TRACED: Execution-aware Pre-training for Source Code

Yangruibo Ding
Columbia University
New York, NY, USA

Benjamin Steenhoek
Iowa State University
Ames, IA, USA

Kexin Pei
Columbia University
New York, NY, USA

Gail Kaiser
Columbia University
New York, NY, USA

Wei Le
Iowa State University
Ames, IA, USA

Baishakhi Ray
Columbia University
New York, NY, USA

ABSTRACT

Most existing pre-trained language models for source code focus on learning the static code text, typically augmented with static code structures (abstract syntax tree, dependency graphs, *etc.*). However, program semantics will not be fully exposed before the real execution. Without an understanding of the program execution, statically pre-trained models fail to comprehensively capture the dynamic code properties, such as the branch coverage and the runtime variable values, and they are consequently less effective at code understanding tasks, such as retrieving semantic clones and detecting software vulnerabilities.

To close the gap between the static nature of language models and the dynamic characteristics of programs, we introduce TRACED, an execution-aware pre-training strategy for source code. Specifically, we pre-train code language models with a combination of source code, executable inputs, and corresponding execution traces. Our goal is to teach code models the complicated execution logic during the pre-training, enabling the model to statically *estimate* the dynamic code properties without repeatedly executing code during task-specific fine-tuning.

To illustrate the effectiveness of our proposed approach, we fine-tune and evaluate TRACED on three downstream tasks: static execution estimation, clone retrieval, and vulnerability detection. The empirical results show that TRACED relatively improves the statically pre-trained code models by 12.4% for complete execution path prediction and by 25.2% for runtime variable value predictions. TRACED also significantly outperforms statically pre-trained models in clone retrieval and vulnerability detection across four public benchmarks.

ACM Reference Format:
Yangruibo Ding, Benjamin Steenhoek, Kexin Pei, Gail Kaiser, Wei Le, and Baishakhi Ray. 2024. TRACED: Execution-aware Pre-training for Source Code. In *2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE '24), April 14–20, 2024, Lisbon, Portugal.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3597503.3608140

ICSE '24, April 14–20, 2024, Lisbon, Portugal

ACM ISBN 979-8-4007-0217-4/24/04…$15.00
https://doi.org/10.1145/3597503.3608140

1 INTRODUCTION

Machine Learning (ML) for source code has enabled many software engineering tasks, such as automated program repair [11, 21–23], bug finding [8, 55], and refactoring [7]. Recently, the common practice of training ML models for source code understanding is based on pre-training a Transformer-based language model on source code. These approaches treat source code programs as *static text* [1, 6, 16, 49], sometimes augmented with program-specific structures such as abstract syntax trees and dependency graphs [10, 17, 18, 35], and adapt pre-training strategies for natural language to learn program representations.

However, many source code understanding tasks require a more comprehensive understanding of *program behavior*. For instance, detecting semantic clones[32] involves determining if two pieces of code behave similarly under similar inputs, even if their structures are apparently different. Likewise, detecting vulnerabilities often requires developers to analyze whether a potentially problematic location can be executed and what kinds of value flows can expose any vulnerability. While existing code models are primarily trained to capture static code properties, they are not effective at reasoning about program behavior. In fact, many of the deeper program semantics only manifest when the code is executed. As a result, they tend to underperform when it comes to tasks that require deeper semantic understanding.

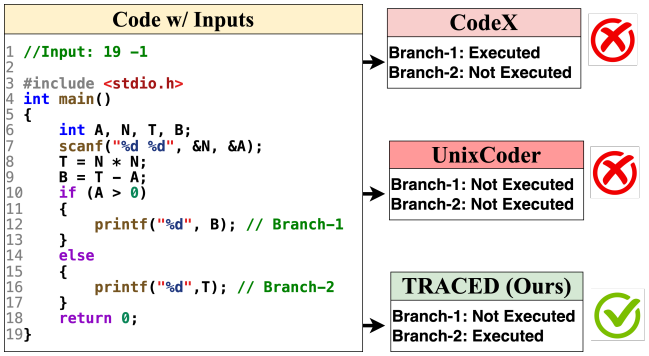

Figure 1: An motivating example from CodeNet's coding challenge No.3597 [41] reveals that statically pre-trained code language models, regardless of their size, could not reason about the branch coverage given a specific input, while TRACED, enhanced with program execution features, correctly identify the execution path.

Motivating Examples. Figure 1 presents an example with simple execution logic to illustrate the failure of statically pre-trained code models on the branch coverage prediction. We query three

pre-trained code models, CodeX [13] (code-davinci-002), UnixCoder [17], and TRACED (ours), to predict the branch coverage, according to the given program inputs. For CodeX, we prompt the model with carefully designed questions, similar to [36], to ask for the branch coverage prediction in the zero-shot setting. Specifically, we augment the prompts by adding comments at the end of lines 12 and 16: `// Will this line be executed? Yes or no?`. To give more hints regarding the data flow, we further add a comment at the end of line 10: `// A is -1, since it accepts the second value of the input`. Unfortunately, even if provided with additional hints of the required data flow for branch prediction, CodeX still failed to predict the correct coverage labels, suggesting it cannot interpret this simple execution.

Besides the zero-shot prompting, we also study whether fine-tuning pre-trained code models to predict execution can lead to better branch prediction. Specifically, we fine-tune another popular pre-trained code model, UnixCoder [17], to predict branch execution while ensuring the motivation example is not seen during training. From the inference results in Figure 1, we notice that UnixCoder cannot predict covered branches even after being fine-tuned. It predicts neither of the branches will be covered, indicating that it does not have the basic understanding that, for this specific example, at least one branch will always be taken on a valid input.

Our approach. To address the limitation of the statically pre-trained code models, we propose TRACED, an execution-aware pre-training strategy to capture the static and dynamic perspectives of the source code. Specifically, we pre-train the Transformer-based language model with multi-task objectives on predicting source code, program states, and execution coverage, forcing the model to reason about both program's runtime behavior and the naturalness of the source code [43] at the same time. We address several technical challenges, such as representing program execution states, encoding the runtime variable values, and representing code coverage, to implement the pre-training strategy.

Representing Program States. During program execution, variables are used to store data that is used by the program. These variables can have different types, such as integers, floating-point numbers, pointers, and arrays. As the program executes, the values of these variables change, reflecting the changes in the program's state. Consequently, software developers typically monitor the variable values, via debugging tools, to observe the execution facts [53] and understand the dynamic behaviors of the program.

In this work, we define the *program state* at a specific time step of the execution as the set of values of every defined variable in the current scope. In other words, the program state is equivalent to the value mapping table of the debugger, which is monitored by the developer when the program is paused by a specific breakpoint.

Value Quantization. While the runtime variable values are traced as concrete values, directly learning them brought challenges to machine learning models. Concrete values span over a wide range of possible values, especially when considering different data types (integers, floating-point numbers, arrays, pointers, etc.), leading to a high-dimensional, complex, but sparse data distribution. This increased data complexity and sparsity challenges the model to learn patterns and relationships between the variable values, as it must deal with many unique inputs, which causes the model to overfit and memorize specific instances rather than generalize to broader patterns. Additionally, noise, outliers, and irregularities of concrete values also mislead the model's learning process. We will empirically demonstrate these limitations in §6.3.

To decrease the data complexity and increase the density, we define thirty value categories, covering a wide range of variable types, to map the continuous but sparse variable values into discrete bins. We call this process as *value quantization*, which is similar in design to the quantization in signal processing[1]. This simplification potentially helps the model to be more resilient to noise and outliers, allowing it to focus on learning the underlying execution patterns and relationships between variables, rather than being sensitive to specific instances or irregularities.

Representing Execution Coverage. While program state labels provide important information about the current state of the program, they do not capture information about how the program arrived at that state. To boost the training with more comprehensive execution features, besides the variable values, we also log the execution coverage during the execution, in terms of which lines are executed and which are not, and construct execution coverage features for the model to learn.

Results. We fine-tune and evaluate TRACED's performance using three tasks: static execution estimation, clone retrieval, and vulnerability detection. On statically predicting the program executions, TRACED substantially improves the statically pre-trained code models by 12.4% for execution path prediction and by 25.2% for runtime variable value predictions. TRACED also obtains state-of-the-art results in code understanding tasks: TRACED reports 91.2% MAP@R on CodeXGLUE-POJ104 [32], 50.4% F1 on ReVeal [8], and 65.9% accuracy on CodeXGLUE-defect-detection [32].

Contributions. We make the following contributions:

- We present a simplified and compact representation of program executions, including the program states and the execution coverage, to effectively guide code models to learn program semantics and reason about program behavior.
- We propose a novel multi-task pre-training strategy to jointly learn the static and dynamic code properties. As a result, the pre-trained model with our approach will be empowered with a decent execution awareness.
- We pre-train TRACED with the proposed trace representation and the execution-aware strategy and evaluate its performance on several downstream tasks. The experiment results demonstrate that TRACED significantly outperforms the statically pre-trained code models in these tasks.
- We will publicly release our data, code, and pre-trained models at https://github.com/ARiSE-Lab/TRACED_ICSE_24.git.

2 OVERVIEW

Figure 2 shows the overview of TRACED, consisting of three main stages: (1) tracing the source code and engineering the features, (2) execution-aware pre-training using the program traces, and

[1] https://en.wikipedia.org/wiki/Quantization_(signal_processing)

(3) loading the pre-trained weights and performing task-specific fine-tuning.

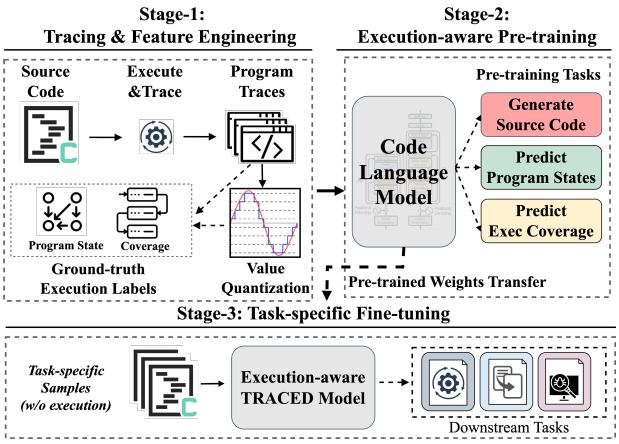

Figure 2: Overview of TRACED workflow.

Stage-1: Tracing & Feature Engineering. The goal of this stage is to prepare the data for pre-training. The process begins with providing a source program and its executable inputs. The first step is to execute the program with each input to generate corresponding traces. The traces record the runtime variable values, together with the execution coverage, logging the full execution history of the program and revealing the changes to program states throughout execution. To reduce the complexity and sparsity of the data, and make it easier for the model to learn patterns and relationships between the variable values, we quantize the concrete runtime values recorded in the traces into pre-defined value ranges. The quantization process maps continuous values to a fixed set of discrete or bins. By quantizing the values, we create a finite set of possible outputs that can be used as ground-truth labels during training. After quantization, we create program state labels and execution coverage labels that will help the model to capture the program executions. The dataset finally ends up with a set of samples and labels, where each sample includes the source code with its program input and the labels represent the execution trace of this sample.

Stage-2: Execution-aware Pre-training with Traces. We utilize the pre-processed samples and labels obtained from Stage-1 to perform supervised pre-training. Specifically, we use a Transformer-encoder-based model [31] to learn the program traces and improve the model's understanding of program execution. The model could be either trained from scratch or loaded by the pre-trained weights of existing code language models. To achieve the goal of producing execution-aware code representation, we propose three pre-training objectives. The first objective is learning to generate the source code. We believe that understanding the naturalness of code text [20, 43] is fundamental for the model to capture more sophisticated signals such as program execution. This objective is implemented with masked language modeling (MLM), which masks a certain percentage of tokens in the source code and trains the model to reconstruct the masked tokens based on the surrounding context. The second objective is learning to predict the program states. By predicting program state labels that were generated in Stage-1, the model learns to capture the data flows and the side effects of code execution. The third objective is to predict the execution coverage. By predicting the execution coverage labels generated by Stage-1, the model learns to capture the dynamic control flow and helps the model understand how the program state is reached and evolving.

Stage-3: Task-specific Fine-tuning. Finally, we apply TRACED to several downstream tasks. We load the pre-trained weights of TRACED, fine-tune the model for a specific task, and keep updating the model weights. Fine-tuning does not require the program to be executed; rather, TRACED will reason about the execution statically with its learned execution signals during the pre-training, and learn to accomplish the task accordingly. In many useful applications, we would not have program traces available. We consider three downstream tasks for TRACED: static execution estimation, which includes execution coverage and runtime variable value predictions, clone retrieval, and vulnerability detection.

3 TRACING & FEATURE ENGINEERING

In this section, we introduce how TRACED builds the learnable features from program traces for models to learn the program executions.

3.1 Representing Program States

To imitate the way that human developers monitor variable values to understand program behavior, we propose to train neural models with the log of runtime variable values to recognize execution patterns and infer dynamic program behaviors in a way that is similar to human intuition. By taking the log of variable values during the execution, we can represent the program states in a more compact and interpretable form that is manageable for deep neural nets to process.

We build the program state by taking snapshots of variable values at program points during execution. When we take a snapshot at a specific time step, similar to the moment that the program is paused by a debugging breakpoint set right after line l, we maintain a value mapping, M, to map the variable to its current value, similar to the value mapping table of the debugger. To record the program state, we take the value snapshot after each line of execution and log the variables' current values.

Definition: *Program State.* Formally, we define the program state after the execution of a specific line, l, as $s(l)$, represented as a set of variable values at this moment:

$$s(l) = \{M(v, l) \mid v \in V,\ l \in L\}$$

V represents the set of all traced variables, and L is the set of lines with source code. Figure 3 shows an illustrative example of a simple factorial program and the comments after the source code indicate the program state after the execution of that line. Also, we do not log the program state for lines without executable code, such as line-8 of Figure 3.

Note that a source code line could be executed multiple times due to a loop or recursion. While a more detailed representation of program execution might provide additional insights, it also increases the complexity and computational requirements of the model. As a trade-off between the complexity and performance, we use the last

```
1  // INPUT: 4
2  int factorial() {
3    int x, y; // {'x': 32767, 'y': 32767}
4    x = atoi(argv[1]); // {'x': 4, 'y': 32767}
5    if (x < 0) { // {'x': 4, 'y': 32767}
6      y = -1;
7      return y;
8    }
9    y = 1; // {'x': 4, 'y': 1}
10   for (int i = 1; i <= x; i++) // {'x': 4, 'y': 24, 'i': 5}
11   {
12     y *= i; // {'x': 4, 'y': 24, 'i': 5}
13   }
14   return y; // {'x': 4, 'y': 24, 'i': 5}
15 }
```

Figure 3: Program states with concrete runtime values.

occurring execution of each line to finalize the program states, so that $s(l)$ keeps getting updated until the execution terminates.

We apply such a trade-off based on the observations of real executions. Specifically, the last occurring values are typically sufficient to capture the results of loops and recursions. For example, when calling a recursive function, only the last occurring value(s) of returned variable(s) will be taken to fulfill the following execution of the caller. Similarly, the final values when loops finish will take part in the future execution. As shown in line-12 of Figure 3, variable y gets multiplied inside a loop to calculate the factorial. Its value changes in each iteration, but it is less informative to reason about the program's overall behavior, as only the final value is used as the return value (line-14). Thus, we would represent y using the value from the last occurring execution of the loop.

3.2 Quantized Variable Values

As we introduced in §1, the distribution of concrete values is sparse and complex, consequently difficult for a statistical model to fit. In addition, concrete values are not always necessary. Some common program behaviors are accompanied by extremely large or small variable values – for example, in C, uninitialized variables are often set to zero or uncommonly large variables, but the concrete values are not meaningful because they depend only on the data remaining on the stack, which could be randomly large or small. The model could represent such behaviors by estimating the value ranges of variables without accurately predicting their concrete values which are not informative or meaningful. Figure 3 displays some of these cases: after the execution of line-3, x and y are uninitialized and randomly initiated as 32,767, which has no concrete meaning but only makes the training data noisy and sparse.

To reduce the data complexity and increase the density, we define 30 categories for quantized values in Table 1. To comprehensively represent the variable values, the proposed quantized categories consider both types, *i.e.,* the data types and value types, that are statically defined, and the dynamic runtime values. Our quantized categories cover the most common variable types and value types, which we have found sufficient to capture important program execution behaviors and relationships. By focusing on the most frequent value types, we can capture the essential features of program execution. This makes our approach effective at capturing the generalized program execution behaviors and patterns. We empirically illustrate our quantization strategy's effectiveness in § 6.3.

Table 1: TRACED's design of quantized variable values.

Data Type	Value Types	Concrete Value	Quantized Value
Basic	Integer	$0 < v \leq 10,000$	Positive Regular
		$10,000 < v$	Positive Large
		0	Zero
		$-10,000 \leq v < 0$	Negative Regular
		$v < -10,000$	Negative Large
	Float/Double	$0.0 < v \leq 1.0$	Positive Small
		$1.0 < v \leq 10,000.0$	Positive Regular
		$10,000.0 < v$	Positive Large
		0.0	Zero
		$-1.0 < v < 0$	Negative Small
		$-10,000.0 \leq v < -1.0$	Negative Regular
		$v < -10,000.0$	Negative Large
	Character	'\0'	Null
		$v \in$ {a-zA-Z}	Alphabetic
		$v \neq$ '\0'; $v \notin$ {a-zA-Z}	Non-alphabetic
	Boolean	0	False
		1	True
	Void	-	Void
Array	Integer	$[v_1, v_2, \ldots, v_n]$;	Initialized
		$quantize(v_i) \in$ Integer	Not Initialized
	Float/Double	$[v_1, v_2, \ldots, v_n]$;	Initialized
		$quantize(v_i) \in$ Float/Double	Not Initialized
	Character	"⟨string⟩"	Initialized
			Not Initialized
Pointer	Integer	0x0	Null
		Not 0x0	Not Null
	Float/Double	0x0	Null
		Not 0x0	Not Null
	Character	0x0	Null
		Not 0x0	Not Null

3.3 Building Learnable Labels for Code Models

We used supervised pre-training with traces. We construct labels for code models to learn two main perspectives of execution: program states and execution coverage.

Program State Labels. As we discussed in previous sections, we first trace the program variables during execution and log their runtime values. We then quantize these values into pre-defined categories. This process results in a sequence of program states, each represented by a set of quantized variable values (as shown in Figure 3), and we build the learnable features for the code model on top of these program states. Specifically, we build labels for variables that can be quantized into Table 1's categories and train the model to predict these labels given their source code representations (§ 4.1.2). The label for each variable is represented as a tuple: *(data type, value type, quantized value)*. For example, in Figure 3, the label of variable X occurring at line-3 is *(Basic, Integer, Positive Large)*, as the current value of X is 32,767. We build such labels for all occurrences of valid variables that can be quantized, and the set combining all labels is considered as the program state labels of the code sample.

Execution Coverage Labels. To unify our design and reduce the complexity of the model's learning process, we also build execution coverage labels for each occurrence of variables, aligning with the program state labels. Concretely, we specify whether a variable is covered or not. The variables within the executed lines will be regarded as covered, and those within the unexecuted lines will be labeled as not covered by execution. For example, in Figure 3, Line-6 is not executed, so y at this line has the program state label of (Basic, Integer, Not Covered), while y at line-9 is executed and

has the program state label with concrete quantized value of (Basic, Integer, Positive Regular).

4 MODEL

In this section, we explain the details of TRACED's components and learning objectives during pre-training and fine-tuning.

Model Architecture. Figure 4 shows the high-level architecture of TRACED's pre-training. The backbone of TRACED is a 12-layer Transformer encoder, similar to BERT [9] and RoBERTa [31], which learns the generic code representations. On top of the backbone Transformer layers, TRACED stacks multiple multi-layer-perceptron (MLP) layers as prediction heads for different tasks. During the pre-training, as shown in Figure 4, TRACED applies a language model prediction head, *i.e.,* LM layer, to predict the masked token given its contextualized representation, a program state prediction head to predict the program states labels that we defined in § 3.3, and an execution coverage head to prediction the execution coverage labels. For the task-specific fine-tuning, the backbone Transformer layers are loaded with the pre-trained weights, while the prediction heads are replaced by a newly initialized head customized for the specific downstream task.

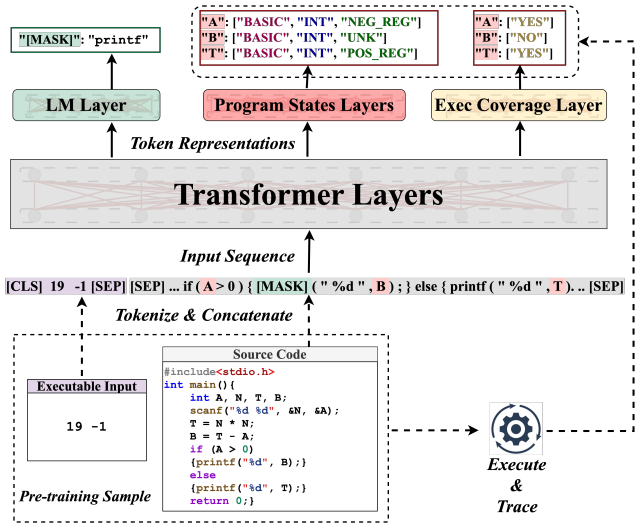

Figure 4: High-level model architecture of TRACED. In the labels for program state layers, NEG_REG means "Negative Regular", UNK means "Unknown", and POS_REG means "Positive Regular", which we have defined in Table 1.

4.1 Execution-aware Pre-training

4.1.1 Model Input of Pre-training. Each pre-training sample includes the source code of an executable program and a valid executable input. As shown in Figure 4, the executable input and the source code are flattened and concatenated as one sequence. To distinguish the input from the source code, as they are different modalities, TRACED uses special `[SEP]` tokens to separate them and indicate individual positions. To alleviate the out-of-vocabulary concern of programming languages [25], TRACED takes a pre-trained SentencePiece [28] subword tokenizer with vocabulary size of 50,000. It uses this tokenizer to divide the concatenated sequence into a new sequence of sub-tokens.

Formally, we define the executable inputs as $E = \{e_1, ..., e_i\}$ and flattened source code as $C = \{c_1, ..., c_j\}$, then the final model input will be $\mathcal{I}$ ={`[CLS]`, e_1, ..., e_i,`[SEP]`,`[SEP]`, c_1, ..., c_j,`[SEP]`}. TRACED truncates the executable inputs and the source code separately if they are too long. TRACED sets the maximum length of the executable input sequence to 64 tokens, and the source code to 960 tokens. These numbers are selected based on the statistics of executable inputs' length of our pre-training dataset (§5.2.1), and fit the rest of the model input with source code.

Note that the execution traces are not part of the model input, but are used as ground truth labels for the model to predict during pre-training.

4.1.2 Learning Execution-aware Code Representations with Traces. TRACED is pre-trained with multiple objectives to jointly capture the static and dynamic perspectives of the source code.

Learning Code Text. Learning code text is the essential first step toward understanding the execution of a program, as code text is the primary source of capturing the code naturalness [20] and other static properties. We implement the code text learning objective by adapting the masked language model objective [9, 16, 31]. Specifically, given the model input sequence, $\mathcal{I}$, TRACED randomly chooses 15% of tokens [9, 31] only from the source code sequence C part and replaces with the special `[MASK]` token (*e.g.,* `printf` in Figure 4 is masked). It leaves the executable input sequence E as is. The model is trained to encode the context of `[MASK]` into its code representation, r_{masked}, and reconstruct the concrete masked tokens conditioned on the representation. We represent the loss of learning code text as:

$$\mathcal{L}_{code-text} = \sum_{masked} -logP(c_{masked} \mid r_{masked}) \quad (1)$$

In Figure 4, the LM (Language Model) layer receives the masked token representation generated by the last Transformer layer. The LM layer then predicts the concrete tokens by mapping the token representation to the probability of each token in the vocabulary, using an MLP (Multi-Layer Perceptron) layer. This process can be thought of as a classification task, where the number of classes is equal to the size of the vocabulary. The goal is to learn a mapping from the masked token representation to the most probable token in the vocabulary, given its context.

Learning Program States. The second pre-training objective, program state prediction (PSP), is designed to enable the model to learn program execution behavior by predicting the program state labels of the traced variables. These program state labels, as defined in §3.3, contain information about the data types, value types, and quantized values of the variables at the end of the program execution. Specifically, TRACED first identifies the variable tokens in the source code sequence, denoted as $\{c_{var} \mid c_{var} \in V\} \subseteq C$, where V is the set of all traced variables and C is the source code sequence. It then extracts the representation, r_{var}, of each variable token and feeds it into the program state layer. The program state layer predicts the variable's joint likelihood of being the ground-truth data type, d_{var}, value type, t_{var}, and quantized value, q_{var}. Note

that if a variable is tokenized as multiple sub-tokens, all belonging sub-tokens share the same program state label. Finally, TRACED computes the loss of PSP as the sum of the losses of all variable tokens used for predicting their program states. Mathematically, the loss is expressed as follows:

$$\mathcal{L}_{program-state} = \sum_{var} -logP(\ d_{var}, t_{var}, q_{var} \mid r_{var}) \quad (2)$$

Learning Variable Coverage. The third pre-training objective, variable coverage prediction (VCP), aims to learn the execution coverage, which is crucial for understanding the control flow of the code given a specific input. Similar to the PSP objective, VCP targets making predictions for variable tokens. Also, sub-tokens belonging to the same variable will be assigned the same coverage label. The loss of VCP is as follows:

$$\mathcal{L}_{var-cov} = \sum_{var} -logP(cov_{var} \mid r_{var}) \quad (3)$$

For the efficiency of joint optimization, we share the weights between the program state layers and the execution coverage layers, as they are instinctively both classifiers optimized by cross-entropy loss. Concretely, the coverage label will be learned jointly with the quantized value: if a variable is covered, it will be assigned a specific quantized value label, and otherwise, it will be assigned as "Not Covered".

Finally, TRACED combines the losses of all three objectives and computes their sum as the final loss of a pre-training sample. It back-propagates the gradients through both the prediction layers and the backbone Transformer layers to update their weights. We denote the full set of TRACED's learnable parameters as θ and represent the loss as follows:

$$\mathcal{L}(\theta) = \mathcal{L}_{code-text}(\theta) + \mathcal{L}_{program-state}(\theta) + \mathcal{L}_{var-cov}(\theta) \quad (4)$$

4.2 Task-specific Fine-tuning

TRACED loads the model weights of Transformers layers, which are pre-trained to produce execution-aware code representations, and further fine-tunes the model for downstream tasks. We consider three downstream tasks as the main applications for TRACED: (1) Static estimation of program execution which includes both execution coverage prediction and runtime variable value prediction; (2) Semantic Clone Retrieval; (3) Vulnerability Detection.

Static Execution Estimation. Our goal of pre-training is to encode the execution patterns into the code representation, so the model could estimate the program execution statically. As a direct application, TRACED fine-tunes the model to predict (1) the execution coverage and (2) runtime variable values using source code and program input. TRACED evaluates the fine-tuned model to estimate the execution of unseen programs in the same way.

Specifically, for execution coverage prediction, TRACED identifies all the branching statements to locate the branches, $B = \{b_1, b_2, ..., b_m\}$, within the source code. It trains the model to predict a binary label, 0 means the branch is not covered by the current execution and 1 means covered, for each $b_i \in B$. For the model's convenience to make predictions, the special token `[MASK]` is inserted at the beginning of each branch. For example, the following `if-else` has two branches that are pre-processed for branch prediction: `if (condition) {[MASK] ...} else {[MASK] ... }`. During the fine-tuning, the Transformer layers learn to encode the branch information into the corresponding `[MASK]` token representation with the built-in bi-directional attention and positional encoding. Then the classification head takes `[MASK]` representations to predict whether a branch is covered by the current execution. For variable value prediction, TRACED identifies variables, $V = \{v_1, v_2, ..., v_n\}$ and trains the model to predict their quantized values (§3.2) during the execution.

Semantic Clone Retrieval. Detecting semantic clones is significant for software maintenance [26, 30], yet very challenging in practice since the token and syntactic structures overlap among semantic clones may be quite limited. This task requires the model to estimate the program behaviors without executing the programs and capture the similarity among them. It evaluates the model's semantic reasoning capacity to identify the code similarity and retrieve clones: given a program as a query, and an arbitrary collection of programs as candidates, the model needs to identify the query's semantic clones from possibly thousands of candidates.

Vulnerability Detection. Vulnerability detection is a crucial task in software security, aiming to identify potential security vulnerabilities in software code that could be exploited by attackers. The vulnerabilities may exist due to various reasons, including programming errors, design flaws, or configuration issues. Detecting these vulnerabilities early in the software development lifecycle can prevent potential attacks, mitigate risks, and save resources. We fine-tune TRACED's pre-trained model on datasets consisting of vulnerable and non-vulnerable code samples, so the model learns to classify code functions as vulnerable or non-vulnerable by estimating their execution behavior.

5 EXPERIMENTAL SETUP

5.1 Trace Collection

In this section, we explain how we traced the dynamic information in programs to produce concrete traces, given the source code and program input.

First, we compile the program using `gcc` with the options `-g -O0`. Option -g preserves debug information, which is necessary in order to read variables and source code locations using the debugger, and option `-O0` disables compiler optimizations, which could optimize out some variables thus preventing them from being read at runtime. We use this option because we seek to model the semantics of the *source code* in terms of variable values rather than the optimized machine code.

Second, we load the program with the given standard input redirected to `stdin` and attach the `gdb`[2] debugger, using the Python API to implement the tracing command. Starting from the entry point (`main`), we execute the program one line at a time using the `step` command. At each line, we print out the concrete values of all variables in scope. We also set breakpoints at the entry of each user-defined function, where we log the values of each parameter. For numeric types, we simply log their string representation. For

[2] https://www.sourceware.org/gdb

char and char * (string) types, we log the human-readable values of the chars/strings. We use gdb's pretty-printer to print struct types and statically allocated array types, such as int[<size>]. For pointer types, we print the memory address of the pointer as a hex code. We only traced the functions that were defined in the source code and skipped over all standard library functions.

5.2 Dataset

5.2.1 Pre-training Dataset. IBM's CodeNet Dataset [41] includes 4,053 programming challenges for several programming languages from the AIZU Online Judge and AtCoder platforms, and each problem has up to thousands of implementations submitted by distinct programmers. In this work, we focus on the C language as the main resource for the pre-training and downstream tasks, so we build our pre-training dataset with programming challenges that have C solutions. Besides the large number of samples and the complexity of programming challenges, we choose CodeNet to build our datasets as it maintains at least one and at most twenty executable inputs for each challenge, so we could execute and trace the implementations of the challenge, and consequently build our execution labels for the model to learn.

Out of 1,900 programming challenges with C solutions, we select 1,805 of them to build the pre-training dataset and leave the other 95 problems as held-out problems for evaluating the model's capacity for the downstream static execution estimation task. Splitting samples strictly by challenge effectively avoids the issue of data leakage from the training set to the held-out set. We randomly sample up to 200 execution traces for each challenge, and this ends up with 121,319 training traces.

5.2.2 Downstream tasks. In this section, we introduce the datasets we use for each downstream task and explain the corresponding evaluation metrics. The statistics of these datasets are in Table 2.

Static Execution Estimation. We build the dataset for this task using CodeNet. We build the training samples from the 1,805 challenges that have been selected by the pre-training, and build evaluation samples from the held-out 95 challenges to avoid model memorization and data leakage.

Metrics. For the execution coverage prediction, we consider evaluation metrics in two granularities: full execution path and branch coverage. Concretely, for a sample with m branches, we denote the full set of their labels as $LB = \{lb_1, lb_2, ..., lb_m\}$, and the model prediction set as $\hat{LB} = \{\hat{lb_1}, \hat{lb_2}, ..., \hat{lb_m}\}$. If $LB == \hat{LB}$, we regard the prediction as matching the full execution path. For the branch coverage, we compute the occurrence of $lb_i == \hat{lb_i}$, where $1 \leq i \leq m$, and report the accuracy, precision, recall, and F1. Similarly, for the n quantized variable values within the program, $QV = \{qv_1, qv_2, ..., qv_m\}$, our model makes predictions as $\hat{QV} = \{\hat{qv_1}, \hat{qv_2}, ..., \hat{qv_m}\}$. If $QV == \hat{QV}$, we say the model accurately predicts the full execution. For the individual value match, we compute the occurrence of $qv_i == \hat{qv_i}$ and report the accuracy.

Semantic Clone Retrieval. We use CodeXGLUE-POJ104 [32, 33] as the dataset for this task. CodeXGLUE-POJ104 contains 104 programming challenges, and each has 500 C/C++ solutions submitted by different programmers. CodeXGLUE [32] reconstructs it as a public benchmark by splitting the dataset into Train (64 challenges), Dev (16 challenges), and Test (24 challenges) sets, with no overlapped challenge between any two sets.

Metrics. MAP@R (Mean Average Precision @ R)[3] is the main metric of this task, where we follow the design of the CodeXGLUE benchmark. Average precision at R is a common metric to evaluate the quality of information retrieval; it measures the average precision scores of a set of the top-R clone candidates presented in response to a query program. The "R" for CodeXGLUE is 499 as it has 500 solutions for each challenge.

Vulnerability Detection. We utilized three publicly available datasets: REVEAL (RV) [8], D2A [54], and CodeXGLUE-Devign (CXG) [32, 55]. The REVEAL dataset was curated by Chakraborty *et al.* to simulate a real-world scenario where bugs are relatively rare, resulting in a ratio of approximately 1:10 between buggy and benign samples. The D2A dataset is a balanced dataset focusing on bug-fixing commits. It labels the previous version of modified functions as buggy and the fixed version as benign. Finally, the CodeXGLUE-Devign dataset, introduced by Zhou *et al.*, is also a balanced dataset that has been reconstructed as a public benchmark by CodeXGLUE, ensuring that all models can be evaluated using the same train/valid/test splits.

Metrics. REVEAL is an imbalanced dataset, so we use F1 as the evaluation metric. D2A and Devign are balanced datasets, so we follow the original benchmark to report the classification accuracy.

Table 2: Details of downstream tasks datasets.

Task	Dataset	Train	Valid	Test
Execution Estimation	CodeNet	121,319	13,116	13,116
Clone Detection	CXG-POJ104	32,000	8,000	12,000
Vulnerability Detection	REVEAL	15,867	2,268	4,535
	D2A	4,644	597	619
	CXG-Devign	21,854	2,732	2,732

5.3 Model Configuration

TRACED's backbone is a standard RoBERTa$_{\text{BASE}}$ architecture [31] with 12 layers of Transformer-encoder, and each layer has 12 attention heads and the hidden dimension is 768. TRACED is initialized with the pre-trained weights from UnixCoder [17][4], and we use its BPE tokenizer to split the rare tokens into BPE sub-tokens. The maximum sequence length is 1024 BPE tokens, and the longer sequence will be truncated. When the code sample is paired with executable inputs, the maximum length for the executable input is 64, and the source code is 960. Our experiments are conducted on 2 × 24GB NVIDIA GeForce RTX-3090 GPUs. We further pre-train the model for 10 epochs to learn the program execution with two learning rates, 5e-5 and 2e-5, and report the best-performing models for downstream tasks. For all the fine-tuning tasks, we use the learning rate of 8e-6. Learning rates typically decrease for later phases [10, 16, 18], so TRACED follows the same design. We use Adam optimizer [27] with the linear learning rate decay. Our model is implemented mainly with Pytorch [12] and Huggingface [14].

[3] https://en.wikipedia.org/wiki/Evaluation_measures_(information_retrieval)#Mean_average_precision

[4] Specifically, we load unixcoder-base-nine, as its pre-training considers C language code samples: https://huggingface.co/microsoft/unixcoder-base-nine. Note that this checkpoint is pre-trained only with the MLM objective, while the original paper [17] reports other better-performing variants that are not released publicly.

6 EVALUATION

In this section, we ask the following four RQs:

- **RQ1:** How effective is TRACED in statically estimating the program execution?
- **RQ2:** How does our proposed training strategy contribute to learning the program execution?
- **RQ3:** Is our proposed quantized values for programs effective in guiding the model to learn program executions?
- **RQ4:** How does TRACED perform *w.r.t.* statically pre-trained baselines for code understanding tasks?

6.1 RQ1. Effectiveness of TRACED in Static Estimation of Execution

In this section, we demonstrate the effectiveness of TRACED in statically estimating program execution. The evaluation is more challenging and realistic than TRACED's pre-training as it requires the model to predict not only for individual variables but also branches and the full execution path.

Baseline. In this RQ, we mainly compare the execution-aware TRACED with UnixCoder [17]. Now we explain the reasons for this choice. First, TRACED is initialized with the pre-trained UnixCoder weights, so comparing TRACED with the UnixCoder performance is a direct assessment of the impact of our proposed pre-training. Second, UnixCoder reports the state-of-the-art performance in many tasks, including clone detection, code search and summarization, and code generation and completion, significantly outperforming other pre-trained code models, such as CodeBERT [16] and GraphCodeBERT [18]. Third, it consumes up to 1,024 tokens, while most pre-trained code models [1, 6, 16, 18, 49] take at maximum 512 tokens. By consuming longer sequences, UnixCoder is able to handle longer programs and make complete predictions without truncating code in many cases. As TRACED is also designed to consume 1,024 tokens, it is not fair to compare it in this task with baselines with a maximum length of 512, as the baselines will necessarily consider fewer branches for prediction.

Table 3: Performance on static execution estimation.

Model	Coverage					Runtime Value	
	Full Path	Branch				Full Exec	Var
	Acc	Acc	Prec	Rec	F1	Acc	Acc
UnixCoder	63.7	79.7	81.7	85.4	83.5	39.3	87.8
TRACED	**71.6**	**83.1**	84.6	**88.1**	**86.3**	**49.2**	**89.2**
-w/o MLM	70.4	82.6	**85.3**	86.0	85.6	49.0	**89.2**
-w/o PSP	69.0	81.4	83.0	86.9	84.9	44.0	87.4
-w/o VCP	66.1	80.3	82.4	85.6	84.0	46.7	89.0
-MLM-only	65.6	81.0	83.1	86.0	84.6	43.0	87.5

Result. The comparison is shown in Table 3, Row-1 *vs.* Row-2. TRACED significantly outperforms UnixCoder in the static estimation of execution coverage and dynamic values of variables, especially when the evaluation granularity is coarse, *i.e.,* full execution path (Full Path column in Table 3) and the runtime values of the full execution (Full Exec column in Table 3). TRACED correctly predicts the complete execution paths for 71.6% held-out samples and accurately predicts all variable values for 49.2% executions, revealing the execution-aware pre-training improves over UnixCoder's performance by 12.4% and 25.2%, respectively.

Case Study with Qualitative Examples. We present two qualitative examples in Figure 5 and 6 to concretely compare TRACED with UnixCoder in execution coverage and runtime value predictions, respectively. Both samples have simple execution logic from the human perspective, but the statically pre-trained UnixCoder still fails to correctly estimate them. Figure 5 illustrates that UnixCoder is not sensitive to distinct inputs that trigger different execution coverage, while TRACED is able to determine the numerical relations among varied values. Figure 6 illustrates TRACED's capacity in exposing abnormal program behaviors.

```
//Input: 19 100
#include <stdio.h>
int main(){
    int A, N, T, B;
    scanf("%d %d", &N, &A);
    T = N * N;
    B = T - A;
    if (A > 0) {printf("%d", B);} // Branch-1
    else {printf("%d", T);} // Branch-2
    return 0;
}
```

UnixCoder Predictions (Wrong)

Branch-1: Not executed

Branch-2: Not executed

TRACED Predictions (Correct)

Branch-1: Executed

Branch-2: Not Executed

Figure 5: The qualitative example of execution coverage prediction. The source code is the same as Figure 1, but the input triggers a different execution path. TRACED correctly flips the prediction while UnixCoder remains the same prediction.

```
//Input: 4 4320 4320 4320
#include <stdio.h>
int main (void) {
    int n, a, max = 0, sum = 0, i;
    for (i = 0; i < n; i++){ // Quantized value of n?
        scanf("\%d", &a);
        if (a > max) max = a;
        sum += a;
    }
    printf("\%d\\n" , sum - max / 2);
    return 0;
}
```

UnixCoder Prediction (Wrong)

n: Zero

TRACED Prediction (Correct)

n: Negative Large

Figure 6: The qualitative example of runtime value prediction. The sample contains a vulnerability of type CWE-457 "Use of Uninitialized Variable". The uninitialized `n`, which is randomly assigned as -32767, is used in the `for-loop`. TRACED successfully exposes this abnormal behavior statically by identifying *n* as a "Negative Large" value while UnixCoder fails. Predictions of other variables are hidden for better illustration.

Result-1: With a similar number of learnable parameters, TRACED outperforms the state-of-the-art pre-trained code model in the static estimation of program execution task. Our proposed pre-training successfully encodes the execution awareness into TRACED's code representations.

6.2 RQ2. Effectiveness of TRACED's Pre-training Objectives

One of the main contributions of this paper is proposing multi-task pre-training to effectively learn the execution-aware code representations. In this RQ, we study the effectiveness and contribution of each of TRACED's objectives, and consequently illustrate the importance of the multiple tasks.

To conduct these experiments, we remove one pre-training objective at a time and pre-train the variant with exactly the same setup as the main model. Then we fine-tune the variant on the static execution estimation task and compare the performance with the main model. We also consider a variant that is pre-trained on our dataset but only with MLM objectives. The results are shown in Row 3-6 of Table 3. Removing any objective hurts TRACED's performance, suggesting that comprehensively learning both static and dynamic code properties is more effective than learning one perspective alone.

Result-2: TRACED's multi-task pre-training helps the model comprehensively learn both static and dynamic aspects of source code. Removing any one of TRACED's three pre-training objectives noticeably hurts the model's performance in statically estimating program executions.

6.3 RQ3. Effectiveness of TRACED's Quantized Variable Values

Another contribution of this paper is that the simplified and compact representation of program executions helps code models to capture dynamic code properties. In this RQ, we empirically reveal that the design of quantized variable values especially contributes to the effective learning of the code models, as it reduces the data sparsity of variable values but still defines sufficiently detailed value categories to distinguish dissimilar values.

To isolate the evaluation of TRACED's quantized values, we pre-train several variants by only recreating quantized value labels, *i.e.*, q_{var} in Equation 2, using different value abstraction strategies. For example, when we pre-train a variant studying the impact of concrete values, we replace TRACED's defined q_{var} with the concrete traced values. As different strategies abstract values at different granularities, it is not feasible to compare them for the value prediction task, since the coarse-grained strategy will benefit. Therefore, we only fine-tune the studied variants for the execution coverage prediction.

Baseline. First, we consider comparing with concrete values, as it is the most intuitive strategy to represent variable values. Then, we consider two data abstractions from LExecutor [45]: coarse and fine-grained. They share similar high-level intuition with us, mapping concrete values to pre-defined bins to reduce data complexity and consequently help the model's learning. Note that LExecutor's data abstraction serves a different goal than TRACED, and focuses on Python while TRACED focuses on C, so we could not directly reuse their pre-defined bins. As their definition of data abstraction is clear and straightforward, we re-implement their data abstraction for the C language and integrate it into our framework for comparison. We discuss and compare LExecutor with TRACED in more detail in the Related Work section (§7).

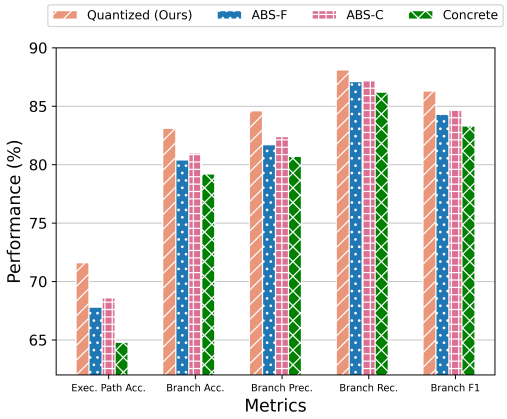

Figure 7: Comparing TRACED's design of quantized variable values with other value abstraction strategies.

Results. The comparison of value abstractions are shown in Figure 7. Unsurprisingly, concrete values report poor performance compared to other data abstractions, empirically revealing the difficulties for code models to fit sparse and complex data distributions. Interestingly, we notice both of LExecutor's abstractions perform slightly worse than TRACED. We speculate that LExecutor is not as sensitive as TRACED to numeric relations in the conditional statements, as they do not distinguish among small, regular, and large values. Note that execution coverage is not the main focus of LExecutor, so more fine-grained categories are not required to serve its goal, while they are empirically proven to be necessary for TRACED's scope.

Result-3: TRACED's quantized variable values directly contribute to the effectiveness of its execution-aware pre-training. It reduces the data sparsity of concrete values but defines sufficiently detailed value categories to distinguish dissimilar values for reasoning about execution paths.

6.4 RQ4. TRACED's Performance in Code Understanding Tasks

In this RQ, we study TRACED's performance on two code understanding tasks: semantic clone retrieval and function-level vulnerability detection. Note that samples for these tasks are not paired with executable inputs, so the model needs to reason about the general code semantics to make predictions.

Baselines. We consider five pre-trained code models with similar parameter sizes to TRACED. CodeBERT [16] pre-trains a RoBERTa

model with MLM and replaced token detection (RTD) tasks. GraphCodeBERT [18] is initialized with CodeBERT and continues pre-training with augmented data flow graphs to learn the static data dependencies. PLBART [1] and CodeT5 [49] both apply the sequence-to-sequence neural architecture, where PLBART adapts the BART [29] model to learn code translation and summarization, and CodeT5 adapts [42] to predict the missing code tokens and locate the identifiers. We also, again, consider UnixCoder as a baseline.

Table 4: Comparison of Clone Retrieval and bug detection.

Task	Clone Retrieval	Vulnerability Detection		
Dataset	**POJ-104**	**RV**	**D2A**	**CXG**
Metric	**MAP@R**	**F1**	**Acc**	**Acc**
CodeBERT	85.2	45.5	61.0	63.2
GraphCodeBERT	86.7	46.6	58.3	62.9
PLBART-base	75.9	46.9	61.7	63.3
CodeT5-base*	65.9	46.5	**62.1**	64.4
UnixCoder	89.5	47.4	61.2	65.3
TRACED	**91.2**	**50.4**	**62.1**	**65.9**

*CodeT5-base has 223M parameters, roughly twice as large as other baselines and TRACED. We report its performance as CodeT5-small has only 60M parameters and performs poorly, and CodeT5 does not provide a ~110M model.

Results. We show the results in Table 4. Even though the samples in these benchmarks do not have executable inputs, TRACED still outperforms the statically pre-trained models by a clear margin. We speculate the reason is that TRACED could estimate the general execution behaviors without specific inputs, and the program semantics regarding these two code understanding tasks could be better captured with such a general sense. Specifically, clone retrieval requires the model to identify the behavioral similarities of code as semantic clones mostly differ in code text and syntax. Also, vulnerable code with potential anomalies could be directly identified by TRACED in some cases like Figure 6.

Result-4: TRACED outperforms statically pre-trained models in clone retrieval and vulnerability detection tasks, suggesting TRACED's general estimation of execution helps it capture the code semantics more effectively.

7 RELATED WORK

Pre-trained Models for Source Code. The research community has shown a growing interest in developing pre-trained Transformer models for source code. These models can be broadly categorized into three primary architectures: Encoder-only [5, 6, 10, 16, 18, 24, 48], Decoder-only [2, 13, 50], and Encoder-decoder [1, 7, 15, 17, 35]. Encoder-only models predominantly employ MLM objective and sequence understanding tasks (*e.g.,* predicting next statement [24] and contrasting semantics [10]). This architecture excels at understanding the static code features. Decoder-only models, on the other hand, are typically trained by predicting code tokens in a left-to-right manner. This architecture focuses on generating code text based on learned patterns. The Encoder-decoder models combine the strengths of both Encoder-only and Decoder-only models and are pre-trained using various tasks, including denoising autoencoding for reconstructing wrongly permuted tokens [1], predicting missing identifiers in the code [49], and recovering method names from the source code [35].

These models primarily focus on learning the static aspects of source code but often miss out on capturing the dynamic properties of code execution. This limitation restricts these models from accurately inferring runtime behaviors, debugging issues, and understanding complex program states.

Modeling Program Execution. Pei et al. [38–40] proposed a series of pioneering works to learn the executions of *binary* programs with Transformer-based models. They used concrete values from registers, which are feasible in their scope because binary programs have a smaller space of possible values and effects compared to source code. On the other hand, our work focuses on encoding execution at the source code level by imitating the developers' code practice. Variables in source code have more complicated data and value types than machine registers. We introduce quantized values in order to decrease the data complexity and sparsity.

Several works [3, 4, 36, 44, 51, 52] have attempted learning to execute programs as a direct goal. Souza and Pradel [45] also proposed LExecutor to predict missing values during execution. While it shares similar intuition of mapping concrete values to discrete categories, LExecutor is distinct from TRACED in several perspectives. First, LExecutor focuses only on predicting the values, while TRACED proposes a general pre-training strategy to encode the comprehensive execution awareness, not only values but also execution coverage, into the code representation. Besides, to yield code representations at a better quality, TRACED jointly learns both code text and dynamic executions rather than sticking to a single perspective. Due to the distinct aims and designs, we empirically illustrate in RQ3 (§6.3) that LExecutor's value abstractions are not perfectly aligned with our scope.

Nie et al. [34] annotated programs with information about the program's possible executions without executing the code but provided only statically available information. Conversely, several works [19, 37, 46, 47] require dynamic traces as input. We show that TRACED's pre-training is able to encode the execution awareness into code representation and estimate the dynamic semantics with static information alone.

8 THREATS TO VALIDITY

Internal Validity. First, the current design of quantized value is not covering all variables within the program due to the complexity of their data structures, value ranges, and/or memory allocations. Second, currently, we only trace the program by feeding it valid and executable inputs which will not terminate the program or throw errors. This might make the model less capable of capturing program termination and error-throwing behaviors.

External Validity. At present, TRACED supports only the C programming language. This limitation is due to the reliance on the capabilities of the tracer used to log the execution history, which may not be readily available or equally effective for other programming languages. In order to extend TRACED's applicability, it is necessary to ensure that the tracer employed can accurately and

consistently capture the required information across different languages. Adapting TRACED to multiple languages would require the development or adaptation of tracers that can effectively handle the intricacies of each language and produce comparable results, enabling a consistent analysis of code behavior across a broader range of programming languages.

9 CONCLUSION

In this paper, we propose TRACED, an execution-aware pre-trained model that jointly learns the static and dynamic code properties, to address the limitation of existing, statically pre-trained code models. The evaluation empirically reveals that TRACED is more effective in estimating code execution statically than statically pre-trained models. TRACED also successfully transfers execution awareness to code understanding tasks.

ACKNOWLEDGMENTS

We appreciate all the anonymous reviewers for their thoughtful feedback and suggestions to improve this work.

This work was supported in part by NSF grants CCF-2313054, CCF-2313055, CCF-1815494, CCF-210740, CCF-1845893, IIS-2221943, and DARPA/NIWC Pacific N66001-21-C-4018. Any opinions, findings, conclusions or recommendations expressed herein are those of the authors and do not necessarily reflect those of the US Government, NSF, or DARPA.

REFERENCES

[1] Wasi Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2021. Unified Pre-training for Program Understanding and Generation. In *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*. Association for Computational Linguistics, Online, 2655–2668. https://www.aclweb.org/anthology/2021.naacl-main.211

[2] Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie J. Cai, Michael Terry, Quoc V. Le, and Charles Sutton. 2021. Program Synthesis with Large Language Models. *CoRR* abs/2108.07732 (2021). arXiv:2108.07732 https://arxiv.org/abs/2108.07732

[3] David Bieber, Rishab Goel, Dan Zheng, Hugo Larochelle, and Daniel Tarlow. 2022. Static Prediction of Runtime Errors by Learning to Execute Programs with External Resource Descriptions. https://openreview.net/forum?id=SIcz2sObJ-5

[4] David Bieber, Charles Sutton, Hugo Larochelle, and Daniel Tarlow. 2020. Learning to Execute Programs with Instruction Pointer Attention Graph Neural Networks. In *Advances in Neural Information Processing Systems*, Vol. 33. Curran Associates, Inc., 8626–8637. https://papers.nips.cc/paper/2020/hash/62326dc7c4f7b849d6f013ba46489d6c-Abstract.html

[5] Nghi D. Q. Bui, Yijun Yu, and Lingxiao Jiang. 2021. Self-Supervised Contrastive Learning for Code Retrieval and Summarization via Semantic-Preserving Transformations. In *SIGIR '21* (Virtual Event, Canada). 511–521. https://doi.org/10.1145/3404835.3462840

[6] Luca Buratti, Saurabh Pujar, Mihaela Bornea, Scott McCarley, Yunhui Zheng, Gaetano Rossiello, Alessandro Morari, Jim Laredo, Veronika Thost, Yufan Zhuang, and Giacomo Domeniconi. 2020. Exploring Software Naturalness through Neural Language Models. arXiv:2006.12641 [cs.CL]

[7] Saikat Chakraborty, Toufique Ahmed, Yangruibo Ding, Premkumar Devanbu, and Baishakhi Ray. 2022. NatGen: Generative pre-training by" Naturalizing" source code. In *2022 The ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. ACM.

[8] Saikat Chakraborty, Rahul Krishna, Yangruibo Ding, and Baishakhi Ray. 2021. Deep Learning based Vulnerability Detection: Are We There Yet. *IEEE Transactions on Software Engineering* (2021), 1–1. https://doi.org/10.1109/TSE.2021.3087402

[9] 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*. Association for Computational Linguistics, Minneapolis, Minnesota, 4171–4186. https://doi.org/10.18653/v1/N19-1423

[10] 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)*. Association for Computational Linguistics, Dublin, Ireland, 6300–6312. https://doi.org/10.18653/v1/2022.acl-long.436

[11] Yangruibo Ding, Baishakhi Ray, Devanbu Premkumar, and Vincent J. Hellendoorn. 2020. Patching as Translation: the Data and the Metaphor. In *35th IEEE/ACM International Conference on Automated Software Engineering* (Virtual Event, Australia) *(ASE '20)*. https://doi.org/10.1145/3324884.3416587

[12] Adam Paszke *et al.*. 2019. *PyTorch: An Imperative Style, High-Performance Deep Learning Library*.

[13] Mark Chen *et al.*. 2021. Evaluating Large Language Models Trained on Code. *CoRR* abs/2107.03374 (2021). arXiv:2107.03374 https://arxiv.org/abs/2107.03374

[14] Thomas Wolf *et al.*. 2020. Transformers: State-of-the-Art Natural Language Processing. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*. Association for Computational Linguistics, Online, 38–45. https://doi.org/10.18653/v1/2020.emnlp-demos.6

[15] Yujia Li *et al.*. 2022. Competition-Level Code Generation with AlphaCode. *ArXiv* abs/2203.07814 (2022).

[16] 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*. Association for Computational Linguistics, Online, 1536–1547. https://doi.org/10.18653/v1/2020.findings-emnlp.139

[17] Daya Guo, Shuai Lu, Nan Duan, Yanlin Wang, Ming Zhou, and Jian Yin. 2022. UniXcoder: Unified Cross-Modal Pre-training for Code Representation. https://doi.org/10.48550/ARXIV.2203.03850

[18] Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie LIU, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, Michele Tufano, Shao Kun Deng, Colin Clement, Dawn Drain, Neel Sundaresan, Jian Yin, Daxin Jiang, and Ming Zhou. 2021. GraphCode{BERT}: Pre-training Code Representations with Data Flow. In *International Conference on Learning Representations*. https://openreview.net/forum?id=jLoC4ez43PZ

[19] 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 (ESEC/FSE 2018)*. Association for Computing Machinery, New York, NY, USA, 163–174. https://doi.org/10.1145/3236024.3236085

[20] Abram Hindle, Earl T. Barr, Zhendong Su, Mark Gabel, and Premkumar Devanbu. 2012. On the Naturalness of Software. In *Proceedings of the 34th International Conference on Software Engineering* (Zurich, Switzerland) *(ICSE '12)*. IEEE Press, 837–847.

[21] Nan Jiang, Kevin Liu, Thibaud Lutellier, and Lin Tan. 2023. Impact of Code Language Models on Automated Program Repair. arXiv:2302.05020 [cs.SE]

[22] Nan Jiang, Thibaud Lutellier, Yiling Lou, Lin Tan, Dan Goldwasser, and Xiangyu Zhang. 2023. KNOD: Domain Knowledge Distilled Tree Decoder for Automated Program Repair. arXiv:2302.01857 [cs.SE]

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

[24] Aditya Kanade, Petros Maniatis, Gogul Balakrishnan, and Kensen Shi. 2020. Learning and evaluating contextual embedding of source code. In *ICML 2020*.

[25] Rafael-Michael Karampatsis, Hlib Babii, Romain Robbes, Charles Sutton, and Andrea Janes. 2020. Big Code != Big Vocabulary: Open-Vocabulary Models for Source Code. In *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*. 1073–1085.

[26] Seulbae Kim, Seunghoon Woo, Heejo Lee, and Hakjoo Oh. 2017. VUDDY: A Scalable Approach for Vulnerable Code Clone Discovery. In *2017 IEEE Symposium on Security and Privacy (SP)*. 595–614. https://doi.org/10.1109/SP.2017.62

[27] Diederik P. Kingma and Jimmy Ba. 2015. Adam: A Method for Stochastic Optimization. *CoRR* abs/1412.6980 (2015).

[28] Taku Kudo and John Richardson. 2018. SentencePiece: A simple and language independent subword tokenizer and detokenizer for Neural Text Processing. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*. Association for Computational Linguistics, Brussels, Belgium, 66–71. https://doi.org/10.18653/v1/D18-2012

[29] Mike Lewis, Yinhan Liu, Naman Goyal, Marjan Ghazvininejad, Abdelrahman Mohamed, Omer Levy, Veselin Stoyanov, and Luke Zettlemoyer. 2020. BART: Denoising Sequence-to-Sequence Pre-training for Natural Language Generation, Translation, and Comprehension. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*. Association for Computational Linguistics, Online, 7871–7880. https://doi.org/10.18653/v1/2020.acl-main.703

[30] Zhen Li, Deqing Zou, Shouhuai Xu, Hai Jin, Hanchao Qi, and Jie Hu. 2016. VulPecker: An Automated Vulnerability Detection System Based on Code Similarity Analysis. In *Proceedings of the 32nd Annual Conference on Computer Security Applications* (Los Angeles, California, USA) *(ACSAC '16)*. 201–213. https://doi.org/10.1145/2991079.2991102

[31] 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. *CoRR* abs/1907.11692 (2019). arXiv:1907.11692 http://arxiv.org/abs/1907.11692

[32] Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin B. Clement, Dawn Drain, Daxin Jiang, Duyu Tang, Ge Li, Lidong Zhou, Linjun Shou, Long Zhou, Michele Tufano, Ming Gong, Ming Zhou, Nan Duan, Neel Sundaresan, Shao Kun Deng, Shengyu Fu, and Shujie Liu. 2021. CodeXGLUE: A Machine Learning Benchmark Dataset for Code Understanding and Generation. *CoRR* abs/2102.04664 (2021).

[33] 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 Thirtieth AAAI Conference on Artificial Intelligence*. 1287–1293.

[34] Pengyu Nie, Rahul Banerjee, Junyi Jessy Li, Raymond J. Mooney, and Milos Gligoric. 2023. Learning Deep Semantics for Test Completion. arXiv. https://doi.org/10.48550/arXiv.2302.10166 arXiv:2302.10166 [cs].

[35] Changan Niu, Chuanyi Li, Vincent Ng, Jidong Ge, Liguo Huang, and Bin Luo. 2022. SPT-Code: Sequence-to-Sequence Pre-Training for Learning Source Code Representations. *CoRR* abs/2201.01549 (2022). arXiv:2201.01549 https://arxiv.org/abs/2201.01549

[36] Maxwell Nye, Anders Johan Andreassen, Guy Gur-Ari, Henryk Michalewski, Jacob Austin, David Bieber, David Dohan, Aitor Lewkowycz, Maarten Bosma, David Luan, Charles Sutton, and Augustus Odena. 2021. Show Your Work: Scratchpads for Intermediate Computation with Language Models. https://doi.org/10.48550/arXiv.2112.00114

[37] Jibesh Patra and Michael Pradel. 2022. Nalin: learning from runtime behavior to find name-value inconsistencies in jupyter notebooks. In *Proceedings of the 44th International Conference on Software Engineering*. ACM, Pittsburgh Pennsylvania, 1469–1481. https://doi.org/10.1145/3510003.3510144

[38] Kexin Pei, Jonas Guan, Matthew Broughton, Zhongtian Chen, Songchen Yao, David Williams-King, Vikas Ummadisetty, Junfeng Yang, Baishakhi Ray, and Suman Jana. 2021. StateFormer: fine-grained type recovery from binaries using generative state modeling. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2021)*. Association for Computing Machinery, New York, NY, USA, 690–702. https://doi.org/10.1145/3468264.3468607

[39] Kexin Pei, Dongdong She, Michael Wang, Scott Geng, Zhou Xuan, Yaniv David, Junfeng Yang, Suman Jana, and Baishakhi Ray. 2022. NeuDep: neural binary memory dependence analysis. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2022)*. Association for Computing Machinery, New York, NY, USA, 747–759. https://doi.org/10.1145/3540250.3549147

[40] Kexin Pei, Zhou Xuan, Junfeng Yang, Suman Jana, and Baishakhi Ray. 2020. Trex: Learning Execution Semantics from Micro-Traces for Binary Similarity. *CoRR* abs/2012.08680 (2020). arXiv:2012.08680 https://arxiv.org/abs/2012.08680

[41] Ruchir Puri, David S. Kung, Geert Janssen, Wei Zhang, Giacomo Domeniconi, Vladimir Zolotov, Julian Dolby, Jie Chen, Mihir R. Choudhury, Lindsey Decker, Veronika Thost, Luca Buratti, Saurabh Pujar, and Ulrich Finkler. 2021. Project CodeNet: A Large-Scale AI for Code Dataset for Learning a Diversity of Coding Tasks. *CoRR* abs/2105.12655 (2021). arXiv:2105.12655 https://arxiv.org/abs/2105.12655

[42] 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. *Journal of Machine Learning Research* 21, 140 (2020), 1–67. http://jmlr.org/papers/v21/20-074.html

[43] Baishakhi Ray, Vincent Hellendoorn, Saheel Godhane, Zhaopeng Tu, Alberto Bacchelli, and Premkumar Devanbu. 2016. On the "Naturalness" of Buggy Code. In *Proceedings of the 38th International Conference on Software Engineering* (Austin, Texas) *(ICSE '16)*. Association for Computing Machinery, New York, NY, USA, 428–439. https://doi.org/10.1145/2884781.2884848

[44] Scott Reed and Nando de Freitas. 2016. Neural Programmer-Interpreters. https://doi.org/10.48550/arXiv.1511.06279 arXiv:1511.06279 [cs].

[45] Beatriz Souza and Michael Pradel. 2023. LExecutor: Learning-Guided Execution. https://doi.org/10.48550/arXiv.2302.02343 arXiv:2302.02343 [cs].

[46] Ke Wang, Rishabh Singh, and Zhendong Su. 2018. Dynamic Neural Program Embeddings for Program Repair. https://openreview.net/forum?id=BJuWrGW0Z

[47] 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 (PLDI 2020)*. Association for Computing Machinery, New York, NY, USA, 121–134. https://doi.org/10.1145/3385412.3385999

[48] Xin Wang, Yasheng Wang, Fei Mi, Pingyi Zhou, Yao Wan, Xiao Liu, Li Li, Hao Wu, Jin Liu, and Xin Jiang. 2021. SynCoBERT: Syntax-Guided Multi-Modal Contrastive Pre-Training for Code Representation. https://doi.org/10.48550/ARXIV.2108.04556

[49] Yue Wang, Weishi Wang, Shafiq Joty, and Steven C.H. Hoi. 2021. CodeT5: Identifier-aware Unified Pre-trained Encoder-Decoder Models for Code Understanding and Generation. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing, EMNLP 2021*.

[50] Frank F Xu, Uri Alon, Graham Neubig, and Vincent J Hellendoorn. 2022. A Systematic Evaluation of Large Language Models of Code. *arXiv preprint arXiv:2202.13169* (2022).

[51] He Ye, Matias Martinez, Xiapu Luo, Tao Zhang, and Martin Monperrus. 2023. SelfAPR: Self-Supervised Program Repair with Test Execution Diagnostics. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering* (Rochester, MI, USA) *(ASE '22)*. Association for Computing Machinery, New York, NY, USA, Article 92, 13 pages. https://doi.org/10.1145/3551349.3556926

[52] Wojciech Zaremba and Ilya Sutskever. 2015. Learning to Execute. https://doi.org/10.48550/arXiv.1410.4615

[53] Andreas Zeller. 2005. *Why Programs Fail: A Guide to Systematic Debugging*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

[54] Yunhui Zheng, Saurabh Pujar, Burn Lewis, Luca Buratti, Edward Epstein, Bo Yang, Jim Laredo, Alessandro Morari, and Zhong Su. 2021. D2A: A Dataset Built for AI-Based Vulnerability Detection Methods Using Differential Analysis. In *2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. 111–120. https://doi.org/10.1109/ICSE-SEIP52600.2021.00020

[55] 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. In *Advances in Neural Information Processing Systems*. 10197–10207.