Predictive Program Slicing via Execution Knowledge-Guided Dynamic Dependence Learning

AASHISH YADAVALLY, University of Texas at Dallas, USA
YI LI, University of Texas at Dallas, USA
TIEN N. NGUYEN, University of Texas at Dallas, USA

Program slicing, the process of extracting program statements that influence values at a designated location (known as the slicing criterion), is helpful in both manual and automated debugging. However, such slicing techniques prove ineffective in scenarios where executing specific inputs is prohibitively expensive, or even impossible, as with partial code. In this paper, we introduce ND-Slicer, a predictive slicing methodology that caters to specific executions based on a particular input, overcoming the need for actual execution. We enable such a process by leveraging execution-aware pre-training to learn the dynamic program dependencies, including both dynamic data and control dependencies between variables in the slicing criterion and the remaining program statements. Such knowledge forms the cornerstone for constructing a predictive backward slice. Our empirical evaluation revealed a high accuracy in predicting program slices, achieving an exact-match accuracy of 81.3% and a ROUGE-LCS F1-score of 95.4% on Python programs. As an extrinsic evaluation, we illustrate ND-Slicer's usefulness in crash detection, with it locating faults with an accuracy of 63.9%. Furthermore, we include an in-depth qualitative evaluation, assessing ND-Slicer's understanding of branched structures such as `if-else` blocks and loops, as well as the control flow in inter-procedural calls.

CCS Concepts: • **Computing methodologies → Neural networks**; • **Software and its engineering → Language features**;

Additional Key Words and Phrases: AI4SE, Neural Networks, Dynamic Slicing, Dynamic Dependence Learning, Execution-Guided Learning, Large Language Models

ACM Reference Format:
Aashish Yadavally, Yi Li, and Tien N. Nguyen. 2024. Predictive Program Slicing via Execution Knowledge-Guided Dynamic Dependence Learning. *Proc. ACM Softw. Eng.* 1, FSE, Article 13 (July 2024), 22 pages. https://doi.org/10.1145/3643739

1 INTRODUCTION

Program analysis is the process of examining a program to reason about its properties, structures, or behaviors. A fundamental class of program analysis is *dependence analysis*, which focuses on reasoning about the dependencies of one component of a program, to some degree, on the other components. *Dynamic dependence analysis*, in particular, encompasses techniques to analyze such relationships during the execution of a program for specific inputs. This is essential for various tasks in software engineering, such as understanding and analyzing runtime behavior [33], detecting runtime errors or exceptions [25], debugging [40], testing [5], and optimizing programs [9].

Traditional methods for dynamic dependence analysis typically require access to the entire source code for a system, as well as the collection and storage of runtime data. These challenges are

Authors' addresses: Aashish Yadavally, University of Texas at Dallas, Dallas, USA, aashish.yadavally@utdallas.edu; Yi Li, University of Texas at Dallas, Dallas, USA, yi.li@utdallas.edu; Tien N. Nguyen, University of Texas at Dallas, Dallas, USA, tien.n.nguyen@utdallas.edu.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).
© 2024 Copyright held by the owner/author(s).
ACM 2994-970X/2024/7-ART13
https://doi.org/10.1145/3643739

exacerbated in security-sensitive and resource-constrained environments. Besides, certain development scenarios pose significant challenges. For instance, when dealing with programs reliant on intricate third-party libraries, enabling dynamic instrumentation or executing them for specific inputs becomes laborious and time-consuming. Next, consider facilitating the understanding of run-time behavior of code snippets from online forums such as StackOverflow. These are often incomplete and non-executable [43], thus hindering compiler-based approaches as well.

The motivation for our work lies in addressing these challenges and establishing an alternative approach to dynamic dependence analysis. Inspired by the success of prior research in learning static program dependencies [42], we propose a novel learning-based paradigm for predicting *dynamic dependencies*, namely, **predictive dependence analysis**. This paradigm marks a step forward in generating approximate dynamic dependencies at a large-scale by side-stepping the need for actual program execution. By utilizing complete code and leveraging prior execution knowledge, machine learning models can be trained for predictive dependence analysis.

In this paper, we seek to enhance the understanding of run-time behavior in programs. Thus, we focus on **predictive slicing**, a use-case of predictive dependence analysis designed to predict the dynamic slices of a program. Dynamic slices comprise subsets of program statements that impact a designated point of interest (referred to as the *slicing criterion*) for particular inputs. Software developers often employ such slicing techniques to concentrate their efforts on specific portions of the codebase, enabling them to understand, debug, and improve software systems efficiently.

Within the framework of predictive slicing, we introduce ND-Slicer, a predictive backward slicing approach that is applicable to both complete and partial programs. To enable predictive slicing [1] within ND-Slicer, it is imperative to capture the nuances of code execution. First, during the training phase, the availability of complete programs and inputs makes it possible to acquire the corresponding execution traces. This facilitates the generation of a dynamic slice for a given slicing criterion – serving as the training data. However, as is the case with incomplete code, the inference phase does not guarantee the availability of an execution trace. To address this limitation, we adopt a sequence-to-sequence approach, wherein, the encoder focuses on *learning the dynamic dependencies* in a program specific to the given input. Such dynamic program dependencies encompass dynamic data and control-dependence between variables at the slicing criterion and the remaining program statements at a fine-grained granularity. Such execution aware-knowledge forms the foundational connections required to construct a predictive slice.

Another significant challenge pertains to the specificity of the criterion within an execution. Unlike static slicing, where the slicing criterion is solely defined at the location of a variable in the program, dynamic slicing requires establishing the criterion at a precise occurrence of a statement in the execution trace. This distinction marks a fundamental difference between static and dynamic slicing techniques. For instance, consider a scenario in which a slicing criterion lies within the scope of a `while`-loop. In this case, the statement may be executed multiple times, causing the variables within it to be affected by different sets of program statements during each executing iteration of the loop. To effectively capture this intricacy, we specifically encode the occurrence of the iteration in ND-Slicer, helping the model establish a coherent connection between the instances of the criterion and the statements within the loop executed prior to the criterion, as well as to those statements executed before the beginning of the loop execution.

The next challenge revolves around constructing the predictive slice. Our task is to predict the sequence of statement/line occurrences that align with the actual execution order. In other words, the model's output is a sub-sequence of the execution trace in the reverse order, starting from the precise occurrence of the slicing criterion in the trace. Note that this sub-sequence may

[1] In the remainder of this paper, we refer to the dynamic backward slice predicted by ND-Slicer as the "predictive slice".

Listing 1. Python Code Example

```
1   a = [5, 2, 9, 11]
2   maxa = max(a)
3   mini = [[] for _ in range(len(a))]
4   for i, v in enumerate(a):
5     r = maxa - 2 * v
6     if (v - r) % 2 == 1:
7       s = v - r + 1
8     else:
9       s = v - r + 2
10    mini[i] = [v, math.sqrt(s)]
11  mini.sort(key = lambda x: x[1])
```

Listing 2. Execution Trace and Slices

```
1 Execution trace:
      1, 2, 3, 4, 5, 6, 9, 10, 4, 5, 6, 7,
      10-(crash)
2 Static backward slice from s at line 10:
      10, 9, 8, 7, 6, 5, 4, 3, 2, 1
3 Dynamic backward slice from s at line 10(2):
      10(2), 7(1), 6(2), 5(2), 4(2), 10(1), 9(1),
      6(1), 5(1), 4(1), 3(1), 2(1), 1(1)
```

Fig. 1. A motivating example

include statements/lines multiple times, all in accordance with the execution order. To address this challenge, we configure the decoder in our sequence-to-sequence framework to learn the conditional probability of generating an output sequence comprising discrete line numbers corresponding to the statements in the input sequence. As a result of the training process, this component learns the subtleties of execution order with respect to the executing iteration of the slicing criterion.

We conducted an extensive empirical evaluation of ND-Slicer on both executable and non-executable Python programs. The dynamic backward slices predicted by ND-Slicer for complete, executable Python programs exactly matches the actual dynamic slices in 81.3% of the cases, while obtaining a ROUGE-LCS score of 95.4%. Meanwhile, it correctly predicts the dynamic backward slices for non-executable, partial Python programs, i.e., those lacking referenced methods/classes in 54.6% of the cases. We explored the usefulness of ND-Slicer in the task of crash detection, observing that it predicts the crash faults with an accuracy of 63.9%. Finally, we conducted an in-depth analysis of our model, revealing its understanding of different statement types, execution iterations, as well as inter-procedural control and data flow. In summary, the key contributions of this paper are:

(1) ND-Slicer: a novel *predictive slicing* approach for both complete and partial programs that predicts the dynamic backward slice for a criterion without the need for actual execution. ND-Slicer is the first neural network-based approach to dynamic program slicing.
(2) We conduct a comprehensive evaluation on the performance of ND-Slicer, including the downstream task of locating crash faults, given an (in)complete program and its inputs.
(3) We present an in-depth analysis of ND-Slicer, demonstrating its understanding of different statement types, execution iterations, and inter-procedural data and control flows.

2 MOTIVATION

2.1 Example on Program Slicing

Consider the Python program in Fig. 1 inspired from an instance in CodeNetMut dataset [24]. This program encounters a crash at run-time (line 10), displaying the "`ValueError: math domain error`" message. The crash originates from the `math.sqrt(s)` method when the value of `s` becomes -4. In Listing 2, we illustrate the execution trace for this program. The bug occurs on line 6 in Listing 1, where the developers failed to ensure that the expression `v - r` has a sufficiently large value such that its assignment to the variable `s` on line 7, that further flows to line 10 remains non-negative.

To debug this defect, one could leverage program slicing. One approach is to analyze the program by using the static backward slice of the variable `s` on line 10 – which includes the lines 10, 9, 7, 6, 5, 4, 3, 2, and 1. We can see that the static backward slice considers all statements that can potentially affect the variable `s` across any program execution, rather than focusing on a specific execution.

This is not always helpful. For instance, when evaluating which statements impact the value of `s` on line 10, a static backward slicing technique must examine both branches of the `if`-statement on line 6, even if only one branch is executed in a specific scenario. In more complex programs with nested loops and conditional blocks, the number of such statements and corresponding paths in a static backward slice can explode exponentially, thus providing little assistance with debugging.

Alternatively, one can employ dynamic slicing for this purpose. The dynamic slice includes statements that affect the value of `s` on line 10 during the second iteration of the `for`-loop. Thereby, we establish the slicing criterion for the variable `s` as "10(2)", the corresponding dynamic backward slice, for which is shown on line 3 of Listing 2. Here, the number in parentheses indicates the occurrence of the statement in the execution trace. Despite being longer, the dynamic slice is more precise about the current execution, as it also considers the executing iterations of loops. This precision can assist developers in tracing back to critical statements in the dynamic backward slice, ultimately leading to locating the bug on line 6. One can potentially fix this bug by adding the verification: `v - r >= 0`. However, in certain development scenarios (as described in Section 1), executing the program might not be feasible, thus limiting such run-time behavior analysis.

2.2 Motivation and Key Ideas

To address these challenges, we develop ND-Slicer, a learning-based predictive slicing approach that predicts the statements belonging to a dynamic backward slice concerning a criterion, without executing the given program. In designing ND-Slicer, we incorporate the following key ideas:

First, during the training phase, we use complete code along with their dynamic backward slices extracted from the respective execution traces to construct our training data. Each training instance consists of three components: (a) the given program and its input, (b) the slicing criterion, and (c) the corresponding dynamic backward slice. In the running example, this translates to the following: (a) Listing 1, with the program input on line 1 (i.e., the list assignment to variable `a`); (b) line 10, which contains the variable `s`, in its second occurrence within the trace; and (c) line 3 in Listing 2.

2.2.1 [Key Idea 1] Dynamic Dependence Learning (DDL). Previous works [15, 42] have highlighted the effectiveness of sequence-based models in learning program dependencies. These approaches benefit from their ability to work with incomplete code. Thus, we chose to adopt a sequence-to-sequence framework for predictive slicing. Within this framework, the encoder aims to learn the nuances of source code from the training instances (as described earlier), capturing both dynamic data and control dependencies between the variable at the criterion and the statements that precede it in the execution trace. Such dependencies are input-dependent and tied to a specific execution. For example, let us consider the variable `s` for the second occurrence of line 10, i.e., 10(2) in the running example. There are two static `def-use` dependencies: one between the definition of `s` on line 7 and its use on line 10, and another between the definition of `s` on line 9 and its use on line 10. The goal of our DDL component, in this case, is to learn that the dynamic dependency for this specific execution arises from the statement 7(1), rather than from the statement 9(1), to the variable `s` at 10(2). In a similar vein, the DDL component must also learn the control dependence between 6(2) and 7(1) along with the connections based on the variables `v` and `r`. These dependencies help connect the statements towards predicting the backward slice.

Second, a criterion must be defined at a precise point in the execution history. For example, to debug the crash in the running example, one can start from the criterion corresponding to the variable `s` at line 10 in the second iteration of the `for`-loop. For this purpose, ND-Slicer needs to understand and distinguish between the execution of line 10 in different iterations of the loop.

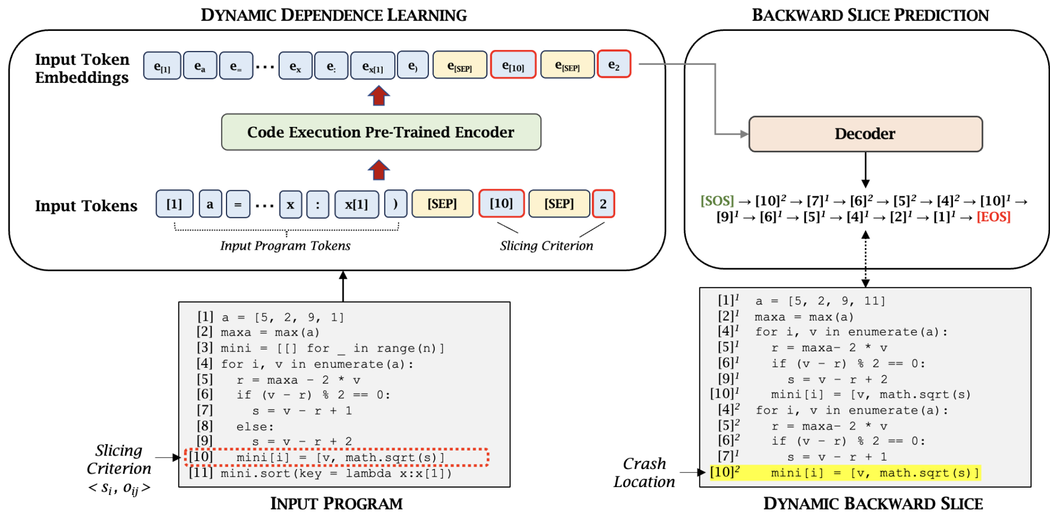

Fig. 2. A general overview of ND-Slicer approach for predictive dynamic slicing

2.2.2 [Key Idea 2] Encoding Execution Iterations. To enable ND-Slicer to understand the executing iteration of the slicing criterion, we explicitly encode the specific occurrence (e.g., first or second occurrence, etc.) of the criterion in the input to our encoder. Such knowledge helps the DDL component in learning the dynamic data/control dependencies between the criterion's occurrence and the statements preceding it in the execution history. Moreover, such a strategy also enhances the model's understanding of execution flow, helping it distinguish between the current and all prior occurrences of the criterion. In the case of the running example, combining Key Ideas 1 and 2 teaches the model to learn the distinct data/control dependencies incorporated in the second and first occurrences of the slicing criterion in the execution history, i.e., the sub-sequences 10(2) $\longrightarrow$ 7(1) $\longrightarrow$ 6(2) and 10(1) $\longrightarrow$ 9(1) $\longrightarrow$ 6(1), respectively. We standardize such an encoding across the program by assigning "1" as the execution iteration for all non-loop slicing criteria.

Third, the *predictive backward slice must be in accordance with the execution order* of the relevant statements in the execution trace, while also taking into consideration their occurrence number within that trace – as a statement can be executed multiple times.

2.2.3 [Key Idea 3] Attention-Based Decoders for Slice Generation. We leverage the Transformer [38] decoder (see Section 5 for details on all variants) within the sequence-to-sequence framework to model the generation of the sequence of statements at a specific occurrence and construct the predictive backward slice. We incorporate line numbers into the input for the DDL component, enabling it to understand statement-level dynamic data and control dependencies. Leveraging this knowledge, the Transformer decoder learns the conditional probability of generating a sequence containing these discrete line numbers, representing the predictive slice. In the running example, it learns to correctly predict the output sequence 10(2) $\longrightarrow$ 7(1) $\longrightarrow$ 6(2) $\longrightarrow$ 5(2)... (line 3 in Listing 2).

3 ND-SLICER: NEURAL NETWORK-BASED PREDICTIVE BACKWARD SLICING

3.1 Framework Overview

Given an (in)complete program, its inputs, and a specific statement with its occurrence in the execution trace as the slicing criterion, ND-Slicer predicts the dynamic backward slice for the

criterion. In Fig. 2, we illustrate the model architecture for ND-Slicer. Broadly, it adopts the sequence-to-sequence framework for predictive slicing, with the following essential components:

3.1.1 Dynamic Dependence Learning. In recent times, specialized source code models [15, 42] have highlighted the effectiveness of learning-based approaches in understanding data and control dependencies. However, their learning objectives do not focus on run-time behavior of code, limiting their suitability for predictive slicing. To address this challenge, we chose to leverage CodeExecutor [24], a model pre-trained on code execution, as the encoder in our framework. This code execution pre-training equips ND-Slicer with the capability to *learn dynamic program dependencies*. Furthermore, treating source code as a sequence of tokens empowers such pre-trained models (PTMs) to operate efficiently for both complete and partial code, thereby overcoming the limitations of dynamic execution, which is typically applicable only to complete code.

Let us consider a program $P = \langle s_1, s_2, \cdots, s_N \rangle$, where s_i represents a statement in the program. Note that the inputs to the program, I_P, are included as assignment statements at the beginning of the program. We represent each statement s_i as a sequence of code tokens $\langle [i], t_1^{(i)}, t_2^{(i)}, \cdots, t_{M_i}^{(i)} \rangle$, where $[i]$ represents the special *line identifier* token included at the beginning of every line. For a given slicing criterion $\langle s_c, o \rangle$, that denotes the task of predicting the slice for the o-th occurrence of the statement s_c in its execution history, the inputs to the encoder in ND-Slicer are represented as:

$$\{[1], t_1^{(1)}, t_2^{(1)}, \cdots, t_{M_1}^{(1)}, \cdots, [\text{N}], t_1^{(N)}, t_2^{(N)}, \cdots, t_{M_N}^{(N)}, [\textbf{SEP}], [c], [\textbf{SEP}], o\}$$

For each token t in this input representation, the DDL module generates a contextually enriched token representation $e_t \in \mathbb{R}^d$ (where d is the model dimension) with the following goals: (1) e_t is syntax, semantics, and execution-aware; (2) it learns the execution-flow based on the model input tokens $t_x \in I_P$; (3) it learns the dynamic data and control dependencies between $t_j \in P$ and $t^{(c)}$ based on the occurrence of the slicing criterion denoted by $[c]$ and o, respectively.

3.1.2 Backward Slice Prediction. The primary challenge in predicting backward slices lies in ensuring that the sequence of statement/line occurrence aligns with the actual execution history, where the output sequence captures all dynamic data and control dependencies on the slicing criterion. Given the proven effectiveness of the *attention mechanism* in propagating information of the input sequence to the decoder within the sequence-to-sequence framework, we chose to adopt the Transformer decoder in ND-Slicer (see Section 5 for all variants).

The encoder in the DDL module maps the model input to a sequence of continuous representations $\mathbf{x} = \{e_{[1]}, e_{t_1^{(1)}}, \cdots, e_{[SEP]}, e_{[c]}, e_{[SEP]}, e_o\}$. Given $\mathbf{x}$, the decoder generates an output sequence $\mathbf{y} = \{y_1, y_2, \cdots, y_k\}$ of symbols one element at a time. Here, each y_k corresponds to the special line identifier token $[i]$ in the input representation. This process is auto-regressive, where the previously generated symbols are consumed as the additional input for generating the next.

3.2 Training Process and Inference

During the training phase, ND-Slicer aims to learn the conditional probability $P(\mathbf{y}|\mathbf{x})$ of generating the sequence of discrete line numbers y_i corresponding to the program statements in P that represent the dynamic backward slice for a specific slicing criterion. Mathematically, this is equivalent to:

$$P(\mathbf{y}|\mathbf{x}) = P(y_1, y_2, ..., y_k|\mathbf{x}) = \prod_{i=1}^{k} p(y_i|y_{<i}, \mathbf{x}) \quad (1)$$

For each instance in the training data, intuitively, we maximize the probability that the model assigns to the correct token at each step of the generation process. Let's assume p_j^* is the one-hot encoded representation of the correct token at position j in the output sequence, while p_j is the

probability distribution predicted by the model for the token at position j over the vocabulary. Making use of the standard *cross-entropy* loss for training the model, formally, the loss associated with a training instance at each step corresponds to:

$$\mathcal{L}_{step}(\theta) = -p^* log(p) = -\sum_{j=1}^{|V|} p_j^* log(p_j) \tag{2}$$

At each step, we maximize the probability the model assigns to the correct token. The total loss for the entire training instance is computed as follows:

$$\mathcal{L}_{instance}(\theta) = -\frac{1}{k}\sum_{i=1}^{k} \log(p(y_i|y_{<i}, x)) \tag{3}$$

During inference, we start with the [**SOS**] token and follow the simple transduction process by decoding from left-to-right, picking the "best" token, i.e., the token with the highest probability at each step, given by $y' = argmax_y \prod_{i=1}^{k} p(y_i|y_{<i}), \mathbf{x})$. We continue this process until the [**EOS**] token is reached. This process can easily be extended with the beam search decoding strategy to extract a set of approximate, plausible predictive slices for each program.

4 EMPIRICAL EVALUATION

We conducted several experiments, seeking to answer the following research questions:

(I) Intrinsic Evaluation

RQ$_1$. Effectiveness on Executable Python Code: *How accurate is ND-Slicer in predicting dynamic slices for a given slicing criterion in complete, executable Python programs?*

RQ$_2$. Effectiveness on Non-Executable Python Code: *How accurate is ND-Slicer in predicting dynamic slices for a given slicing criterion in incomplete and non-executable Python programs?*

(II) Extrinsic Evaluation

RQ$_3$. Crash Detection: *How useful is ND-Slicer in locating crash faults in Python program?*

(III) In-Depth Study of Model Performance

RQ$_4$. Statement Types: *How well does ND-Slicer perform on different types of statements?*

RQ$_5$. Execution Iteration in Loops: *How well does ND-Slicer perform when slicing statements in the scope of a loop for different execution iterations?*

RQ$_6$. Probing Inter-Procedural Slicing: *How well does ND-Slicer perform when predicting the inter-procedural slices, and when the slices cross the boundaries of procedure calls?*

5 PREDICTIVE SLICING OF EXECUTABLE PYTHON PROGRAMS (RQ$_1$)

5.1 Data Collection

5.1.1 Input Programs. To evaluate the effectiveness of our approach on Python programs, we utilized the CodeNetMut dataset [24]. This dataset comprises executable Python programs, which are essentially mutation variations of the Project CodeNet dataset [32]. Notably, these programs do not rely on specific external resources such as file contents, external modules, or third-party packages. Furthermore, each program instance includes the input values at the beginning of the program in the form of assignment statements, assigning the values to input variables.

5.1.2 Slicing Criteria. For each program in the dataset, execution trace is provided which consists of: the order in which the computer executes statements; and how the states of the variables change when jumping from one statement to another. The execution history can have statements repeating multiple times. Thus, we select all statements with all of their occurrences as the slicing criteria.

5.1.3 Ground-Truth Dynamic Backward Slices. Agrawal and Horgan [4] present several approaches to computing dynamic slices, of which, we adopt their Algorithm 2. For a given program, first, we build the static program dependence graph (PDG) by leveraging the *python-graphs* [2] tool. Initially, none of the edges in the PDG are *marked*. Next, starting from the occurrence of the slicing criterion in the statement execution history, we mark the edges of the PDG based on the dependencies that arise during program execution. Finally, we traverse the graph only along the marked edges to identify the set of all statements that make up the resulting dynamic slice.

In the CodeNetMut dataset, Liu *et al.* [24] make 19,541 Python programs and their execution traces available. Among these, we skipped the ones for which *python-graphs* fails to build the PDGs. Based on the data construction steps detailed above, we combined the execution traces for these programs with their respective static PDGs to compute the ground-truth dynamic slices, and extract a total of 168,289 $\langle P, [\mathrm{c}], o, S\rangle$ tuples for predictive slicing. Here, each denotes the input program, the statement in the slicing criterion, the occurrence of the slicing criterion in the execution trace, and the dynamic backward slice, respectively. Next, we split them at the problem-level to avoid data leakage, in the 80%-10%-10% ratio, dedicating a total of 104,464 instances for training, 13,770 instances for validation, and 12,203 instances for testing. Overall, our dataset has programs with a maximum of 95 statements; and the lengths of the dynamic slices range from 2 to 10.

5.2 Experiment Methodology

5.2.1 Baselines. First, we compared our approach with a naive baseline (B_0) to establish a reference for the improvements of predictive slicing over static analysis. In B_0, we computed execution traces by deterministically selecting the following statement as the next executed statement, employing random selection in the case of `if`-conditions, and iterating only once over loops. We then combined these traces with the Dynamic Slicing Algorithm outlined in Section 5.1.3 to extract the dynamic slices for corresponding slicing criteria.

Next, in place of the arbitrary traces (as in B_0), we used *pre-trained CodeExecutor* [24] to predict the execution traces for all programs in our evaluation dataset. To establish this baseline (B_1), we then combined the predicted traces with the Dynamic Slicing Algorithm to compute the dynamic slices. Note that B_1 required no additional training. For the next baseline (B_2), we *fine-tuned CodeExecutor* for the downstream task of predictive slicing, in the encoder-decoder mode. In this case, the dynamic backward slices are directly used as targets to train the model for this task.

In ND-Slicer, due to realizing the predictive slicing task as a sequence-to-sequence framework, it gives us the flexibility to plug in different encoders and decoders. We conducted experiments with two encoders: GraphCodeBERT [15] and CodeExecutor [24]. The rationale behind choosing GraphCodeBERT is its data flow-specific pre-training objective, which encodes the relation of where the value of a variable comes – making it suitable for program slicing. In essence, the sequence-to-sequence framework leveraging GraphCodeBERT as the encoder learns to pick the specific execution path corresponding to the inputs from among all possible program paths. In the case of CodeExecutor, the knowledge from code execution-specific pre-training makes it suitable for learning the dynamic data and control dependencies for different program inputs.

The backbone of both GraphCodeBERT and CodeExecutor encoders is the standard RoBERTa-base [26] architecture which has 12 Transformer-encoder layers, each possessing 12 attention heads. We used the byte-pair encoding (BPE) scheme in the pre-trained RobertaTokenizer for splitting the given source code into sub-tokens. In addition, we added up to 200 *line identifiers* (i.e., [i] tokens) as special tokens to the tokenizer that are independent of the specific input.

In our sequence-to-sequence framework, we employed two decoders: the standard Transformer decoder [38] and the Pointer-Transformer [31]. The Transformer decoder comprises a stack of

Table 1. Model performance on executable Python programs (RQ1)

Approach		Evaluation Metrics (in %)			
		R-L$_P$	*R-L*$_R$	*R-L*$_F$	*EM*
Naive	Arbitrary Execution Trace + Dyn. Slicing Algorithm (B_0)	33.8	36.3	34.3	29.7
Pre-Trained	CodeExecutor + Dyn. Slicing Algorithm (B_1)	52.9	53.6	53.1	49.2
Fine-Tuned	CodeExecutor (B_2)	94.5	89.9	90.9	58.8
	GraphCodeBERT				
	+ Pointer-Transformer (B_3)	85.9	90.8	86.6	62.1
	+ Transformer (B_4)	93.5	94.9	93.2	75.8
	CodeExecutor				
	+ Pointer-Transformer (B_5)	83.0	92.9	85.5	62.4
	+ Transformer (B_6)	95.4	96.4	**95.4**	**81.3**

6 layers, while the Pointer-Transformer is a hybrid of the Transformer decoder and the Pointer-Generator Network [34]. The general idea for the Pointer-Transformer is that it copies from the input sequence to generate the output sequence. Pointer-Transformer achieves the state-of-the-art performance in solving combinatorial/ordering problems over a finite set of points (e.g., Traveling Salesman Problem) – which aligns with the task of detecting the sequence of program statements belonging to the dynamic slice. This makes it suitable for our task, as the dynamic backward slice predicted by ND-Slicer is a sequence of *line identifiers* selected from the input sequence. To facilitate the pointer mechanism, we expand the vocabulary with special position markers pointing to input positions. Finally, it predicts a sequence of these position markers, which are subsequently mapped back to the input to generate the predictive slice.

We used all combinations of the aforementioned encoders and decoders as baselines: GraphCodeBERT + Pointer-Transformer (B_3), GraphCodeBERT + Transformer (B_4), CodeExecutor + Pointer-Transformer (B_5), and CodeExecutor + Transformer (B_6). We conducted all our experiments on an NVIDIA RTX A6000 GPU, training all variants for 10 epochs. We initialized them with learning rates of 1e-4 and 5e-4, and report the best-performing models.

5.2.2 Evaluation Metrics. We use the following metrics to assess our model performance: (a) *Exact Match Accuracy*, a stricter version of accuracy which ensures that the predicted dynamic backward slice by a model exactly matches the ground-truth dynamic slice; (b) *ROUGE-L*, which compares the predicted and ground-truth backward slices based on the longest common sub-sequence (LCS). The rationale behind adopting *ROUGE-L* scores is that it helps assess *the order* of the output sequence. Mathematically, given the ground-truth backward slice S and the predicted slice $\hat{S}$ of lengths L_S and $L_{\hat{S}}$, respectively: *ROUGE-L Precision* ($R\text{-}L_P$) = $\frac{LCS(S,\hat{S})}{L_S}$, *ROUGE-L Recall* ($R\text{-}L_R$) = $\frac{LCS(S,\hat{S})}{L_{\hat{S}}}$, and *ROUGE-L F1-Score* ($R\text{-}L_F$) = $\frac{2.R\text{-}L_P.R\text{-}L_R}{R\text{-}L_P+R\text{-}L_R}$.

5.3 Empirical Results

In Table 1, we report the performance of ND-Slicer in the dynamic backward slicing task on complete, executable Python programs. Our approach, combining CodeExecutor with Transformer decoder within the sequence-to-sequence framework (B_6), produces the most competitive results, predicting the dynamic backward slices with an *Exact Match Accuracy* of 81.3% and *ROUGE-L F1-score* of 95.4%. Compared to the naive baseline (B_0), we observe an improvement in both metrics

by 173.7% and 178.1%, indicating the non-trivial nature of this task. In addition, the gains range from 7.3% to 65.4% and 2.3% to 79.7%, respectively, against all learning-based baselines.

When we employed the pre-trained CodeExecutor (B_1) to predict execution traces and subsequently derive dynamic backward slices, we observed the least favorable results. A potential explanation for this lies in the inherent complexity of execution trace prediction, where the model in its pre-trained state, also attempts to predict program states containing ⟨*variable*, *value*⟩ pairs for each statement, thus introducing a cascading error that affects its performance. Conversely, predicting the dynamic backward slice directly seems to be the better strategy for this task, especially since all baselines B_2–B_6 modeled in this manner outperform B_1.

Next, we assessed the fine-tuned version CodeExecutor (B_2, row 2 in Table 1) for this task. This baseline achieved an *Exact Match Accuracy* of 58.8% and a *ROUGE-L F1-score* of 90.9% in predicting dynamic backward slices. While this represents a notable improvement over using the pre-trained CodeExecutor directly (B_1), with gains of 19.6% and 71.2% for both metrics, it still falls short of the best-performing baseline (B_6) by 38.3% and 5.0%, respectively. Further examination revealed that the fine-tuned CodeExecutor continued to predict tokens corresponding to program states despite being trained extensively on predictive slice data. Such tokens were subsequently discarded via post-processing. This observation, however, suggests potential limitations in the applicability of CodeExecutor directly in ⟨*encoder*, *decoder*⟩-style downstream tasks for source code.

GraphCodeBERT leverages code structure and data-flow knowledge to guide its pre-training learning objectives, resulting in an understanding of static dependencies within code. This is evident when combining the model with Pointer-Transformer (B_3) and Transformer (B_4) decoders, where they outperform baseline B_2 by 5.6% and 28.9% in *Exact Match Accuracy*, respectively. Notably, the Pointer Transformer variant (B_3), achieves a lower *ROUGE-L F1-score* than B_2, with the metric declining by 5.0%. This observation can be attributed to programs with loops, particularly those where the slicing criterion corresponds to a higher execution iteration. In these cases, Pointer Transformer struggled to effectively leverage the data-flow knowledge to select the right slicing path, given the inherent complexity. On the other hand, the Transformer variant (B_4) exhibited better performance in handling such scenarios, as indicated by its improvement over B_2 by 2.5% in *ROUGE-L F1-score*. In direct comparison with B_6, B_3 and B_4 exhibit lower performance in exactly matching the ground-truth backward slices, underperforming by 31.0% and 7.3%, respectively.

CodeExecutor leverages code execution during pre-training. Thus, in baselines B_5 and B_6, we employed CodeExecutor as the encoder, pairing it with Pointer Transformer and Transformer as the decoders in ND-Slicer, respectively. B_5 demonstrates performance similar to B_3, underscoring the constraints of Pointer Transformer in the context of predictive slicing. In contrast, B_6 is the top-performing baseline, improving over the next best model, GraphCodeBERT + Transformer by 7.3% in *Exact Match Accuracy*, and *ROUGE-L F1-score* in 2.4%, respectively. As a result, we select B_6 as the primary model in ND-Slicer for the remainder of this paper. We also explore its performance in various experiment settings, addressed via RQ_2 – RQ_6.

6 PREDICTIVE SLICING OF NON-EXECUTABLE PYTHON PROGRAMS (RQ_2)

In the case of partial programs, i.e., when the code is non-executable due to missing variable declarations, missing referenced methods/classes, or missing import statements for external libraries, etc., it is not possible to compute the dynamic slices from their execution trace and corresponding program dependence graphs (as described in Section 5.1). In contrast, ND-Slicer is not limited by the completeness of the program. We set up this experiment to exhibit our tool's ability in predicting dynamic program slices for non-executable Python programs.

Table 2. Model performance on non-executable Python programs (RQ2)

Approach	Statement Type	Evaluation Metrics (in %)			
		R-L_P	*R-L_R*	*R-L_F*	*EM*
Naive (B_0)	*Branchless*	44.7	48.9	45.8	39.1
	Conditions, Loops	29.5	31.7	29.9	26.2
	Overall	38.1	41.4	38.9	33.5
ND-Slicer (B_6)	*Branchless*	81.8	88.0	81.9	59.5
	Conditions, Loops	81.7	85.2	81.3	55.0
	Overall	81.8	86.8	81.6	**57.5**

6.1 Experiment Methodology

Extracting ground-truth dynamic backward slices for partial programs is not possible without manual intervention. As a result, we picked Python programs from the test set in Section 5 and omitted the `import`-statements for modules whose utilities are being used in the code – thus imitating a non-executable program. In this process, we excluded any instances which: (a) do not have `import`-statements; (b) length of resulting dynamic backward slices is less than 2. Overall, we obtained 1,849 instances, of which 1,044 are branch-free, and the rest contain `if-else` blocks or loops.

We evaluate the performance of the best-performing variant model of ND-Slicer in RQ_1, i.e., CodeExecutor + Transformer, on this non-executable Python program dataset. To assess its performance in this experiment, we adopt the same evaluation metrics as in Section 5.3.

6.2 Empirical Results

In Table 2, we report the performance of ND-Slicer on non-executable Python programs, further categorizing them based on the presence or absence of branches. We can see that ND-Slicer achieves an *Exact Match Accuracy* of 57.5% and a *ROUGE-L F1-score* of 81.6% in matching the ground-truth dynamic backward slices, improving over the naive baseline by 71.6% and 109.8%, respectively. Notably, this accuracy decreases by 4.6% in cases involving code with branches but increases by 3.37% in branch-free code, both of which are significantly higher in the case of B_0.

Upon further examination, we identified that mispredictions in B_6 predominantly arise from the statements referencing methods belonging to the external libraries. In cases where the model did not trace back along such statements, irrespective of branching, ND-Slicer predicted the dynamic backward slices correctly. This observation highlights that CodeExecutor does not have an understanding of the libraries themselves, as it was not trained with the source code for these libraries. This underscores potential gaps in the pre-training strategy employed by CodeExecutor for acquiring execution knowledge, suggesting enhancements in future on pre-training with libraries.

7 PREDICTIVE PROGRAM SLICES FOR CRASH DETECTION (RQ_3)

As illustrated in Section 2, dynamic backward slicing plays a crucial role in predicting program crash faults by pinpointing the specific sequence of statements responsible for the crash. In practice, manually isolating the code segment that triggers a crash is challenging as it requires a deep understanding of the code at hand. In this experiment, we seek to evaluate the effectiveness of ND-Slicer in producing the backward slice to locate crash faults within Python programs.

7.1 Experiment Methodology

We enabled this experiment by injecting synthetic `ZeroDivisionError` faults in the test Python programs in Section 5.1. These often occur when a division or modulo operation is attempted with a

```python
1   s = "level"
2   n = len(s)
3   ans = "No"
4   cnt = 0
5   if s[::1] == s[::-1]:
6      injected_var = s / cnt
7      cnt += 1
8   if s[:int((n - 1) / 2) : 1] == s[int((n - 1) / 2) - 1 :: -1]:
9       cnt += 1
10  if s[int((n + 1) / 2) :: 1] == s[ : int((n - 1) / 2) : -1]:
11     cnt += 1
12  if cnt == 3:
13     ans = "Yes"
14  print(ans)
```

Fig. 3. A Python Code Example in CodeNetMut Dataset with an Injected Crash Fault

denominator or divisor of zero. To facilitate such a crash injection, first, we identified the instances in which the value of any of the variables in the program becomes zero. Let the line number for this statement be m and the variable for which the value becomes zero be x. Next, we identified the statement where the value of the variable changes from zero to a different one. Let the line number for this statement be n ($n > m$). We then injected a crash fault of the form: `injected_var = y / x`, where y is one of the other variables in the program states at line $n - 1$. We then applied the Dynamic Slicing Algorithm described in Section 5.1.3 considering the location of the crash fault (i.e., line $n - 1$) as the slicing criterion to compute the ground-truth dynamic backward slices.

Let us consider the Python program in Fig. 3 to elucidate the procedure for crash injection. Here, the value of `cnt` is initialized as `0` on line 4. On line 7, this value changes from `0` to `1`. We pick line 6 as the location to inject the crash fault. Considering the first occurrence of line 8 as the slicing criterion, i.e., 8(1), we construct its ground-truth dynamic backward slice: {8, 5, 4, 2, 1}. This slice includes the additional statements not dependent on `cnt`, i.e., 2 and 1 due to the variable `s` on line 6.

Overall, we obtained 236 such crash-injected Python programs. Next, we leveraged the best-performing model in Section 5, i.e., fine-tuned CodeExecutor+Transformer, to predict the dynamic backward slices for the crash locations. Note that not all statements in the dynamic backward slice could be crash-inducing, and only those that share dynamic data dependencies with the variable x on line m are potential crash locations (e.g., `cnt` on line 4 in Fig. 3). Thus, to assess ND-Slicer in crash detection, we identified the number of instances in which the predicted slice contains at least one of the statements having dynamic data dependencies with the crash location on variable x.

7.2 Empirical Results

In this experiment, we evaluated the performance of ND-Slicer in identifying crash locations and predicting dynamic backward slices in real-world faulty programs. Our results show that ND-Slicer is able to correctly identify crash locations in 63.9% of cases, and predict the dynamic backward slice exactly the same in 47.5% of the cases. This is a promising result, as it demonstrates the potential of ND-Slicer to be a useful tool for debugging real-world programs. We further analyzed the cases where ND-Slicer was not able to correctly identify crash locations or dynamic backward slices. We found that the most common source of error was choosing the wrong path in an `if-else` statement, or while predicting the dynamic backward slice for a higher executing iteration in a loop. While these errors are still limiting, we believe they can be addressed in future work by developing strategies to improve ND-Slicer's understanding of execution flow in a loop.

Table 3. ND-Slicer's performance on Python programs with different statement types (RQ4)

Approach	Statement Types	Evaluation Metrics (in %)			
		R-L$_P$	*R-L$_R$*	*R-L$_F$*	*EM*
Naive (B_0)	*Branchless*	42.7	45.6	43.3	38.1
	Conditional	43.2	44.8	43.5	40.4
	Loops	17.6	19.7	18.1	14.3
ND-Slicer (B_6)	*Branchless*	97.0	97.4	96.8	86.9
	Conditional	96.1	95.1	95.0	78.7
	Loops	93.1	95.2	93.4	72.5

8 IN-DEPTH STUDY OF MODEL PERFORMANCE (RQ_4 – RQ_6)

In this section, we conduct an in-depth qualitative analysis of ND-Slicer by examining different aspects of model performance, and further validate our hypotheses and findings with case studies.

8.1 Statement Types (RQ_4)

Analyzing various control structures provides fine-grained insights into program comprehension. For instance, slicing programs without branches directly attests to ND-Slicer's understanding of dynamic data dependencies. When tracing through `if-else` blocks, we can assess whether ND-Slicer understands the variables involved in conditions and how they influence branch selection or operations within branches. Likewise, with a loop, we can determine ND-Slicer's ability to understand data/iteration dependencies within the loop, as well as its exit conditions.

8.1.1 Experiment Methodology. To facilitate this experiment, we picked Python programs in the test set in Section 5.1, stratifying them based on whether the program is: (a) *branch-free*, i.e., has no conditional statements or loops; (b) contains an `if-else` block; (c) contains a `for/while` loop. In the case of conditional statements, we only considered instances with `if-else` blocks that are independent of loops. Moreover, in both (b) and (c), we ensured that the slicing criterion is within the scope of these control structures. Here, we assess the performance of the best-performing CodeExecutor+Transformer (from Section 5.3), while also adopting the same evaluation metrics.

8.1.2 Empirical Results. In Table 3, we report the performance of ND-Slicer on Python programs with different statement types. We can see that in the case of branch-free code, the predictive slices from our tool considering different slicing criteria within the program, exactly matches the ground truth in 86.9% of the instances, yielding a *ROUGE-L F1-score* of 96.8%. Compared to the overall performance of ND-Slicer (Table 1), this is an improvement by 6.9%. Such a high performance for these instances reiterates ND-Slicer's capabilities in understanding *dynamic data dependencies.*

Next, let us examine Python programs with conditional statements. In this context, ND-Slicer demonstrates a 78.7% *Exact Match Accuracy* in predicting the dynamic backward slice and achieves a *ROUGE-L F1-score* of 96.1%. When compared to ND-Slicer's overall performance, this is a decrease by 3.3%. This is reasonable because the code with conditional statements is more complex than the one without them. Overall, a high performance for instances with conditional statements underscores ND-Slicer's proficiency in understanding dynamic control dependencies.

In the case of Python programs with loops, we observe that ND-Slicer exactly predicts the dynamic backward slices 72.5% of the times, with a *ROUGE-L F1-score* of 96.8%. A more detailed analysis revealed that in code with no conditional statements within the scope of the loops, the accuracy improves to 76.4%. Conversely, in cases when programs have `if-else` blocks within the

Table 4. ND-Slicer's performance by execution iteration for loop examples: with `if-else` blocks, without `if-else` blocks, all examples. Here, *EM* denotes Exact Match, and $R\text{-}L_x$ denotes ROUGE-LCS scores.

Split (→)	w/ `if-else` (D_b)				w/o `if-else` ($D_{b'}$)				Overall (D_{loop})			
Iteration (↓)	$R\text{-}L_P$	$R\text{-}L_R$	$R\text{-}L_F$	*EM*	$R\text{-}L_P$	$R\text{-}L_R$	$R\text{-}L_F$	*EM*	$R\text{-}L_P$	$R\text{-}L_R$	$R\text{-}L_F$	*EM*
1	89.4	92.1	89.1	72.0	96.9	97.5	96.8	90.8	93.2	94.9	93.1	**81.7**
2	88.8	91.4	88.9	61.7	96.9	97.1	96.7	85.0	93.8	94.9	93.7	76.1
3	86.9	90.7	87.5	51.5	96.5	97.2	96.4	81.3	93.2	95.0	93.4	71.2
4	85.4	88.4	85.8	53.9	95.3	96.6	95.4	75.9	91.9	93.8	92.1	68.4
5	87.5	90.7	88.0	57.1	94.8	95.8	94.7	72.8	93.5	94.9	93.5	70.0
6 – 10	92.5	95.1	93.3	60.3	90.9	95.7	92.4	58.0	91.1	95.6	92.5	58.2
All	–											**63.6**

scope of the loops, the accuracy drops to 63.3%. Such programs are complex. For instance, the backward slice for a statement can flow from the `if`-block in one execution iteration, and the `else`-block in the other (Section 8.4). Therefore, the drop in performance by 5.4% relative to the overall performance of ND-Slicer can be attributed to such complex programs.

Overall, we can see that ND-Slicer captures intricate *dynamic data, execution iteration, and control dependencies*, which corroborates our key ideas to produce better dynamic program slices.

8.2 Execution Iteration in Loops (RQ_5)

The iterative and potentially complex nature of loops makes dynamic slicing of loops extremely challenging. It requires an intricate understanding of the control-flow of the program. For instance, loops can carry dependencies across execution iterations, tracking which is crucial for accurately predicting the dynamic slice. ND-Slicer encodes the execution iteration of the loop as a part of its input representation. Accordingly, we designed this experiment to assess ND-Slicer's performance for slicing program statements within the scope of a loop, across multiple execution iterations.

8.2.1 Experiment Methodology. To set up this experiment, we collected different instances in our test set that have a `for`-loop. Let us call this D_{loop}. Next, we stratified D_{loop} based on whether they contain branches in the form of `if-else` blocks or not. Let the resulting subsets be D_b and $D_{b'}$, respectively. We then grouped the instances in D_b, D_b', and D_{loop} based on their executing iterations (or occurrences) in the loop. Finally, we measured the performance of the best-performing CodeExecutor+Transformer model in ND-Slicer on all such groups, recording the same evaluation metrics as in Section 5.3. Note that this experiment setting is across all programs and criteria, that is, we group by the occurrences, irrespective of the program and slicing criteria.

Next, we also stratified D_{loop} based on the slicing criteria $[c]$ for all programs P, retrieving data instances of the form $\langle P, [s], \{o_1, o_2, ..., o_N\}\rangle$, where o_i refers to the execution iteration of the criterion $[c]$. Here, we record the *percentage of instances* in which the predictive slices from ND-Slicer exactly matches the ground-truth dynamic backward slice for all occurrences o_i. This experiment setting, in contrast, is within individual program and slicing criteria. Overall, there are 4,126 loop examples in our dataset (i.e., $|D_{loop}|$), of which 1,313 have `if-else` blocks within them. There are a total of 1,090 instances in the within program and criteria experiment setting.

8.2.2 Empirical Results. In Table 4, we present the results for these experiment settings. In general, we can see a consistent trend of decreasing *Exact Match Accuracy* as the execution iteration o_i increases, highlighting ND-Slicer's evolving challenge in maintaining accuracy as the iterations progress: a decrease from 81.7% to 70.0% as o_i goes from 1 → 5, and a further decline to 58.2% when

$o_i \in [6, 10]$. Furthermore, we see a significant disparity in ND-Slicer's performance when dealing with loop examples with and without `if-else` blocks. This suggests that `if-else` blocks introduce complexities that occasionally lead to incorrect branch decisions while predicting the dynamic backward slice. This effect is pronounced for lower values of o_i. Surprisingly, for a higher value of o_i, say when $o_i \in [6, 10]$, we can see that the accuracy of ND-Slicer on D_b is slightly higher than that of $D_{b'}$. This hints that the errors in $D_{b'}$ may be arising from challenges in understanding intricate loop-exit conditions, which tends to get more complex for higher execution iterations.

Interestingly, despite these challenges, the *ROUGE-L* scores for all data subsets D_b, $D_{b'}$, and D_{loop}, remain consistently high across all execution iterations. This observation indicates that *the model predicts the majority of the dynamic backward slice accurately*, with only a small portion of the slice being predicted incorrectly – which can be tied to the factors mentioned above. Furthermore, we can see that ND-Slicer accurately predicts slices across all occurrences of a particular slicing criterion in a program in **63.6%** of the cases. Overall, these findings collectively shed light on the nuanced interplay between execution iterations, `if-else` blocks, and ND-Slicer's predictive capabilities, offering valuable directions for future research and optimization.

8.3 Probing Inter-Procedural Slicing (RQ_6)

In this experiment, we delve into the performance of ND-Slicer in inter-procedural program slicing. Inter-procedural dynamic analysis, even by traditional techniques, poses a substantial challenge as it requires the comprehension on the control flows spanning multiple methods through the complex setting of dynamic instrumentations. In particular, in the scenarios involving incomplete code, the developer will need to ascertain the availability of the referenced methods/classes in both caller and callee functions to enable a seamless execution. Through this experiment, we aim to provide insights on ND-Slicer's proficiency in understanding intricate caller-callee relations, within the context of inter-procedural dynamic slicing.

8.3.1 Experiment Methodology. To enable this experiment, we curated the Python programs in the original CodeNetMut dataset featuring inter-procedure calls. Note that none of these instances were included in our dataset for intrinsic evaluation (in Section 5.1), as the PDG building tool, *python-graphs*, failed for these cases. As a result, ND-Slicer was exclusively trained in intra-procedural slicing scenarios – making this investigation into inter-procedural slicing particularly intriguing.

In a program P containing methods M_1 and M_2, let us assume M_1 calls M_2 on line x^{M_1} in M_1. Given that the static PDG-building tool fails for P overall, we build the static PDG for M_1 independently and extract the lines $y_i^{M_1}$ in M_1 having dynamic data dependencies with x^{M_1} based on the corresponding statement execution history. Finally, we use ND-Slicer to predict the dynamic backward slices considering the set of statements $\{x^{M_1}\} \cup \{y_i^{M_1}\}$ as the slicing criteria. Let S be the predictive slice so constructed. Overall, we retrieved 579 such instances with inter-procedural calls.

We define two metrics to quantify the model performance in this experiment: soft-linkage accuracy (SLA), and hard-linkage accuracy (HLA). Soft-linkage accuracy is the ratio of the total number of cases in which the predictive slice contains at least one line from the callee method M_2 to the total number of instances; and the hard-linkage accuracy is the ratio of the total number of cases in which the predictive slice contains the line corresponding to the `return`-statement in the callee method M_2 to the total number of instances. Mathematically, these can be defined as:

$$SLA = \frac{\sum_{P \in D} \begin{cases} 1 & |S_P \cap \{z | z \in M_2\}| \neq 0 \\ 0 & otherwise \end{cases}}{|D|} \tag{4}$$

$$HLA = \frac{\sum_{P \in D} \begin{cases} 1 & |S_P \cap \{r | r \in M_2, r = M_2^{return}\}| \neq 0 \\ 0 & otherwise \end{cases}}{|D|} \tag{5}$$

where M_2^{return} indicates that r is the line number corresponding to the `return`-statement in the callee method M_2. Note that HLA is a stricter evaluation metric, while SLA is the relaxed version.

8.3.2 Empirical Results. In this experiment, we leveraged two best-performing models in Section 5.3 within the ND-Slicer framework to predict the dynamic backward slices: GraphCodeBERT+Transformer (B_4) and CodeExecutor+Transformer (B_6). We observed that B_4 achieves an HLA of 40.2% and an SLA of 72.5%. In contrast, B_6 achieves an HLA of 44.0% and an SLA of 76.0%, outperforming B_4 by 9.4% and 4.6%, respectively. These results are promising, more so because both GraphCodeBERT (off-the-shelf and during fine-tuning) and CodeExecutor (during fine-tuning) were trained on individual methods. This is an indication that those models have some capability of understanding the nuances in an inter-procedural execution.

While the lack of ground-truth backward slices poses a challenge in the direct assessment of the inter-procedural predictive slices, an alternative strategy is to extend ND-Slicer for M_1 and M_2 in P independently, considering l and r as the slicing criterion, respectively. Finally, the individual slices thus obtained (say $S_P^{M_1}$ and $S_P^{M_2}$), can be combined by inserting $S_P^{M_2}$ in $S_P^{M_1}$ following l. Further empirical research is needed to assess this strategy. In brief, ND-Slicer scales to an inter-procedural setting, showing its capability in understanding the nuances in an inter-procedural execution.

8.4 ND-Slicer in Action: A Case-Based Assessment

Let us use a few examples to illustrate and assess ND-Slicer in action.

Listing 3. An example with ID 12592

```
1   def solve(a, b, k):
2     l = []
3     for i in range(min(a, b)):
4       if a % (i + 1) == 0 and b % (i + 1) == 0:
5         l.append(i + 1)
6     return l[-k]
7
8   def main():
9      A = 8; B = 12; K = 2
10     print(solve(A, B, K))
11
12  if __name__ == '__main__':
13    main()
```

Listing 4. Execution information

```
1 Slicing Criterion: 10(2)
2 Predictive Slice for 2nd occurrence of line 10:
      10, 6, 5, 4, 3, 4, 3, 5, 4, 3, 5, 4, 3, 1
3 Execution Trace:
      1, 8, 12, 13, 8, 9, 10, 1, 2, 3, 4, 5, 3, 4, 5, 3,
      4, 3, 4, 5, 3, 4, 3, 4, 3, 4, 3, 4, 6, 10
```

Fig. 4. An illustration of inter-procedural, predictive slicing

8.4.1 Example 1. Fig. 4 illustrates an example from the CodeNetMut dataset (ID 12,592) for which the execution trace depicted in line 3 of Listing 4, reveals that the control flows from `main` method (line 8) to `solve` method (line 1). Interestingly, within the body of `solve`, we can see that the `for` loop iterated multiple times, wherein, the `if`-condition on line 4 directed the flow along different paths for each iteration. Using the second occurrence of statement 10 as the slicing criterion, we observed that ND-Slicer was able to correctly produce the predictive backward slice.

In this example, we observe a range of ND-Slicer's capabilities. Initially, it demonstrates a proficiency in conducting *inter-procedural slicing* seamlessly spanning from `main` to `solve` in the *accurate temporal sequence*. For instance, within the body of the method `solve`, the sub-slice begins at

the `return` statement on line 6 and unwraps accurately. Secondly, it exhibits a comprehension of the control flow intrinsic to the `return` statement, effectively grasping the *dynamic control dependency*, thus ensuring precise inter-procedural backward slicing. Thirdly, ND-Slicer shows its aptitude in capturing the dynamic intricacies inherent in the complex execution, particularly those involving conditions within an iterative construct (lines 3–5). This underscores its capability to comprehend the *dynamic data and control dependencies pertaining to intra-procedural dynamic slicing*. Moreover, it demonstrates accurate slicing through a `for` loop in the correct temporal sequence and with the highly correct number of iterations, substantiating our earlier findings in previous sections. Finally, with the encoding of inputs through assignments (line 9), ND-Slicer gains the ability to discern the *dynamic inter-dependencies between the inputs and the statements in the slice*.

Listing 5. An Example in CodeNetMut

Score	Line	Code
1.00	1	`n = 7`
0.99	2	`a = [1, 2, 3, 2, 1, 999999999, 1000000000]`
0.06	3	`up = False`
0.00	4	`down = False`
0.00	5	`ans = 1`
0.05	6	`for i in range(1, n):`
0.96	7	`  if up:`
0.09	8	`    if a[i] < a[i-1]:`
0.00	9	`      ans += 1`
0.04	10	`      up = False`
0.10	11	`  elif down:`
0.17	12	`    if a[i] > a[i-1]:`
0.00	13	`      ans += 1`
0.02	14	`      down = False`
0.00	15	`  else:`
0.10	16	`    if a[i] > a[i-1]:`
0.25	17	`      up = True`
0.03	18	`    elif a[i] < a[i-1]:`
0.21	19	`      down = True`
0.00	20	`print(ans)`

Listing 6. Execution and Slices

Slicing Criterion: 8(1)
✓ Predictive Slice: 8, 6, 16, 6, 1

Slicing Criterion: 8(2)
✓ Predictive Slice: 8, 6, 8, 6, 16, 6, 1

Slicing Criterion: 9(1)
✓ Predictive Slice: 9, 5

Slicing Criterion: 10(1)
✓ Predictive Slice: 10, 7, 17, 7, 3

Slicing Criterion: 17(1)
✓ Predictive Slice: 17, 7, 3

Slicing Criterion: 17(2)
✓ Predictive Slice: 17, 7, 10, 7, 17, 7, 3

Slicing Criterion: 18(1)
✓ Predictive Slice: 18, 16, 6, 8, 6, 8, 6, 16, 6, 1

Fig. 5. Backward predictive slicing for a program with complex logic (left), for different slicing criteria (right). Here, the heatmap represents attention scores ND-Slicer associated with each statement while slicing 17(1).

8.4.2 Example 2. Fig. 5 illustrates an example from the CodeNetMut dataset with intricate nested `if-else` structures within the `for` loop (Listing 5). We observed that ND-Slicer produces accurate predictive backward slices for various slicing criteria, including at lines 8, 9, 10, 17, and 18 (refer to Listing 6). In particular, let us discuss the slicing criteria at lines 8 and 17. We can see that during execution, line 17 is encountered twice, as distinct occurrences within different iterations of the `for` loop at line 6. Each of these iterations entails traversing through different branches within the loop, spanning lines 7 to 19. In the first iteration, the flow proceeds from lines 7 to 17, while in the subsequent iteration, it follows the path from line 7 to line 10. In the third as well as the last iteration, the flow again traces from lines 7 to 17. First, it is noteworthy that ND-Slicer not only correctly anticipated the number of iterations in the loop, but also the sequence of executed statements in each iteration in the correct order. Second, it adeptly discerns the correct branches for all iterations within a nested network of conditional statements. These observations underscore ND-Slicer's proficiency in capturing dynamic control dependencies within intricate logic structures. Third, it demonstrates a clear comprehension of data dependencies between input assignment statements and pertinent statements, yielding an exemplary performance in slicing through intricate execution paths encompassing multiple nested branches and loop iterations.

Upon a closer examination of the statement at line 8, we observed a similar demonstration of ND-Slicer's capabilities. We can see that line 8 is executed in two distinct iterations of the `for` loop, and in the first iteration, it remained inactive with the control flowing directly to line 16. In brief, we can see that ND-Slicer consistently showcases abilities in accurately predicting: 1) the iteration count, 2) branches traversed in each iteration, 3) dynamic control dependencies among input assignment statements and other statements, both individually and collectively, and 4) intra-procedural data dependencies among statements derived from the criterion. Thus, we can corroborate the fundamental principles driving our design, as elucidated in Section 2.2, which result in ND-Slicer's superior performance in predicting backward slices.

In Fig. 5, we also present a heatmap representing the attention scores for the corresponding statements in Listing 5 during the first step of the decoding chain, with the first occurrence of line 17 as the slicing criterion. We employed Attention Visualizer [1] to obtain the attention scores for all input (sub)tokens making up a statement, and record their maximum value as representing the statement in the heatmap. A higher attention score for a statement indicates a greater contribution to the model's prediction of the slice. The ground-truth dynamic backward slice, as well as the predictive slice from ND-Slicer for 17(1) encompasses the statements 17, 7, and 3 in that specific order. In the heatmap, we can see that an attention score of 1 is assigned to the input statement on line 1. Furthermore, a high value is also assigned to line 7 conditioned on the variable `up`, which when `False`, directs the flow to line 17. This illustrates ND-Slicer's ability to learn the dynamic relations between the input values and the statements in the backward slice.

9 DISCUSSION

9.1 Limitations

9.1.1 Programming Languages. In this study, we focused on the task of predictive slicing exclusively for Python programs because CodeExecutor, the execution-aware PLM used in ND-Slicer was only trained on Python. As a result, ND-Slicer is limited in its use and effectiveness for programs written in other programming languages. However, one can easily extend ND-Slicer framework to support a new programming language L by (1) pre-training an LLM on L for learning the inherent dynamic dependencies, (2) constructing similar parallel corpora with the tuples of programs, slicing criterion, and corresponding ground-truth dynamic backward slices. This underscores the potential for expanding of our tool to cater to diverse programming environments.

9.1.2 Third-Party Libraries. Another limitation of our tool is that the CodeNetMut dataset used for training our model does not include third-party libraries. This was so done because code repositories, such as those on GitHub, often present challenges for large-scale execution, mainly owing to their internal dependencies and need to set up run-time environments. However, the superior performance of ND-Slicer in our experiments and its exhibition of understanding dynamic data and control dependencies strongly indicates its potential for extension to specific projects. This can be achieved by extending the pre-training of CodeExecutor for individual projects, which will enable ND-Slicer to meet the unique demands of real-world code. Such an approach will facilitate the use of ND-Slicer for debugging project-specific code. This can be explored in future work.

9.1.3 Complex Programs. When faced with complex programs, ND-Slicer's performance can potentially be impacted by the following: (1) programs possessing the API elements in third-party libraries, (2) slicing-specific dependencies, (3) language-specific semantics, etc. While the LLM used in our work (i.e., CodeExecutor) is execution-aware to enable intra-procedural analysis, we reckon that large-scale LLMs (e.g., GPT-4) that possibly possess an understanding of the aforementioned aspects will help scale to even more complex programs. We could explore this in future work.

9.2 Threats to Validity

9.2.1 Ground-Truth Backward Slices. In this work, we leverage the statement execution history as a resource for extracting ground-truth dynamic backward slices, making use of the Dynamic Slicing Algorithm [4] presented in Section 5. While this approach has proven effective, it's worth noting, as pointed out by Agrawal and Harman [4], that it may occasionally result in larger dynamic slices due to some variables having multiple reaching definitions across different occurrences of that statement in the execution trace. An alternative approach involves leveraging dynamic instrumentation-based tools to extract ground-truth dynamic slices, albeit with significant performance overhead. Additionally, our approach to building the ground truth can be scaled and improved by combining different dynamic dependence analysis techniques as described in Section 10 (each, used according to situations it is best suited for), to enhance the training corpus with more precise and complete labels. The exploration of such techniques remains a promising avenue for future research.

9.2.2 Crash Fault Identification. Most real-world crash detection datasets (for e.g., BugsInPy [39]) record the crash faults within the `assert` statements in the test suite, often spanning multiple methods spread across different files. Due to ND-Slicer's design to probe for dynamic data and control dependencies in statements within individual files, it made it difficult to incorporate such datasets into our crash detection experiment (in Section 7). As a result, we constructed a synthetic crash detection dataset by injecting `ZeroDivisionError` faults into Python programs. An issue with such synthetic data, however, is that we do not know the precise fault locations. Thus, to assess model performance, we inspected whether the predictive slice from our tool contains at least one of the plausible fault locations, retrieved from the static PDG. By adopting a hybrid approach that combines traditional program analysis with our tool, wherein ND-Slicer is utilized for slicing independent method-level code, and combining such predictive slices based on method call history, such a framework can be facilitated for real-world crash detection as well.

9.3 Future Work

Looking ahead, there are several avenues for further exploration. Firstly, we could enhance the ND-Slicer framework by incorporating a broader range of dynamic and slicing-specific dependencies, such as dynamic forward slicing, conditioned slicing, amorphous slicing, and pre/post-conditioned slicing, along with static program slicing. For instance, we could integrate a data flow-aware Large Language Model (LLM) like GraphCodeBERT [15] to capture static program dependencies and produce corresponding slices. Secondly, predictive slicing is language-independent and this framework can be extended to different languages. This opens up the possibility of analyzing software written in multiple programming languages simultaneously. Thirdly, we can extend our neural-network-based dependence analysis to accommodate the specific dependencies inherent in various programming paradigms with differing program semantics. This includes scenarios like event-driven programs where program dependence may not be explicitly encoded in the source code. Fourthly, when dealing with incomplete code, we aim to explore the balance between the accuracy of approximating slices and the degree of incompleteness in the source code. This would be useful for programming assistant tools for code under editing. Lastly, to make ND-Slicer applicable to large codebases, we intend to investigate the trade-offs between scalability and accuracy of the predictive slices. This holds promise for enhancing downstream tasks reliant on dynamic slicing.

10 RELATED WORK

In the realm of program slicing, extensive research has been conducted, as evidenced by a comprehensive body of work [17, 35, 37]. However, none of the existing methodologies have been tailored to handle incomplete code snippets.

NeuralPDA [42], a deep learning paradigm, is capable of deriving the static program dependencies in code snippets, whether complete or incomplete. It demonstrates the feasibility of a learning-based approach to learn static program dependencies – which motivated us to explore such a paradigm for learning more precise dynamic dependencies corresponding to a specific execution.

CodeExecutor [24], a Transformer model, employs code execution pre-training to predict execution traces and associated variable values. With these predicted traces, one can construct dynamic backward/forward slices using established slice construction algorithms [4]. Our experimental results demonstrate superior performance of ND-Slicer compared to this combined approach.

LExecutor [36], a learning-guided technique for executing code snippets, anticipates missing values that might otherwise impede program execution. It rectifies this by injecting the requisite values into the execution process. In a similar vein, TRACED [12] utilizes pre-trained code language models, incorporating source code, executable inputs, and corresponding execution traces for fine-tuning to predict execution outcomes. Both approaches possess program execution-specific knowledge, and hold promise for establishing further predictive dependence analyses.

Surveys have been conducted on various facets of program slicing techniques [6, 8, 17–19, 27, 37, 41]. Silva [35] and Harman *et al.* [17] present extensive surveys with multi-dimensional classifications of program-slicing methodologies.

There exist static slicing methodologies grounded in diverse static analyses, including incremental slicing [30], call-mark slicing [29], proposition-based slicing [21], stop-list slicing [14], and amorphous slicing [16]. Additionally, dynamic slicing methodologies [7, 22, 23, 28], encompassing language-independent slicing [7], generate a slice for a specific execution. There exist other families of slicing: conditioned program slicing family [10, 11], constraint slicing [13], and pre/post-conditioned slicing [20], which define an initial state through specified conditions.

11 CONCLUSION

In conclusion, this paper introduces ND-Slicer, an innovative predictive slicing methodology designed to address the limitations of traditional slicing techniques in scenarios where executing specific inputs is impractical or infeasible. By leveraging execution-aware pre-training, ND-Slicer acquires a deep understanding of dynamic program dependencies, encompassing both dynamic data and control dependencies between variables in the slicing criterion and the remaining program statements. This knowledge forms the foundation for constructing a predictive backward slice. Our empirical evaluation demonstrates the high accuracy of ND-Slicer, achieving an exact-match accuracy of 81.3% and a ROUGE-LCS F1-score of 95.4% on Python programs. In addition, as demonstrated in the extrinsic evaluation, ND-Slicer proves its utility in crash detection, exhibiting an impressive fault detection accuracy of 63.9%. An in-depth qualitative assessment further underscores ND-Slicer's proficiency in comprehending complex program structures, including `if-else` blocks, loops, and control flow in inter-procedural calls. These results affirm the efficacy and promise of ND-Slicer as a valuable tool in the realm of program slicing and debugging. ND-Slicer could serve as a foundation for a new direction on machine learning (ML)-based program dependence analysis.

12 DATA AVAILABILITY

Our datasets and models are all publicly available [3].

ACKNOWLEDGMENTS

We extend our thanks to the reviewers for their constructive feedback and motivating remarks. This work was supported in part by the US National Science Foundation (NSF) grant CNS-2120386 and the National Security Agency (NSA) grant NCAE-C-002-2021.

REFERENCES

[1] [n. d.]. https://github.com/AlaFalaki/AttentionVisualizer.

[2] 2022. *A static analysis library for computing graph representations of Python programs.* https://github.com/google-research/python-graphs

[3] 2023. *Replication package for ND-Slicer.* https://github.com/aashishyadavally/nd-slicer

[4] Hiralal Agrawal and Joseph R. Horgan. 1990. Dynamic program slicing. In *Proceedings of the ACM SIGPLAN 1990 Conference on Programming Language Design and Implementation* (White Plains, New York, USA) *(PLDI '90)*. Association for Computing Machinery, New York, NY, USA, 246–256. https://doi.org/10.1145/93542.93576

[5] Esben Andreasen, Liang Gong, Anders Møller, Michael Pradel, Marija Selakovic, Koushik Sen, and Cristian-Alexandru Staicu. 2017. A Survey of Dynamic Analysis and Test Generation for JavaScript. *ACM Comput. Surv.* 50, 5, Article 66 (sep 2017), 36 pages. https://doi.org/10.1145/3106739

[6] David Binkley and Keith Gallagher. 1996. Program Slicing. *Journal of Advanced Computing* 43 (1996), 1–50.

[7] David Binkley, Nicolas Gold, Mark Harman, Syed Islam, Jens Krinke, and Shin Yoo. 2014. ORBS: Language-independent Program Slicing. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 109–120.

[8] David Binkley and Mark Harman. 2004. A survey of empirical results on program slicing. *Journal of Advanced Computing* 62 (2004), 105–178.

[9] Derek Bruening, Timothy Garnett, and Saman Amarasinghe. 2003. An infrastructure for adaptive dynamic optimization. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization* (San Francisco, California, USA) *(CGO '03)*. IEEE Computer Society, USA, 265–275.

[10] G. Canfora, A. Cimitile, and A. De Lucia. 1998. Conditioned Program Slicing. *Inf. Soft. Technology* 40, 11-12 (1998), 595–608.

[11] Andrea de Lucia, Anna Rita Fasolino, and Malcolm Munro. 1996. Understanding Function Behaviors Through Program Slicing. In *Proceedings of the 4th International Workshop on Program Comprehension*. IEEE Computer Society, 9–18.

[12] Yangruibo Ding, Benjamin Steenhoek, Kexin Pei, Gail E. Kaiser, Wei Le, and Baishakhi Ray. 2024. TRACED: Execution-aware Pre-training for Source Code. In *Proceedings of the International Conference on Software Engineering (ICSE'24)*. ACM Press.

[13] John Field, G. Ramalingam, and Frank Tip. 1995. Parametric Program Slicing. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM, 379–392.

[14] Keith Gallagher, David Binkley, and Mark Harman. 2006. Stop-List Slicing. In *Proceedings of the Sixth IEEE International Workshop on Source Code Analysis and Manipulation*. IEEE Computer Society, 11–20.

[15] 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 B. Clement, Dawn Drain, Neel Sundaresan, Jian Yin, Daxin Jiang, and Ming Zhou. 2021. GraphCodeBERT: Pre-training Code Representations with Data Flow. In *ICLR*. OpenReview.net.

[16] Mark Harman and Sebastian Danicic. 1997. Amorphous Program Slicing. In *Proceedings of the 5th International Workshop on Program Comprehension*. IEEE Computer Society, 70–79.

[17] Mark Harman, Sebastian Danicic, Yoga Sivagurunathan, and Dan Simpson. 1996. The Next 700 Slicing Criteria. In *Proceedings of the 2nd U.K. Workshop on Program Comprehension*.

[18] Mark Harman and Keith Gallagher. 1998. Program Slicing. *Inform. Softw. Technol.* 40 (1998), 577–582.

[19] Mark Harman and Robert Hierons. 2001. An overview of program slicing. *Softw. Focus* 3 (2001), 85–92.

[20] Mark Harman, Robert Hierons, Chris Fox, Sebastian Danicic, and John Howroyd. 2001. Pre/Post Conditioned Slicing. In *Proceedings of the IEEE International Conference on Software Maintenance*. IEEE Computer Society, 138–147.

[21] John Hatcliff, Matthew B. Dwyer, and Hongjun Zheng. 2000. Slicing Software for Model Construction. *Higher Order Symbolic Computation* 13, 4 (2000), 315–353.

[22] Ranjit Jhala and Rupak Majumdar. 2005. Path Slicing. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 38–47.

[23] Bogdan Korel and Janusz Laski. 1988. Dynamic Program Slicing. *Inf. Process. Lett.* 29, 3 (Oct. 1988), 155–163.

[24] Chenxiao Liu, Shuai Lu, Weizhu Chen, Daxin Jiang, Alexey Svyatkovskiy, Shengyu Fu, Neel Sundaresan, and Nan Duan. 2023. Code Execution with Pre-trained Language Models. In *ACL (Findings)*. Association for Computational Linguistics, 4984–4999.

[25] Tongping Liu, Charlie Curtsinger, and Emery D. Berger. 2016. DoubleTake: fast and precise error detection via evidence-based dynamic analysis. In *Proceedings of the 38th International Conference on Software Engineering* (Austin, Texas) *(ICSE '16)*. Association for Computing Machinery, New York, NY, USA, 911–922. https://doi.org/10.1145/2884781.2884784

[26] 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).

[27] Andrea De Lucia. 2001. Program slicing: Methods and applications. In *Proceedings of the 1st IEEE International Workshop on Source Code Analysis and Manipulation*. IEEE Computer Society, 142–149.

[28] J. Maras, J. Carlson, and I. Crnkovic. 2011. Client-side web application slicing. In *26th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE Press, 504–507.

[29] Akira Nishimatsu, Minoru Jihira, Shinji Kusumoto, and Katsuro Inoue. 1999. Call-mark Slicing: An Efficient and Economical Way of Reducing Slice. In *Proceedings of the 21st International Conference on Software Engineering*. ACM, 422–431.

[30] Alex Orso, Saurabh Sinha, and Mary Jean Harrold. 2001. Incremental slicing based on data-dependence types. In *Proceedings of the IEEE International Conference on Software Maintenance*. IEEE Computer Society, 158–167.

[31] Nikhil Prabhu, Sewoong Min, Haewoon Nam, Girma Tewolde, and Jaerock Kwon. 2020. Integrated Framework of Autonomous Vehicle with Traffic Sign Recognition in Simulation Environment. In *2020 IEEE International Conference on Electro Information Technology (EIT)*. 514–521. https://doi.org/10.1109/EIT48999.2020.9208241

[32] 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).

[33] Gregor Richards, Sylvain Lebresne, Brian Burg, and Jan Vitek. 2010. An analysis of the dynamic behavior of JavaScript programs. In *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation* (Toronto, Ontario, Canada) *(PLDI '10)*. Association for Computing Machinery, New York, NY, USA, 1–12. https://doi.org/10.1145/1806596.1806598

[34] Abigail See, Peter J. Liu, and Christopher D. Manning. 2017. Get To The Point: Summarization with Pointer-Generator Networks. In *ACL (1)*. Association for Computational Linguistics, 1073–1083.

[35] Josep Silva. 2012. A Vocabulary of Program Slicing-based Techniques. *ACM Comput. Surv.* 44, 3, Article 12 (June 2012), 41 pages.

[36] Beatriz Souza and Michael Pradel. 2023. LExecutor: Learning-Guided Execution. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (, San Francisco, CA, USA,) *(ESEC/FSE 2023)*. Association for Computing Machinery, New York, NY, USA, 1522–1534. https://doi.org/10.1145/3611643.3616254

[37] Frank Tip. 1994. *A Survey of Program Slicing Techniques*. Technical Report. Amsterdam, The Netherlands.

[38] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Ł ukasz Kaiser, and Illia Polosukhin. 2017. Attention is All you Need. In *Advances in Neural Information Processing Systems*, I. Guyon, U. Von Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett (Eds.), Vol. 30. Curran Associates, Inc. https://proceedings.neurips.cc/paper_files/paper/2017/file/3f5ee243547dee91fbd053c1c4a845aa-Paper.pdf

[39] Ratnadira Widyasari, Sheng Qin Sim, Camellia Lok, Haodi Qi, Jack Phan, Qijin Tay, Constance Tan, Fiona Wee, Jodie Ethelda Tan, Yuheng Yieh, Brian Goh, Ferdian Thung, Hong Jin Kang, Thong Hoang, David Lo, and Eng Lieh Ouh. 2020. BugsInPy: a database of existing bugs in Python programs to enable controlled testing and debugging studies. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Virtual Event, USA) *(ESEC/FSE 2020)*. Association for Computing Machinery, New York, NY, USA, 1556–1560. https://doi.org/10.1145/3368089.3417943

[40] Baowen Xu, Zhenqiang Chen, and Hongji Yang. 2002. Dynamic slicing object-oriented programs for debugging. In *Proceedings. Second IEEE International Workshop on Source Code Analysis and Manipulation*. 115–122. https://doi.org/10.1109/SCAM.2002.1134111

[41] Baowen Xu, Ju Qian, Xiaofang Zhang, Zhongqiang Wu, and Lin Chen. 2005. A brief survey of program slicing. *SIGSOFT Softw. Eng. Notes* 30, 2 (mar 2005), 1–36. https://doi.org/10.1145/1050849.1050865

[42] Aashish Yadavally, Tien N. Nguyen, Wenbo Wang, and Shaohua Wang. 2023. (Partial) Program Dependence Learning. In *Proceedings of the 45th International Conference on Software Engineering* (Melbourne, Victoria, Australia) *(ICSE '23)*. IEEE Press, 2501–2513. https://doi.org/10.1109/ICSE48619.2023.00209

[43] Tianyi Zhang, Di Yang, Crista Lopes, and Miryung Kim. 2019. Analyzing and Supporting Adaptation of Online Code Examples. In *Proceedings of the 41st International Conference on Software Engineering (ICSE '19)*. IEEE Press, 316–327. https://doi.org/10.1109/ICSE.2019.00046

Received 2023-09-29; accepted 2024-01-23