gong, Ke Xu, Daxin Jiang, Ming Zhou, and Nan Duan. 2021. Cosqa: 20,000+ web queries for code search and question answering. In *Association for Computational Linguistics*.

Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. 2019. Codesearchnet challenge: Evaluating the state of semantic code search. *arXiv preprint arXiv:1909.09436*.

Sungmin Kang, Juyeon Yoon, and Shin Yoo. 2023. Large language models are few-shot testers: Exploring llm-based general bug reproduction.

Diederik Kingma and Jimmy Ba. 2014. Adam: A method for stochastic optimization. *The International Conference on Learning Representations*.

Guillaume Lample and Alexis Conneau. 2019. Cross-lingual language model pretraining. *arXiv preprint arXiv:1901.07291*.

Xiaonan Li, Yeyun Gong, Yelong Shen, Xipeng Qiu, Hang Zhang, Bolun Yao, Weizhen Qi, Daxin Jiang, Weizhu Chen, and Nan Duan. 2022a. Coderetriever: Unimodal and bimodal contrastive learning. In *Conference on Empirical Methods in Natural Language Processing*.

Xiaonan Li, Daya Guo, Yeyun Gong, Yun Lin, Yelong Shen, Xipeng Qiu, Daxin Jiang, Weizhu Chen, and Nan Duan. 2022b. Soft-labeled contrastive pre-training for function-level code representation. *arXiv preprint arXiv:2210.09597*.

Jiongliang Lin, Yiwen Guo, and Hao Chen. 2024. Intrusion detection at scale with the assistance of a command-line language model. *arXiv preprint arXiv:2404.13402*.

Chenxiao Liu, Shuai Lu, Weizhu Chen, Daxin Jiang, Alexey Svyatkovskiy, Shengyu Fu, Neel Sundaresan, and Nan Duan. 2023a. Code execution with pre-trained language models. *ACL*.

Shangqing Liu, Bozhi Wu, Xiaofei Xie, Guozhu Meng, and Yang Liu. 2023b. Contrabert: Enhancing code pre-trained models via contrastive learning. In *International Conference on Software Engineering*.

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*.

James M Lucas and Michael S Saccucci. 1990. Exponentially weighted moving average control schemes: properties and enhancements. *Technometrics*, 32(1):1–12.

Lili Mou, Ge Li, Lu Zhang, Tao Wang, and Zhi Jin. 2016. Convolutional neural networks over tree structures for programming language processing. In *Proceedings of the AAAI conference on artificial intelligence*, volume 30.

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

Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, et al. 2019. Language models are unsupervised multitask learners. *OpenAI blog*, 1(8):9.

Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J Liu. 2020. Exploring the limits of transfer learning with a unified text-to-text transformer. *The Journal of Machine Learning Research*, 21(1):5485–5551.

Max Schäfer, Sarah Nadi, Aryaz Eghbali, and Frank Tip. 2023. Adaptive test generation using a large language model. *arXiv preprint arXiv:2302.06527*.

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

Eui Chul Shin, Illia Polosukhin, and Dawn Song. 2018. Improving neural program synthesis with inferred execution traces. *Advances in Neural Information Processing Systems*, 31.

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.

Yonglong Tian, Dilip Krishnan, and Phillip Isola. 2020. Contrastive representation distillation. *ICLR*.

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

Laurens Van der Maaten and Geoffrey Hinton. 2008. Visualizing data using t-sne. *Journal of machine learning research*, 9(11).

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. *Advances in neural information processing systems*, 30.

Huanting Wang, Zhanyong Tang, Shin Hwei Tan, Jie Wang, Yuzhe Liu, Hejun Fang, Chunwei Xia, and Zheng Wang. 2024. Combining structured static code information and dynamic symbolic traces for software vulnerability prediction. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, pages 1–13.

Ke Wang and Mihai Christodorescu. 2019. Coset: A benchmark for evaluating neural program embeddings. *arXiv preprint arXiv:1905.11445*.

Ke Wang, Rishabh Singh, and Zhendong Su. 2017. Dynamic neural program embedding for program repair. *arXiv preprint arXiv:1711.07163*.

Ke Wang and Zhendong Su. 2020. Blended, precise semantic program embeddings. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 121–134.

Yue Wang, Hung Le, Akhilesh Deepak Gotmare, Nghi D. Q. Bui, Junnan Li, and Steven C. H. Hoi. 2023. Codet5+: Open code large language models for code understanding and generation.

Weimin Xiong, Yiwen Guo, and Hao Chen. 2023. The program testing ability of large language models for code. *arXiv preprint arXiv:2310.05727*.

Andreas Zeller, Rahul Gopinath, Marcel Böhme, Gordon Fraser, and Christian Holler. 2019. The fuzzing book.

Jianyu Zhao, Yuyang Rong, Yiwen Guo, Yifeng He, and Hao Chen. 2023. Understanding programs by exploiting (fuzzing) test cases. *ACL*.

Yaqin Zhou, Shangqing Liu, Jingkai Siow, Xiaoning Du, and Yang Liu. 2019. Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks. *Advances in neural information processing systems*, 32.

A Implementation, Datasets and Evaluation protocols

Datasets. Puri et al. (2021) proposed a large-scale dataset CodeNet, consisting of over 14 million code samples and about 500 million lines of code, which is intended for training and evaluating code models. We adopted the C++1000, C++1400, Python800, Java250 benchmark datasets of CodeNet to be fuzzed as the training data of FuzzPretrain. We then evaluated FuzzPretrain extensively on four code understanding benchmark datasets of CodeXGLUE (Lu et al., 2021): (1) another subset of **CodeNet** (Puri et al., 2021) collected by Guo et al. (2022) consisting of 50K functions implemented in Python, Java, and Ruby for solving one of $4,053$ online coding problems; (2) **POJ-104** (Mou et al., 2016) which contains 104 C/C++ coding problems with 500 code submissions to each; (3) **Devign** (Zhou et al., 2019) which is composed of vulnerable functions from four large and popular C-language open-source projects with manual labels; (4) **CosQA** which contains 20,604 pairs of code and real-world web queries (Huang et al., 2021) with annotations from human experts indicating whether the questions raised by the queries can be properly addressed by the code. All the data used for fine-tuning and testing are carefully aligned with previous studies (Feng et al., 2020; Guo et al., 2022).

Evaluation protocols. We first investigated the discrimination ability of the learned code representations by code-to-code search (abbreviated as *code search* in the paper) on the subset of CodeNet collected by Guo et al. (2022). In this task, submissions of the same coding problems are assumed to share the same semantics regardless of their implementations. The feature distances between code pairs were adopted to measure their semantic similarity and the mean average precision (**mAP**) was reported to quantify the quality of the retrieval results. We then studied the effects of FuzzPretrain to several downstream tasks in unseen domains, including clone detection, defect detection and text-to-code search (abbreviated as *text search*). The objective of clone detection is similar to that of code search but with fine-tuning in target domains. We followed the same protocol of Feng et al. (2020)'s work to test on POJ-104 and use **mAP@**R to assess the results, with only the top-R ($R = 499$) most similar samples were considered in retrieval. In the task of text search, which requires retrieving code snippets according to textual queries, the mean reciprocal rank (**MRR**) is adopted as the metric following Guo et al. (2022)'s work. This evaluation was conducted on CosQA. Defect detection was carried out on Devign and the accuracy (**Acc**) of binary classification is adopted with a fixed threshold of 0.5.

Implementation details. Both our base models, *i.e.* CodeBERT (Feng et al., 2020) and UniXcoder (Guo et al., 2022), followed Liu et al. (2019) to take a 12-layer transformer with 125M learnable parameters for sequence encoding. We followed their designs to set the batch size to 2048 and 1024 while the maximum sequence length to 512 and 1024 for CodeBERT and UniXcoder, respectively. In inputs, 400 and 800 tokens are reserved for code and AST, respectively, and the rest are for test cases. The test cases of each program were concatenated with the code or the AST by the separation token until reaching the length limits, while the rest was dropped. The FuzzPretrain model was updated by the Adam optimizer (Kingma and Ba, 2014) during training with a learning rate of $2e-5$ for $10K$ steps. For dynamic information distillation $\mathcal{L}_{\text{DID}}$

Licensed under the Apache License, Version 2.0
Licensed under Creative Commons Zero v1.0 Universal

	C++†	C++‡	Python	Java	Overall
CodeBERT	13.95	13.22	31.23	26.72	21.28
CodeBERT-MLM	26.34	24.08	48.71	34.94	33.52
CodeBERT-MLM+RTD	11.61	11.51	25.41	10.23	14.69
FuzzCodeBERT	**69.98**	**68.65**	**78.13**	**69.98**	**71.69**
UniXcoder	17.57	15.89	55.28	45.49	33.56
UniXcoder-MLM	32.84	30.28	46.79	46.90	39.20
UniXcoder-MLM+Contrast	47.47	43.99	60.65	51.54	50.91
FuzzUniXcoder	**71.72**	**68.40**	**80.27**	**77.43**	**74.45**

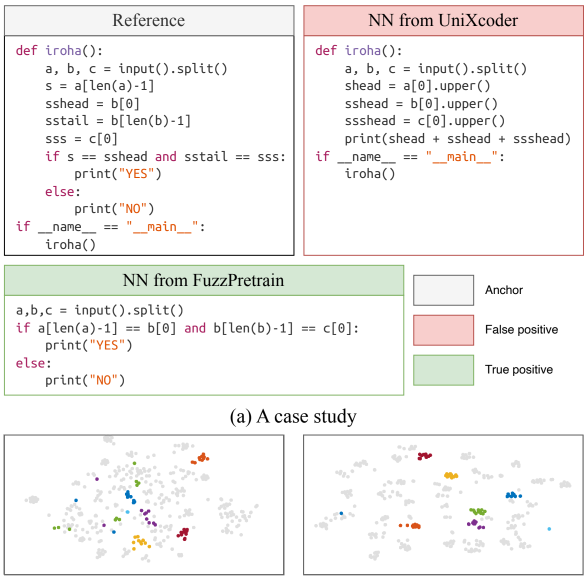
Table 4: Evaluations on inductive code search. To guarantee that no test data is seen by any models even in the unsupervised pre-training, the mAP scores (%) are reported on the test splits of C++1000 ("C++†"), C++1400 ("C++‡"), Python800 ("Python"), and Java250 ("Java") that are all completely disjoint from the pre-training code data. Here, "Overall" indicates the average mAP performance overall.

(Eq. (3)), we followed He et al. (2020) to set the momentum coefficient $m = 0.999$, the temperature $\tau = 0.07$, and the number of random samples $|H^-| = 2^{16}$. The overall pre-training process took around 12/20 hours on 8 Nvidia V100 GPUs for training with code and AST, respectively.

B Additional experiments and analysis

Inductive zero-shot code search. We adopted the testing split provided by UniXcoder (Guo et al., 2022) for evaluation of code search, it is likely to overlap with our training data in CodeNet by sharing over 70% of the coding problems. Therefore, we consider the searching of those overlapping samples as transductive inference problems. This is also a practical scenario given that the training data of the latest code models covers a large proportion of open-source projects in Github and is likely to involve the code-of-interests to users. We have also evaluated in an inductive setup where the query and the candidate code snippets are submissions to 50 coding problems of each programming language that have never been seen during pre-training. As shown in Table 4, the superiority of our FuzzPretrain over both the base models and our baselines still holds. That is, these results show that our model is effective not only in the transductive inference setup for code search, but also in an inductive setup where no training/test overlap exists.

Qualitative studies. We further showed an example of code search in Fig. 5 (a) to exhibit the nearest neighbors of a reference code snippet decided by either UniXcoder or its FuzzPretrain counterpart. Together with the t-SNE (Van der Maaten and Hinton, 2008) visualization of the python code submissions to 50 randomly selected problems (classes) encoded by either of the two models in Fig. 5 (b) and (c) respectively, it is obvious that our FuzzPretrain is sensitive to the functionality of programs regardless of their implementation variations, which results in more compact clusters to be consistent with the underlying semantics of code.



```
def iroha():
    a, b, c = input().split()
    s = a[len(a)-1]
    sshead = b[0]
    sstail = b[len(b)-1]
    sss = c[0]
    if s == sshead and sstail == sss:
        print("YES")
    else:
        print("NO")
if __name__ == "__main__":
    iroha()
```

```
def iroha():
    a, b, c = input().split()
    shead = a[0].upper()
    sshead = b[0].upper()
    ssshead = c[0].upper()
    print(shead + sshead + ssshead)
if __name__ == "__main__":
    iroha()
```

```
a,b,c = input().split()
if a[len(a)-1] == b[0] and b[len(b)-1] == c[0]:
    print("YES")
else:
    print("NO")
```

⬜ Anchor
🟥 False positive
🟩 True positive

(a) A case study

(b) Features from UniXcoder (c) Features from FuzzPretrain

Figure 5: Qualitative studies for code search. The functional equivalence of code snippets are marked by their shared colors. Only a few classes are highlighted with bright colors to be visually distinguishable.

Comparisons to commercial language models. Commercial language models have recently shown remarkable zero-shot capability to various code understanding and generation downstream tasks. Whilst our proposed ideas are generic and integrable into any static-based models regardless of their scale, we further conducted a preliminary comparison to the "text-embedding-ada-002" model from OpenAI to demonstrate our specialization on the task of semantic code search. To be concrete, we followed OpenAI's instruction of getting code embeddings to evaluate their model for python-to-python code search in CodeNet (Puri et al., 2021). The OpenAI's model yielded 35.91% mAP while ours are 30.47% and 47.77% when adopting either CodeBERT or UniXcoder as the base model, respectively. Given that CodeNet carefully removed near-duplicated submissions to the same coding problems with over-high syntactic similarity, such initial evaluation results indicate

https://platform.openai.com/docs/guides/embeddings/use-cases that semantic code search is fundamentally challenging and the test cases we adopted are strong indicators of program's functionality, which ensure our competitiveness to the larger and more complex models.

C Future works and Limitations

Fuzzing code corpus. Our current pre-training data is restricted to OJ-like code corpus (*i.e.*, CodeNet) (Puri et al., 2021), which refrains us from ablating affecting factors in the data distribution in making fair comparison to existing methods. To be more specific, most commonly adopted code corpus (Husain et al., 2019) are composed of standalone functions spread over various software projects (*e.g.*, CodeSearchNet), whose test cases cannot be easily obtained. Whilst OJ data is showing some unique characteristics to benefit our FuzzPretrain model on understanding similar code snippets as indicated by our remarkable performance advantages on POJ-104 (Mou et al., 2016) (Table 3), this also limits our model's generalization ability to other type of code corpus, *e.g.* the F1-score of clone detection on BigCloneBench (Svajlenko et al., 2014) yielded by our FuzzUniXcoder was 1% lower than that by UniXcoder pre-trained on CodeSearchNet. Yet, when both pre-trained on the same selected subset of CodeNet, our FuzzPretrain leads to +0.9% F1 gain in comparison to existing pre-training strategies using, for example, the MLM loss on CodeBERT. Exploring fuzzing on more diverse code corpus help address this limitation.

Text-code tasks. Following the discussion about fuzzing code corpus in the previous paragraph, we would like to mention that, since CodeNet does not contain text description of each code, pre-training on it may not fully unleash the power of pre-training on text-code downstream tasks. That is to say, although we have shown the effectiveness of our FuzzPretrain on the text code search task in Tables 2 and 3, even better results can be obtained if we can pair the CodeNet data with text descriptions or if we can pre-train on a dataset with not only texts and code but also test cases. This also withholds FuzzCodeBERT and FuzzUniXcoder from surpassing every state-of-the-art methods on text-code tasks. In addition to exploiting datasets, extensive experiments presented in this paper also verifies complementary effects of dynamic program modeling to these methods, which implies that combining more advanced methods (Wang et al., 2023; Li et al., 2022b) with our FuzzPretrain also leads to superior performance than that of FuzzCodeBERT and FuzzUniXcoder.

Code generation. Our designs for dynamic information modeling are all about the holistic comprehensions of code in a global picture, while how to benefit token-wise code understanding by using it is not straightforward. We tested UniXcoder with and without our FuzzPretrain on the python dev split of the line-level code completion task in the CodeXGLUE benchmark (Lu et al., 2021), our FuzzUniXcoder yielded $42.73\%/72.03\%$ Exact Match/Edit Sim *vs.* $42.68\%/71.88\%$ by UniXcoder. We did not observe clear improvements brought by FuzzPretrain on code generation tasks which are usually conducted at token-level, leaving an interesting problem to be studied in the future.

D Ethical Consideration

Our approach leverages fuzzing test cases to enhance program understanding. The improved semantic comprehension of programs can be further employed to patch vulnerabilities or address defects in software and systems. However, we strongly encourage careful consideration in advance when applying this method to these applications. Additionally, as fuzz testing is utilized, a notable number of crashes and hangs have been observed in the adopted datasets. We refrain from presenting test cases that lead to these issues to prevent any potential misuse.