NEURAL MODELING OF REASONING ABOUT PROGRAM BEHAVIORS

by

Aashish Yadavally

APPROVED BY SUPERVISORY COMMITTEE:

Tien N. Nguyen, Chair

Wei Yang

Shiyi Wei

Baishakhi Ray

NEURAL MODELING OF REASONING ABOUT PROGRAM BEHAVIORS

by

AASHISH YADAVALLY, BTech, MS

DISSERTATION

Presented to the Faculty of

The University of Texas at Dallas

in Partial Fulfillment

of the Requirements

for the Degree of

DOCTOR OF PHILOSOPHY IN

COMPUTER SCIENCE

THE UNIVERSITY OF TEXAS AT DALLAS

August 2025

ACKNOWLEDGMENTS

This dissertation is the result of a long and exciting journey, marked by many sleepless nights and countless cups of coffee. I would like to take a moment to express my deepest gratitude to everyone who has supported and guided me along the way.

First, I would like to thank my advisor, Dr. Tien N. Nguyen, whose mentorship has shaped the way I think about research. His guidance on both personal and professional fronts, and belief in me and my work have made this journey both meaningful and fulfilling. I am also thankful to the members of my doctoral committee, Drs. Wei Yang, Shiyi Wei, and Baishakhi Ray, for their time and support, and for the constructive feedback throughout this process.

I would like to thank my labmates and collaborators, Dr. Shaohua Wang, Dr. Yi Li, Yuchen Cai, Wenbo Wang, Hridya Dhulipala, Smit Patel, and Xiaokai Rong, for the brainstorming sessions, shared wins (and frustrations) that made research a truly collaborative experience.

To my family: my mom, dad and my sister, Aishwarya, thank you for your endless love, encouragement, and sacrifices. You dreamed of this with me and your belief in me has been the foundation of everything I have achieved. One moment I often think about is a conversation with my dad during a car ride, when I told him I wanted to pursue teaching. It planted the seed for this journey. This dream was also shaped by my maternal grandfather, D.V. Raman, whose own PhD pursuit remained unfulfilled, but who carried a lifelong aspiration to see someone in the family complete it. I am thankful to have brought it to life!

To my friends: Faisal, Abhilash, Hemanth, Anuj, Sumer, Sushanth, Aravind, Ishika, Saurabh, and the many others who stood by me through late nights, deadlines, and emotional roller-coasters. Thank you for the laughter, coffee, and perspective. A special thanks to Shravani, for her patience, compassion, and the countless ways she reminded me to believe in myself.

Finally, I am grateful to the Computer Science Department at UT Dallas for providing a supportive and stimulating environment throughout my PhD. I am also thankful to my

mentors and colleagues at Amazon, particularly Gauthier Guinet, Hoan Nguyen, and Omer Tripp. Their mentorship helped me gain an industry perspective, enriching the way I approach problems and communicate ideas. The support I received from them, and from many others beyond academia, has played a meaningful role in shaping my professional journey.

June 2025

NEURAL MODELING OF REASONING ABOUT PROGRAM BEHAVIORS

Aashish Yadavally, PhD
The University of Texas at Dallas, 2025

Supervising Professor: Tien N. Nguyen, Chair

Programming languages, much like natural languages, exhibit a high degree of repetitiveness and regularity, often referred to as the *naturalness of software*. This characteristic, combined with the improved capabilities of neural language models (NLMs) to statistically learn from such patterns, has led to their widespread adoption in software engineering (SE) tasks ranging from code generation to automated bug detection and program repair. While these applications of automated software engineering offer a useful proxy for assessing the downstream performance of NLMs, their ability to reason about intrinsic program properties, such as structure, semantics, and execution behaviors, *remains underexplored*.

This dissertation addresses this gap through the lens of program analysis, using the latter's formalisms to probe the reasoning capabilities of NLMs over intrinsic program behaviors. In general, analyzing programs entails either examining all possible behaviors based on program semantics (*i.e.*, static) or establishing precise execution behaviors by running the entire test suite (*i.e.*, dynamic), each with trade-offs in generalizability and scalability. As an alternative, we introduce a new paradigm of *predictive program analysis*, which aims to learn to analyze program behaviors from similar analyses of open-source software repositories. This approximation helps extend such analyses to partial programs, enables a static estimation of runtime behaviors, and facilitates multilingual program analysis, all at scale. Using dependence analysis as a representative setting, this dissertation investigates how NLMs can

model program structure, semantics, and execution behaviors across three key dimensions: (*i*) the *granularity of dependencies*, ranging from inter-statement and variable-statement to inter-constraint dependencies; (*ii*) *nature of reasoning*, spanning both static and dynamic program behaviors; and (*iii*) *reasoning modality*, which involves reasoning in the latent space or through verbalized natural language explanations. Overall, these contributions show that predictive analysis can generalize, bridging the gap between static and dynamic analysis, while offering insights into how language models internalize reasoning about program behaviors.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1

INTRODUCTION

Computer programs written by software developers, much like natural language, are mostly simple and exceedingly repetitive (Hindle et al., 2016). This inherent "naturalness", combined with access to large-scale software repositories (*e.g.*, 100M+ projects on GitHub) opened the door to applying data-driven techniques to multiple software engineering (SE) tasks, ranging from code completion and bug detection to automated program repair. Early approaches to such automated software engineering relied on statistical language models or shallow machine learning techniques. However, recent advances in neural network-based language models (NLMs) such as Transformers (Vaswani et al., 2017) have significantly expanded the scope and effectiveness of these systems. While the broad spectrum of SE tasks provides a useful proxy for measuring their extrinsic applicability, a critical question arises:

How well do neural language models understand the intrinsic properties of source code?

At the center of this question lies the need to move beyond evaluating language models solely based on their ability to capture surface-level statistical correlations, which often conflates memorization with comprehension. We posit that a deep understanding of source code involves reasoning about its structural properties (*e.g.*, syntax, control flow), semantic properties (*e.g.*, data or type dependencies), and its behavioral properties (*e.g.*, how the program evolves during execution). The ability to model these intrinsic aspects is essential for generalizing across diverse SE tasks, while enabling interpretability and scalability.

Within the field of *natural language processing* (NLP), researchers have increasingly recognized the limitations of evaluating NLMs through only downstream task performance. Thus, recent efforts have focused on designing diagnostic probing tasks (Conneau et al., 2018; Belinkov, 2022) to analyze the models' understanding of intrinsic linguistic knowledge (*e.g.*,

syntax, semantics, and morphology). These probes provide valuable insights into what NLMs truly learn, as well as their limitations in generalization.

In contrast, in their applicability to programming languages and software engineering, such an intrinsic evaluation *remains underexplored*. Unlike natural language, source code has formally defined syntax and semantics, enabling precise reasoning about program properties and behaviors through rule-based methodologies, collectively termed *program analysis*. This characteristic of programming languages presents a unique opportunity: we can construct targeted evaluations of NLMs *grounded in program analysis* to probe their understanding of code structure, semantics, and execution behaviors. Such evaluations provide a foundational framework for *simulating reasoning about intrinsic program behaviors using NLMs*.

To this end, this dissertation introduces the paradigm of **predictive program analysis**, which reformulates traditional program analysis tasks as predictive problems. Rather than executing code or symbolically simulating all possible program paths, predictive program analysis leverages the ability of neural language models to *approximate* structural, semantic, and behavioral properties. This enables reasoning at scale and is particularly suited for under-specified scenarios involving incomplete (*e.g.*, due to dependency injection or partial snippets from online forums) or multilingual code. Furthermore, it sidesteps the need for actual program execution and runtime monitoring, an overhead for traditional analyses that is often impractical in security-sensitive and resource-constrained environments.

In particular, I focus on *dependence analysis*, a fundamental class of program analysis that examines how different parts of a program influence each other, as a representative setting for predictive analysis. This provides a natural testbed for reasoning using NLMs, as it spans structural (*e.g.*, control dependencies), semantic (*e.g.*, data or logical dependencies) and behavioral (*e.g.*, execution-order dependencies) program properties. The central philosophy driving this line of work is that such reasoning, typically conducted using heuristic-based static or symbolic techniques, can instead be *learned* from similar analyses of complete code

repositories. In general, the proposed predictive analysis paradigm bridges the gap between formal, rule-based approaches and data-driven techniques, enabling scalable, execution-free analyses that provide deeper insights into the reasoning capabilities of neural language models.

1.1 Research Objectives

Predictive program analysis presents a new approach to probe the ability of NLMs to reason about source code. To operationalize this paradigm, this dissertation formulates three primary research objectives, each targeting a distinct dimension of reasoning about program behaviors:

① *Dependence Granularity.* This objective investigates whether NLMs can simulate reasoning about dependencies between program components at different levels of dependence granularity, ranging from fine-grained inter-statement and variable-statement dependencies to coarse-grained inter-constraint dependencies. Each level encodes different kinds of structural and semantic interactions, capturing: (a) control and data flow between code blocks, (b) how specific variables influence or are influenced by operations, (c) logical entailment to determine feasibility of program paths, *etc.*

② *Static v/s Dynamic Program Behaviors.* This dimension explores whether NLMs can reason about both static and dynamic program behaviors. The former involves reasoning over properties derived from program structure and semantics without execution (*e.g.*, control or data dependencies, type dependencies, *etc*). In contrast, the latter setting focuses on execution-specific behaviors, such as how variable values evolve or statements are ordered during runtime. This objective assesses whether predictive program analysis can widen the scope of static analyses by implicitly or explicitly filling in necessary missing information, and enhance dynamic analyses by enabling generalization to inputs beyond the test suite–thereby bridging the gap between both.

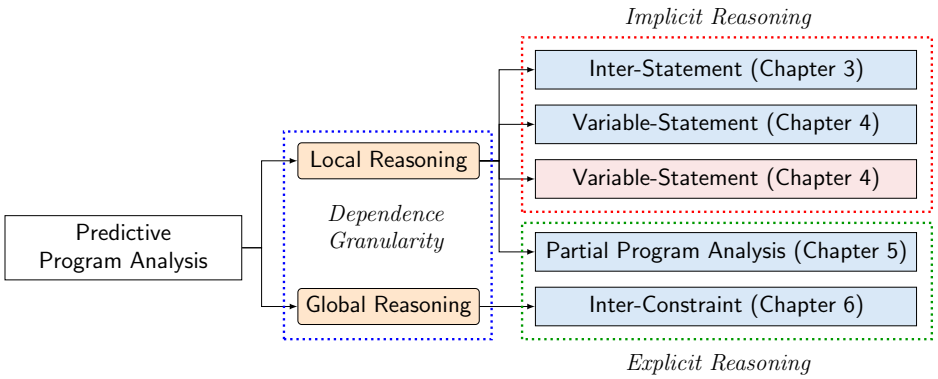


Figure 1.1: A taxonomy of different predictive program analysis tasks, organized along three key dimensions: (1) dependence granularity (local *v/s* global), (2) static (□) *v/s* dynamic (□) program behaviors, and (3) reasoning modality (implicit *v/s* explicit).

(3) *Reasoning Modality.* Based on the inherent architecture of NLMs, reasoning can be actualized either *implicitly*, in the latent space, or *explicitly*, through natural language explanations. In the first setting, reasoning about dependencies between program components is not directly observable; it can be qualitatively inferred from probing latent representations and quantitatively through performance on predictive program analyses. In contrast, the second setting produces verbalized reasoning steps alongside predictions, enabling more direct inspection of the model's rationale. Overall, this objective examines how the two modalities provide interpretable signals of how NLMs reason about intrinsic program behaviors across varied predictive formulations.

1.2 Contributions

1.2.1 Publications and Chapter Organization

The research conducted as a part of this dissertation addresses the limitations of traditional program analyses (discussed in more detail in Chapter 2) while providing a framework for evaluating how NLMs reason about intrinsic program behaviors at different levels of abstraction or dependency granularity. In Figure 1.1, we present a taxonomy of our contributions, which spans the three dimensions ((1)–(3)) introduced in Section 1.1:

[1] Reasoning about Inter-Statement Dependencies in Latent Space

Chapter 3 investigates the ability of NLMs to learn to analyze dependencies between pairs of program statements, specifically, control and data dependencies. These form the basis of program dependence graphs (PDGs), a widely-used representation in tasks like optimization, debugging, and program comprehension. To this end, we reformulate traditional compiler-based PDG construction as *structured link prediction* and introduce NeuralPDA, a neural network-based PDG builder that *implicitly* learns the semantic relationships between program statements by projecting them into a latent space as dense, contextualized representations.

Our hypothesis is grounded in an empirical study on the semantic repetitiveness of source code (Nguyen et al., 2016), which demonstrates that inter-statement dependencies exhibit regularities that can be statistically learned. Trained on complete programs, NeuralPDA generalizes well to incomplete code, achieving high accuracy in both Java and C/C++ while operating 380× faster than a traditional, state-of-the-art PDG-building tool. We also demonstrated its effectiveness in detecting vulnerabilities in incomplete StackOverflow code snippets, highlighting the usefulness of approximate PDGs from NeuralPDA in downstream analyses that can tolerate low levels of imprecision.

This chapter explores the ⟨*local, static, implicit reasoning*⟩ setting within the broader research design space, in particular, reasoning about inter-statement dependencies.

[2] Reasoning about Variable-Statement Dependencies in Latent Space

Chapter 4 investigates whether NLMs can *implicitly* reason about variable-statement dependencies, *i.e.*, determine which parts of a program may affect or be affected by a variable at a specific point of interest. Such analyses are essential for program slicing. Traditional slicing techniques, both static and dynamic, rely on constructing system dependence graphs (SDGs) and/or runtime trace analysis, which are computationally intensive and infeasible for

incomplete programs. Missing type or variable declarations, and ambiguities arising due to the absence of external libraries further limit their applicability.

To address these limitations, we introduce two neural network-based slicing approaches that model variable-statement dependencies directly in latent space. NS-Slicer focuses on static slicing and learns from structural and data flow-aware representations of code. In contrast, ND-Slicer targets dynamic slicing by incorporating execution-aware pre-training, bypassing the need for actual program execution. Both approaches produce *approximate yet highly accurate* program slices and generalize well to incomplete programs. We further demonstrate their usefulness in detecting vulnerabilities and localizing crash faults, respectively, highlighting the potential of pre-trained NLMs as reasoning engines for program slicing.

This chapter explores both ⟨*local, static, implicit reasoning*⟩ and ⟨*local, dynamic, implicit reasoning*⟩ settings within the broader research design space, in particular, reasoning about variable-statement dependencies.

[3] Toward Comprehensive Partial Program Dependence Analysis

Dependence analysis (DA) tools typically assume access to the complete codebase, using syntactic and semantic information to construct an accurate model of dependencies. However, in many real-world scenarios, due to modular development or reusing code from online forums such as StackOverflow, only partial programs are available. These often lack definitions of variables, functions, import statements, and third-party libraries. Given such incomplete code, Chapters 3 and 4 explored the use of NLMs to directly learn to predict the presence or absence of dependencies between program components. In contrast, Chapter 5 investigates whether NLMs can explicitly reason about type dependencies and reconstruct its syntactically and semantically complete variant. This disambiguation enables DA tools to precisely reason about previously unresolved program elements. More broadly, by harnessing the complementary strengths of NLMs for augmenting missing context (*i.e.*, improving recall) and of DA tools

for enforcing semantic correctness (*i.e.*, ensuring high precision), this chapter introduces a blended approach that advances dependence analysis for partial programs.

This chapter explores the ⟨*local, static, explicit reasoning*⟩ setting within the broader research design space, in particular, reasoning about intra-fragment dependencies.

[4] Chain-of-Thought Reasoning about Inter-Constraint Dependencies

Following the *implicit* reasoning about dependencies between program components (Chapters 3 and 4) and the *explicit* reasoning over type dependencies (Chapter 5), Chapter 6 shifts focus to reasoning about logical constraints that govern program execution paths. In particular, this chapter explores whether NLMs can reason about inter-constraint dependencies and their contribution to unsatisfiability within string-based path conditions. In doing so, it explores the generation of natural language rationales (*i.e.*, Chain-of-Thought) by NLMs to explain their internal reasoning processes in the context of symbolic analysis.

We introduce SafeMin, a framework that minimizes a given string constraint system by identifying a (much) smaller subset that preserves unsatisfiability and reflects the main sources of inconsistency. Traditional approaches using SMT solvers typically rely on an exhaustive exploration of constraint space, which can be inefficient. In contrast, SafeMin leverages NLMs to macro-reason about inter-constraint relationships (*e.g.*, semantic equivalence, syntactic containment, logical contradictions) to identify redundancies. It also incorporates a *sample-and-enumerate* strategy to enable parallelized, partial enumeration of such inconsistencies. Overall, this chapter extends reasoning capabilities of NLMs from structural and type analyses to symbolic domains, enabling applications such as infeasible path detection.

This chapter explores the ⟨*global, static, explicit reasoning*⟩ setting within the broader research design space, in particular, reasoning about inter-constraint dependencies.

1.2.2 Additional Publications and Broader Impact

Beyond the research presented in this dissertation, I have contributed to complementary research efforts, resulting in 13 peer-reviewed publications (10 full and 3 short papers). I have also served the research community by serving as a Program Committee member or Reviewer for conferences and journals including ICSE, ICLR, MSR, EASE, TSE, and EMSE. I was recognized with the **Distinguished Reviewer Award** for my service on the MSR 2024 Junior Program Committee. A complete list of my publications, service roles, and outreach activities can be found in the *Curriculum Vitae* section at the end of this dissertation.

CHAPTER 2

BACKGROUND AND RELATED WORK

2.1 Naturalness of Software ⇝ Neural Language Models for Software Engineering

Linguists often analyze the syntactic, semantic, and morphological properties of natural language. While these aspects contribute to its theoretical complexity, everyday language is far simpler and repetitive. This predictability motivated computational linguists to apply statistical methods to natural language texts, resulting in the field of *natural language processing*. Drawing a parallel, (Hindle et al., 2016) extend the same argument to programming languages and hypothesized that:

> "Programming languages, in theory, are complex, flexible and powerful, but the programs that real people actually write are mostly simple and rather repetitive, and thus they have usefully predictable statistical properties that can be captured in statistical language models and leveraged for software engineering tasks."

In exploring the "naturalness" of programming languages, (Gabel and Su, 2010) conducted one of the earliest large-scale studies, analyzing over 420 million lines of code across three different programming languages, finding that code fragments were often syntactically redundant. Building on this, (Hindle et al., 2016) demonstrated that statistical language models (*e.g.*, n-gram models) capture project-specific code patterns beyond syntax. These findings have since prompted the adoption of statistical models in diverse SE tasks, including code completion (Nguyen et al., 2013; Schmidhuber, 2004), bug detection (Wang et al., 2016), and syntax error localization (Campbell et al., 2014).

Despite their early success, n-gram models are limited in their ability to capture deeper syntactic or semantic information, often resulting in poor predictive performance (*i.e.*, low precision). With the increasing availability of large-scale code repositories (*e.g.*, more than 100M code repositories on GitHub), there was a transition from phrase-based statistical

machine translation (MT) approaches (Nguyen et al., 2013; Karaivanov et al., 2014; Nguyen et al., 2022) to neural architectures (originally designed for processing natural language). These include tree-based (Chakraborty et al., 2020; Zhang et al., 2019) and graph-based (Allamanis et al., 2018) neural networks, which explicitly model structure in source code. Following the successes of attention in machine learning, Transformers (Vaswani et al., 2017)-based model architectures further advanced this line of work. Encoder-only models have been used for generating source code representations (Feng et al., 2020a; Guo et al., 2021, 2022), useful for code classification tasks such as clone detection (Feng et al., 2020a; Svajlenko et al., 2014), decoder-only models for code generation (OpenAI, 2023a; Li et al., 2023), and encoder-decoder models for code translation (Wang et al., 2021), summarization (Li et al., 2023; Husain et al., 2019), and editing (Chakraborty et al., 2020), among others.

2.1.1 Attention is All You Need

While early neural language models (NLMs) relied on recurrent neural networks (RNNs) and their variants for sequence modeling, they suffered from limited parallelizability and difficulties in capturing long-range dependencies. To address these limitations, (Bahdanau et al., 2015) introduced the attention mechanism, enabling models to dynamically focus on relevant parts of the input sequence. This significantly improved performance on a range of NLP tasks. Building on this idea, (Vaswani et al., 2017) proposed the Transformer architecture, which discards recurrence and relies solely on self-attention. It adopts an encoder-decoder framework composed of stacked layers of multi-head self-attention and feed-forward neural networks.

Formally, given an input sequence of token representations $X \in \mathbb{R}^{n \times d}$, self-attention computes a weighted representation of the sequence using the following:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^\top}{\sqrt{d_k}}\right)V$$

where query, key, and value matrices are $Q = XW^Q$, $K = XW^K$, and $V = XW^V$, respectively; and W^Q, W^K, W^V are learned parameter matrices. This formulation allows each token to

attend to all other tokens in the sequence, with attention weights determined by the scaled dot-product similarity between query and key vectors.

The Transformer encoder is a stack of identical layers, each consisting of two sublayers: a multi-head self-attention mechanism and a position-wise feed-forward neural network. Given input token embeddings augmented with positional encodings, which inject information about token order into the model, it maps them into a sequence of continuous representations through self-attention. Here, each token representation is updated by attending to all other tokens in the sequence, enabling the encoder to capture contextual relationships.

The Transformer decoder is similarly composed of stacked layers, but with three sublayers in each: masked multi-head self-attention, encoder-decoder attention, and a feed-forward neural network. The masked self-attention ensures that each token can only attend to earlier positions in the output sequence, preserving the autoregressive property needed for generation. The encoder-decoder attention sublayer allows the decoder to attend over all positions in the encoder's output, enabling the integration of source-side context during generation.

2.1.2 Pre-Trained Language Models

While the original Transformer was designed for sequence-to-sequence tasks, as noted earlier, its modular architecture has enabled the development of specialized variants optimized for distinct modeling objectives: encoder-only models for understanding tasks, decoder-only models for generation, and encoder-decoder models for translation and summarization. In this section, we introduce different classes of Transformer-based NLMs:

Encoder-Only Models. These are primarily designed for language understanding tasks such as text classification or named-entity recognition by training on specialized learning objectives such as masked language modeling (MLM) (Devlin et al., 2019), replaced token detection (Clark et al., 2020), *etc.* Encoder-only models build deep, context-aware representations of the input text. The following form the base for most encoder-only models:

- **BERT** (Devlin et al., 2019). Unlike the directional models that read text as a sequence in a particular direction (left-to-right or right-to-left), BERT reads the whole text as a sequence, thus *bidirectional*. It is primarily pre-trained using two learning objectives: masked language modeling (MLM) and next sentence prediction (NSP). In MLM, a subset of tokens in the input sequence is randomly masked, and the model is trained to predict the original tokens conditioned on the unmasked tokens (akin to a fill-in-the blanks task). The MLM loss is computed as a cross-entropy over the masked positions:

$$\mathcal{L}_{\text{MLM}}(\Theta) = \sum_{i=1}^{n} \sum_{t=1}^{T} \mathbf{1}\{x_t^{(i)} = \texttt{[MASK]}\} \cdot \log p_\theta(x_t^{(i)} \mid \mathbf{x}_{<t}^{(i)}, \mathbf{x}_{>t}^{(i)})$$

  In addition to MLM, BERT is also trained using the next sentence prediction (NSP) objective, which aims to predict whether two input segments appear consecutively in the original corpus. This dual-objective pretraining enables BERT to learn rich, contextual representations at both the token and sentence levels, making it highly effective for a wide range of downstream natural language understanding tasks.

- **RoBERTa** (Liu et al., 2019). Motivated by the hypothesis that BERT's performance was limited by under-training, RoBERTa was proposed as a more robust variant that retains the same model architecture but optimizes the training process. In particular, it removes the NSP learning objective, extends the number of training iterations, increases batch size, and leverages a substantially larger and more diverse corpus. In contrast to BERT's static masking, RoBERTa introduces dynamic masking, in which token masking is applied at runtime during training, allowing the model to encounter different masking patterns across epochs. Collectively, these changes lead to more stable optimization and improved generalization across a wide range of language understanding benchmarks.

For our experiments, we consider the following source code-specific encoder-only models:

- **CodeBERT** (Feng et al., 2020a). A bimodal pre-trained model jointly trained on natural language and programming languages using a masked language modeling (MLM) and replaced token detection (RTD) objective (Clark et al., 2020). It is based on the RoBERTa architecture and trained on a large-scale corpus of paired code-comment data, enabling it to learn cross-modal representations useful for code search, code summarization, and documentation generation.
- **GraphCodeBERT** (Guo et al., 2021). An extension of CodeBERT that incorporates structural information from program graphs. It is trained with both data flow edges and sequential token information, enabling it to better capture syntax and semantics through additional pretraining tasks such as edge prediction. GraphCodeBERT improves performance on downstream tasks like code-to-code translation and clone detection.

Decoder-Only Models. These are designed for generation tasks, where text is produced autoregressively, *i.e.*, one token at a time, conditioned on the leftward context. To this end, they are trained using the causal language modeling (CLM) learning objective, which involves predicting the next token given all previous tokens. Formally, the CLM loss is computed as:

$$\mathcal{L}_{\text{CLM}}(\Theta) = -\sum_{i=1}^{n}\sum_{t=1}^{T} \log p_\theta(x_t^{(i)} \mid \mathbf{x}_{<t}^{(i)})$$

This makes them particularly effective for code generation and completion. We consider the following decoder-only models in our experiments:

- **GPT** (OpenAI, 2023b): A family of large-scale autoregressive language models that are composed of stacked Transformer decoder blocks that include masked self-attention, position-wise feed-forward layers, residual connections, and layer normalization. They are trained on large-scale web corpora using maximum likelihood estimation, and generate contextually relevant completions given a natural language or code prompt.

- **StarCoder** (Li et al., 2023): A family of decoder-only models trained specifically on source code and natural language from The Stack, a code corpus spanning over 80 programming languages. StarCoder incorporates features such as fill-in-the-middle (FIM) training and identifier-aware tokenization, making it well-suited for code completion and infilling tasks within software development workflows.

Encoder-Decoder Models. These are designed to perform sequence transduction tasks, where an input sequence is encoded into a latent representation and then decoded into an output sequence. This architecture is especially well-suited for applications such as code summarization, translation, and generation. Typical pre-training objectives for encoder-decoder models include denoising autoencoding (Lewis et al., 2020), span corruption (Raffel et al., 2020), and masked span prediction for source code (Wang et al., 2021). More recent models have extended these objectives with cross-modal pre-training across code and natural language (Guo et al., 2022)) or with dynamic behavior modeling (Liu et al., 2023).

We consider the following encoder-decoder models in our experiments:

- **UniXCoder** (Guo et al., 2022). A unified cross-modal encoder-decoder model designed to support both unimodal and bimodal tasks involving source code and natural language. It is pre-trained using a denoising objective over multilingual code and natural language data, enabling it to capture both syntactic and semantic correspondences across modalities. Notably, UniXCoder introduces a mode-specific control mechanism that allows it to operate in encoder-only, decoder-only, or encoder-decoder settings within a shared architecture. This flexibility enables strong performance across a wide range of tasks, including code retrieval, summarization, and translation.
- **CodeExecutor** (Liu et al., 2023). Built upon the UniXCoder base model, CodeExecutor is a Transformer-based encoder-decoder model that significantly enhances its semantic comprehension of code through code execution pre-training and curriculum learning.

Unlike models that primarily rely on static source code and syntactic structures, CodeExecutor is uniquely trained to predict execution traces, including dynamic information like line order and intermediate program states during execution. This pre-training learning objective allows it to understand and predict the dynamic behavior and flow of code, making it particularly effective for tasks such as code execution prediction, zero-shot code-to-code search, and text-to-code generation.

2.1.3 Closed-Source Large Language Models

In addition to open-source pre-trained models, this dissertation evaluates the reasoning capabilities of several prominent closed-source large language models (LLMs), including GPT from OpenAI (OpenAI, 2023b), Claude from Anthropic (Anthropic, 2023), and Gemini from Google (DeepMind, 2024). These models are trained on large-scale corpora using undisclosed architectures and optimization strategies, which often include reinforcement learning from human feedback (RLHF) (Ouyang et al., 2022) and other alignment techniques. Despite limited transparency, they have demonstrated state-of-the-art performance across a wide range of natural language and code-related tasks. In this dissertation, they are used as off-the-shelf agents in prompting-based evaluations of program analysis tasks. Their inclusion provides a practical benchmark for assessing how well general-purpose, instruction-tuned models can simulate reasoning about program behaviors without explicit fine-tuning.

2.2 Foundations of Program Analysis

2.2.1 Reasoning about Static and Dynamic Program Behaviors

Program analysis is the process of examining a program to reason about its properties, structures, or behaviors. It enables several downstream applications in software engineering, including program verification (Griggio, 2009), optimization (Bruening et al., 2003), debugging (Xu et al., 2002), comprehension, and test generation (Andreasen et al., 2017). There

are two main themes of program analysis: *static* and *dynamic*. The former uses program semantics to examine and reason over all possible behaviors that might arise at runtime, while the latter uses actual execution for specific inputs to establish precise runtime behaviors.

While static analyses are often comprehensive, they scale super-linearly and introduce complex interactions. Moreover, since they approximate certain aspects of program behavior (such as user input or network state), they also need to deal with the resulting unknowns, leading to a tendency to *overestimate* potential behaviors. Thus, they are less precise, thereby limiting their overall effectiveness. By contrast, dynamic analyses provide precise insights over observed executions of the test suite. As such, it is an *under-approximation* and can not generalize, as the test suite may not cover all possible execution paths the program can take. It is desirable to bridge the gap between static and dynamic analyses, compromising a bit on the soundness of the former and the precision of the latter (Ernst, 2004).

A fundamental class of program analysis is dependence analysis, which focuses on reasoning about the extent to which one component of a program depends on another. Two forms of dependence are typically studied: *control dependence*, which determines whether the execution of one statement is predicated on another (*e.g.*, conditionals and loop guards); and *data dependence*, which captures whether a value produced by one statement is consumed by another. The program dependence graph (PDG) (Ferrante et al., 1987) is a standard representation for modeling such dependencies. They are useful in understanding program behaviors and serve as the basis for multiple applications, including optimization (Ferrante et al., 1987; Kuck et al., 1981), slicing (Weiser, 1984), debugging (Podgurski and Clarke, 1990), testing (Zhu et al., 1997), and model checking (Visser et al., 2000).

Compiler-Based Analysis. Traditional program analysis are built on static analyses performed by compilers. These operate over different program representations such as abstract syntax trees (for extracting structural information), control and data flow (for extracting

semantic information), and support downstream tasks like type checking, reachability analysis, program slicing, *etc.* These methods assume access to the complete codebase to provide sound approximations of program behaviors. However, when the program under analysis is incomplete, due to missing declarations of variables or functions, import statements, unresolved bindings, or being sourced from online forums (*e.g.*, StackOverflow), compiler-based pipelines often fail to parse or analyze the code altogether.

Partial Program Analysis. PPA (Dagenais and Hendren, 2008a) estimates missing bindings in partial Java programs using heuristics and partial type inference. GRAPA (Zhong and Wang, 2017) addresses this challenge by resolving unknown identifiers via archive-based resolution; however, its performance degrades when the surrounding context is too sparse or conflicting. JCoffee (Gupta et al., 2020), a static analysis tool, extends this line of work by leveraging compiler feedback to iteratively transform partial Java snippets into compilable code by generating stubs for missing components and incrementally repairing the modified program until it becomes syntactically valid. However, it still relies on syntactic correctness, often failing when initial parsing is blocked by incomplete constructs. ReACC (Lu et al., 2022), a retrieval-augmented model, takes a data-driven approach by using sparse and dense retrieval mechanisms to enhance code completion from partial code. However, its effectiveness diminishes for semantically uncommon snippets due to limitations in retrieval coverage. Mainstream IDEs such as IntelliJ IDEA (JetBrains, 2024) and Eclipse (Eclipse Foundation, 2024) also attempt to build ASTs from incomplete code fragments, but these techniques often fail in the absence of declarations or import context. These limitations collectively motivate the need for context-aware models that can reason about incomplete programs.

Program Slicing. There is a rich literature on program slicing, including surveys comprising comprehensive taxonomies (Silva, 2012; Harman et al., 1996; Binkley and Gallagher, 1996; Xu et al., 2005). These classify slicing approaches along dimensions such as form (*i.e.*, static, dynamic, or conditioned), granularity, direction (*i.e.*, forward/backward), and representation

(*i.e.*, PDG/SDG-based). In particular, static program slicing includes a variety of specialized implementations. Incremental slicing (Orso et al., 2001) focuses on reusing slices across program versions, while call-mark slicing (Nishimatsu et al., 1999) leverages inter-procedural boundaries. Proposition-based slicing (Hatcliff et al., 2000), stop-list slicing (Gallagher et al., 2006), and amorphous slicing (Harman and Danicic, 1997) generalize or relax traditional syntactic constraints to improve slicing accuracy. Several extensions of static slicing have also emerged, such as conditioned slicing (Canfora et al., 1998; de Lucia et al., 1996), constraint slicing (Field et al., 1995), and pre/post-conditioned slicing (Harman et al., 2001), which incorporate symbolic conditions or partial inputs into the slicing process.

SDG-based approaches remain popular due to their comprehensive representation of inter-procedural control and data dependencies (Galindo et al., 2022). However, as noted earlier, these approaches assume access to complete codebases, and often fail when applied to incomplete or syntactically invalid code snippets. Condition-based slicing approaches too, model slicing under specific initial states, but still rely on traditional program representations and completeness assumptions, thus limiting their applicability to partial programs. Our proposed NS-Slicer (Chapter 4), is loosely related to SDG-based techniques, as it leverages an SDG-based slicing tool, JavaSlicer (Galindo et al., 2022), to enable the training process. However, unlike traditional methods, NS-Slicer does not explicitly build an SDG during inference. Instead, it directly predicts the static program slices from code and slicing criteria, making it applicable even when structural dependencies cannot fully be resolved.

Instrumentation-Based Analysis. Dynamic analysis relies on runtime instrumentation to collect execution traces and monitor program behaviors during execution. In particular, dynamic slicing, first introduced by (Korel and Laski, 1988) and later formalized by (Agrawal and Horgan, 1990), is useful for tasks such as debugging, fault localization, and test case minimization. Unlike static slicing, which must conservatively over-approximate all possible executions, dynamic slicing provides precise information about actual runtime dependencies.

Dynamic slicing has been extended in multiple ways in subsequent research. (Jhala and Majumdar, 2005) introduced path slicing, which improves its scalability by abstracting control flow paths and slicing based on symbolic path conditions. (Maras et al., 2011) adapted dynamic slicing to the client-side web application setting, where slices must account for user interactions, browser behaviors, and dynamic content changes. Similarly, (Tonella and Ricca, 2005; Ricca and Tonella, 2001, 2002) proposed techniques for web applications with dynamically generated code. (Binkley et al., 2014) proposed ORBS, a language-independent framework that abstracts slicing mechanisms away from language-specific syntax or semantics, making it applicable to a broader class of programming environments.

However, dynamic slicing techniques come with inherent limitations: they only reflect the execution path of the specific input and fail to generalize to unobserved behaviors. Furthermore, their reliance on complete code and all runtime observations makes them difficult to apply to partial or non-executable code fragments, motivating the need for generalizable, learning-based dynamic slicing (ND-Slicer, explored in Chapter 4).

Symbolic Analysis. Several SE tasks can be modeled as constraint satisfaction problems, including symbolic execution and automated test case generation (Cadar et al., 2008; Godefroid et al., 2012), type checking, program verification (Barnett et al., 2005; Griggio, 2009; D'Silva et al., 2008), security analysis (Backes et al., 2020), and optimization (Schkufza et al., 2016). In these tasks, program states and their transitions are encoded as logical formulas, and Satisfiability Modulo Theories (SMT) solvers such as Z3 (de Moura and Bjørner, 2008) and CVC4/5 (Liang et al., 2016; Barbosa et al., 2022) are used as back-end reasoning engines.

In symbolic execution, for instance, program paths are encoded as logical constraints, and variables are treated as symbolic inputs. SMT solvers are then used to aid a systematic exploration of all execution paths. Similarly, in model checking, specifications of safety properties (such as "no `null` pointer is ever dereferenced") are expressed in temporal logic,

and the analysis verifies whether error states are unreachable. However, symbolic reasoning is limited. Most notably, an increase in program complexity due to loops, recursion, or complex heap structures results in path explosion. Furthermore, SMT solvers do not inherently leverage any domain-specific contextual information, thereby limiting scalability.

To mitigate these issues, recent work has explored solver optimizations such as heuristic-based conflict analysis strategies (de Moura and Bjørner, 2008; Liang et al., 2016; Barbosa et al., 2022), eagerly factoring out assumptions (Lagniez and Biere, 2013), and proof-based trimming of input formulae (Belov et al., 2014). (Marques-Silva et al., 2021) proposed a tree search-based algorithm that constructs a solution through a combination of backtracking and learning new clauses from conflicts. Nevertheless, these strategies are still limited and can not scale, motivating the need for more scalable or learning-assisted alternatives.

2.2.2 Data-Driven Approaches to Program Analysis

Dependence Analysis. Several studies have explored probabilistic and observation-based alternatives to traditional compiler-based dependence analysis. Probabilistic PDGs (Baah et al., 2008) extend PDGs by annotating structural dependencies with statistical estimates derived from test executions, capturing the likelihood of data flow between node states. (Feng and Gupta, 2010) introduced the Error-Flow Graph, a Bayesian Network constructed from dynamic dependence graphs across multiple program runs. Building on this, (Yu et al., 2017) proposed the Bayesian Network-based Program Dependence Graph (BNPDG), which enables the inference of indirect or non-adjacent dependencies by modeling probabilistic relations across nodes. Moving beyond binary dependence, MOAD (Modeling Observation-based Approximate Dependency) (Lee et al., 2019) reformulates program dependency as a likelihood score, allowing for more flexible and context-sensitive interpretations. Notably, (Lee et al., 2020) presents a scalable approximate dependence analysis approach that merges lexical signals, partial execution observations, and static structure to estimate dependence

probabilities. These techniques indicate that traditional models alone are not sufficient and observational signals can significantly enhance the utility of dependence analysis.

SMT Optimization. A growing body of research explored the integration of *learning-based* components in conflict-driven clause learning (CDCL) SMT solvers. For example, (Selsam and Bjørner, 2019) propose guiding branch selection during solving by learning to predict conflict-causing variables. (Wang et al., 2021) focus on bottlenecks in clause deletion, learning which variable assignments are likely to be useful. More recently, SatFormer (Shi et al., 2023) attempts to directly predict the conflict-inducing clauses themselves. While these approaches show promise, they are typically trained on specific solver theories and behaviors, limiting their ability to generalize across diverse constraints encountered in real-world software analysis tasks. In contrast, our proposed framework, SafeMin (Chapter 6), leverages LLMs to macro-reason about inter-constraint relationships. Rather than predicting conflicts directly, SafeMin focuses on identifying *non-conflict-causing* constraints to safely reduce the size of an unsatisfiable formula. This approach builds on recent progress in automated reasoning using LLMs (Lewkowycz et al., 2022; Morishita et al., 2023; First et al., 2023), offering a way to incorporate domain-specific context, generalize across SMT theories, and improve the scalability of symbolic reasoning in software engineering.

Machine Learning for Program Analysis. Machine learning models, especially large language models based on Transformer (Vaswani et al., 2017) architecture can enable or enhance program analysis. First, these models take sequence of tokens as input, thus circumventing the need for completeness of the program. Moreover, pre-training tasks (Feng et al., 2020a; Guo et al., 2021) are useful to embed syntactic and semantic knowledge into the models. LExecutor (Souza and Pradel, 2023) present a learning-guided strategy for executing arbitrary code snippets by predicting the missing values that would otherwise cause the execution to fail. TRACED (Ding et al., 2024) pre-trains code language models on a

mixture of source code, inputs, and execution traces to embed both semantic and execution behaviors into the models. CodeExecutor (Liu et al., 2023) introduces a Transformer-based architecture trained to predict execution traces and the program states. These models incorporate execution-aware knowledge and hold promise for enabling predictive analyses of dynamic properties, relaxing the assumptions of syntactic completeness or full executability.

Other complementary efforts explore reasoning over program elements in partial code to infer the missing type information. These range from statistical learning (Phan et al., 2018), deep learning (Peng et al., 2022; Mir et al., 2022; Allamanis et al., 2020; Pradel et al., 2020; Ivanov. et al., 2021; Wang et al., 2025; Li et al., 2023), to large language models (Mai et al., 2024; Chen et al., 2024). (Huang et al., 2022) train a masked language model to recover fully qualified names (FQN) from incomplete code snippets. SOChecker (Chen et al., 2024) uses CodeLlama with vanilla prompting to complete incomplete code snippets, further employing symbolic execution for security vulnerability analysis in smart contracts.

Overall, these lines of work suggest that pre-trained and fine-tuned language models can be effective in formulating program analysis as predictive tasks, which is particularly useful when the code is incomplete or dynamic behavior is not directly observable. The research presented in this dissertation (Chapters 3–6) extend this paradigm across different analyses.

CHAPTER 3

REASONING ABOUT INTER-STATEMENT DEPENDENCIES IN LATENT SPACE: LEARNING TO BUILD PROGRAM DEPENDENCE GRAPHS

3.1 Overview

Dependence analysis (DA) is a fundamental technique in program analysis that examines the relationships between different program elements. A program dependence graph (PDG) (Ferrante et al., 1987) is a standard representation used to model such dependencies. They are useful in understanding program behaviors and serve as the basis for multiple applications, including optimization (Ferrante et al., 1987; Kuck et al., 1981), program slicing (Weiser, 1984), debugging (Podgurski and Clarke, 1990), testing (Zhu et al., 1997), and model checking (Visser et al., 2000). However, traditional tools like Joern (Joern, 2023) are *ineffective* when working with partial or incomplete code (*e.g.*, snippets from StackOverflow), and fail to build an accurate model of dependencies (see Section 3.2.1).[1]

These limitations motivate a key question:

Can we alternatively learn to approximate PDGs by reasoning about pairs of program statements in latent space, capturing the underlying data and control dependencies?

To this end, we present NeuralPDA, a neural network-based (partial) program dependence analysis framework that learns to derive dependencies directly from source code. Such a learning-based formulation is grounded in two central observations:

- **PDGs are repetitive**. In an empirical study on the repetitiveness, containment, and composability of PDGs in open-source projects, (Nguyen et al., 2016) reported that among

[1]The content presented in this chapter is based on the following publication (Yadavally et al., 2023):

> **Aashish Yadavally**, Tien N. Nguyen, Wenbo Wang, and Shaohua Wang. 2023. "(Partial) Program Dependence Learning". In *45th IEEE/ACM International Conference on Software Engineering*, ICSE 2023, Melbourne, Australia, May 14-20, 2023. IEEE, 2501–2513.

17.5M PDGs with 1.6B PDG subgraphs, 14.3% of the PDGs have all of their subgraphs repeated across different projects. Furthermore, in 15.6% of the PDGs, at least 90% of their subgraphs are likely to have appeared before in other projects.

- **Semantic relations are learnable in latent space.** In natural language processing, neural network-based dependency parsers have successfully shown to learn semantic relations between words in a sentence using large-scale text corpora (Chen and Manning, 2014).

Drawing from both ideas, we posit that PDG construction can benefit from data-driven generalization across software repositories, where statement within a given program are analogous to words in a sentence. Accordingly, we train NeuralPDA on annotated PDGs derived from large corpora of complete programs, enabling it to learn patterns of control and data flow across programs so as to *implicitly* reason about the inter-statement dependencies. NeuralPDA formulates this task as supervised link prediction between pairs of statements. It employs a hierarchical self-attention network that captures both intra-statement and inter-statement context, relaying local and global information within and across program statements. This representation-driven approach supports scalable, and language-agnostic (evaluated on Java and C/C++ programs) dependence analysis.

3.2 Motivation

3.2.1 Empirical Study

In this section, we present an empirical study to qualitatively probe the CFG/PDGs constructed by traditional program analysis tools like Joern (Joern, 2023) for partial programs.

For this purpose, we used a dataset comprising 99 incomplete code snippets from StackOverflow (S/O) that span 31 different vulnerability types that were later incorporated into 2,589 GitHub repositories (Verdi et al., 2022). Our goal is to assess how well Joern can construct PDGs for the code snippets, particularly focusing on control and data dependencies.

To this end, we: first, ran Joern on the 99 S/O code snippets (when needed, some instances were wrapped around dummy method signatures); (b) manually inspected the generated CFG/PDGs. We conducted a fine-grained analysis for all 99 instances, grouping the Joern outputs into four categories, listing the root causes for its failure in each category:

[1] **Incorrect Outputs**: In 47 cases, Joern either misses or incorrectly predicts multiple control flow, or data and control dependence edges.

[2] **Erroneous Instances**: In 30 cases, Joern produces error messages, typically of the form "`Could not find type member. type=XYZ, member=abc`". Here, `XYZ` is a type name and `abc` is the corresponding identifier (*i.e.*, the name of a variable or field) in the code snippet. Note that Joern can produce multiple such errors in a single code snippet. Moreover, running Joern on incomplete code snippets directly without wrapping them with dummy method signatures increases the number of such erroneous instances to 49.

[3] **Empty CFG, PDG, or both**: In 7 of the cases, Joern does not produce any nodes/edges for either the CFG, PDG, or both.

Broadly, the reasons for the cases in [1]–[3] can be summarized as follows:

- All data dependencies related to a parameter with unknown parameter type are ignored.
- For an unresolved external API class/method/field or an unresolved external data type, dependencies to/from the statements referencing it are ignored.
- All edges related to objects constructed with an unresolved/undeclared class are ignored.
- Inaccessible header files lead to undeclared variables, and all dependencies concerning the undeclared variables are missed.
- Missing variable declarations.
- Missing declared data types; unresolved data types from missing external libraries.

```
1   std::shared_ptr<FILE> pipe(popen(cmd, "r"), pclose);
2   if (!pipe) return "ERROR";
3   char buffer[128];
4   std::string result = "";
5   while (!feof(pipe.get()) ) {
6     if (fgets(buffer , 128, pipe.get()) != NULL)
7       result += buffer;
8   }
```

Figure 3.1: Motivating example for NeuralPDA: Code snippet from StackOverflow answer ID #478960, provided as a response to "*Execute a command within a C++ program and get output*" is prone to OS command injection (CWE-78, CWE-1019) (Verdi et al., 2022).

- Due to missing class hierarchies, Joern fails to recognize the inherited attributes and skips the corresponding data dependencies.
- Unresolved API references result in Joern skipping edges to/from these statements.
- Joern cannot handle templates, `typedef` declarations, *etc.*

[4] **Correct Outputs**: Joern produces correct CFG/PDGs in 15 of the cases, mainly because:

- Some of the instances are complete methods, classes, or files.
- In some, all the declarations are available, or the variables use primitive data types.
- Some code snippets possess no dependencies as it is just a group of structure definitions (*i.e.*, group of `struct` objects).

RQ. How reliable are the PDGs produced by Joern for incomplete code?

RA. Our findings highlight the inadequacies of traditional program analysis tools in effectively capturing inter-statement program dependencies in incomplete code.

3.2.2 Illustrating Example

Consider the code snippet in Figure 3.1 provided in response to a question on StackOverflow (S/O) seeking to execute a user-input command within a C++ program and retrieve its

output (answer ID #478960). This is vulnerable to OS command injection attacks, as the commands input by the user were not validated. For example, it is possible for an attacker to execute privilege-level commands without any errors or warnings. This vulnerability was reported in two Common Weakness Enumeration (CWE): CWE-78 and CWE-1019.

We can see that this code snippet is incomplete and: (a) variable `cmd` is undeclared; (b) object type `FILE` and the library `std` are undefined; (c) it contains references to functions from external libraries such as `pipe`, `popen`, `pclose`, `feof`, `fgets`, `pipe.get`, *etc.* The command injection fault occurs in line 1, where an unvalidated user input in `cmd` flows directly into the external API `popen` without sanitization. To detect this vulnerability, an effective taint analysis must trace the data flow of `cmd` from its source through `popen` and the subsequent program. However, since `popen` and other associated call sites like `feof` and `fgets` are treated as external or ambiguous APIs, the resulting PDG from Joern for such an analysis is *under-representative*, failing to capture crucial inter-statement dependencies. Therefore, Joern is insufficient for downstream analyses that rely on complete dependence information.

3.3 NeuralPDA

3.3.1 Observations and Key Ideas

Aside from detecting vulnerabilities in incomplete code, such partial program dependence analysis is beneficial to automated code completion tools as well. For example, in Figure 3.1, assume that a developer editing the code invokes the code completion tool at line 5, *i.e.*, `! feof (pipe._`. With a complete model of program dependencies, the code completion tool could suggest `pipe.get` due to the knowledge of a control dependency between the statement `pipe(popen(...))` on line 1 and the potentially suggested candidate `pipe.get`.

Observation 3.1 (Partial Program Dependence Analysis)**.** *In software engineering tasks where completely analyzable code is not available, and a low level of errors and imprecision in deriving the dependencies is tolerable, partial program dependence analysis is desirable.*

```
1   #include <cstdio>
2   #include <iostream>
3   #include <memory>
4   #include <stdexcept>
5   #include <string>
6   #include <array>
7
8   std::string exec(const char* cmd) {
9     std::array<char, 128> buffer;
10    std::string result;
11    std::unique_ptr<FILE, decltype(&pclose)> pipe(popen(cmd, "r"),
      pclose);
12    if (!pipe) {
13      throw std::runtime_error("popen()␣failed!");
14    }
15    while (fgets(buffer.data(), buffer.size(), pipe.get()) != nullptr) {
16      result += buffer.data();
17    }
18    return result;
19  }
```

Figure 3.2: Motivating example for reasoning about inter-statement dependencies in latent space: Complete code with same POSIX elements as in Figure 3.1.

Now, consider the complete code example in Figure 3.2 from S/O post 10702464 (Fisherman, 2015) alongside the incomplete snippet in Figure 3.1. While there are some differences in a few details (constants, error messages, *etc.*), the presence of many similar statements indicates that the data and control dependencies are comparable: line 1 and line 11, line 2 and lines 12–14, lines 3–4 and lines 9–10, lines 5–6 and line 15, line 7 and line 16, respectively. Thus, the program dependencies between the statements in the incomplete code snippet in Figure 3.1 can be learned from those extracted for the complete code example in Figure 3.2.

Observation 3.2 (Learn to Analyze Program Dependencies)**.** *Analyses of inter-statement program dependencies in a given incomplete code snippet can be guided by patterns learned from complete code extracted from existing code corpora.*

Following Observations 3.1—3.2, we design NeuralPDA with the following key ideas:

Key Idea 3.1. *Neural Network-Based Approach to Partial Program Dependence Analysis*

Instead of deterministically producing the program dependencies in a best-effort manner, following Observation 2, we design a neural language model (NLM) to learn to analyze the program dependencies among the statements in the given source code. By leveraging the program dependencies extracted by program analysis techniques (Joern, 2023) for the complete code in open-source projects (*e.g.*, GitHub) in the training process, the NLM can derive the inter-statement program dependencies for a given code snippet.

Key Idea 3.2. *Program Dependence Decoding from Dense Statement Representations*

We draw inspiration from neural network-based dependency parsing approaches (Chen and Manning, 2014) in NLP which effectively learn semantic relations between words in a sentence by modeling dependencies between their latent representations in an embedding space. In a similar vein, we design NeuralPDA to learn latent representations of program statements, enabling the prediction of program dependence relations between them.

Key Idea 3.3. *Enhancing Statement Representations with Intra-Statement and Inter-Statement Context Learning*

The quality of the statement representations determines NeuralPDA's ability to predict program dependence relations accurately. To this end, we rely on two types of contextualization. **Intra-statement context** captures the program entities represented by code tokens within an individual statement, which helps the model derive the control/data dependencies across statements. For example, in Figure 3.1, the variable declaration on line 3 contains the token (*i.e.*, variable) `buffer`, which is also referenced in the assignment on line 7. Intra-statement contextualization makes information about the local context within the individual statements on lines 3 and 7 available globally, helping the model recognize the declaration and reference of the same variable. This, in turn, enables the model to recognize the def-use data dependency on account of the variable `buffer` between the two statements.

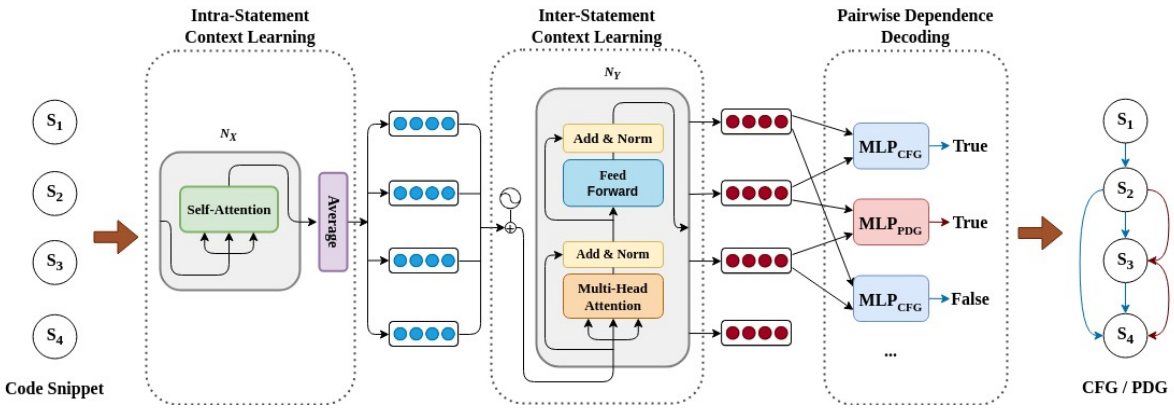


Figure 3.3: Model architecture for NeuralPDA

Inter-statement contextualization, on the other hand, helps NeuralPDA model the effect of surrounding statements on the two under consideration. For example, in Figure 3.1, knowing that the `if` statement on line 6 is nested within the `while` loop on lines 5–8 helps the model recognize that the execution of the assignment on line 7 is conditioned not only by the predicate on line 6, but also the loop condition on line 5. This broader context is crucial for modeling control dependencies that arise due to nested control structures.

3.3.2 Approach Overview

Given a code snippet covering the statements s_1, s_2,.., s_N, where the statement s_i contains the tokens $t_1^{(i)}$, $t_2^{(i)}$.., $t_M^{(i)}$, as illustrated in Figure 3.3, NeuralPDA is realized via a hierarchical self-attention network (SAN)-based model architecture. Each component of NeuralPDA is designed to capture different aspects of contextualization, thereby enabling the learning of all program dependencies between statement pairs $\langle s_i, s_j \rangle$ (where $1 \leq i, j \leq N$):

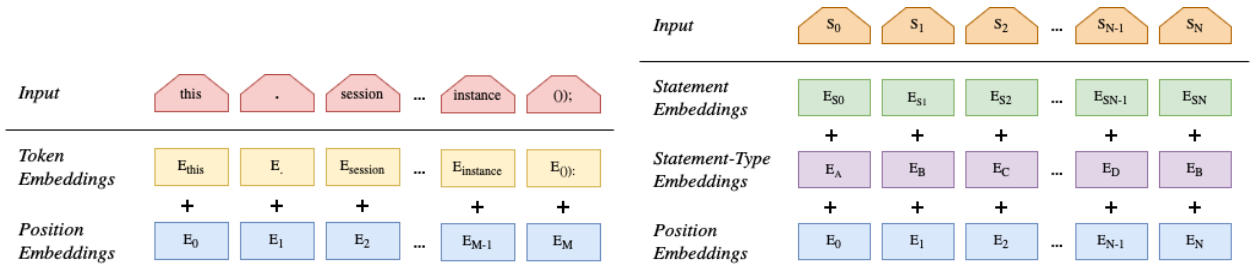
- *Intra-Statement Context Learning.* The syntactic and semantic knowledge of the code tokens within a single statement must be made available globally to other statements in a program to learn the inter-statement program dependencies effectively. For the tokens $t_1^{(i)}$, $t_2^{(i)}$.., $t_M^{(i)}$ in the statement s_i, the goal of this component is to map them into an embedding space $\mathbb{R}^d$, and generate a context-dependent representation $u_i \in \mathbb{R}^d$ for statement s_i.

We enable this via a *1-Layer* (*i.e.*, N_X=1) Self Attention Network (1L-SAN). The self-attention layer in an 1L-SAN inputs $x_1, x_2, ..., x_n \in \mathbb{R}^d$, performs self-attention once by projecting the inputs from all attention heads $\in \mathbb{R}^{d_h}$ into the head dimension space d_h via linear transformations, and generate outputs $y_1, y_2, ..., y_n \in \mathbb{R}^d$ which are linear combinations of the concatenated attention head values. We use one attention head (*i.e.*, h=1) for the self-attention layer in 1L-SAN, and the size of the input representations, *i.e.*, d is set to 512. Our experiments revealed only a marginal performance gain by expanding the 1L-SAN to a 2L-SAN, which also came with high computational overhead. Besides, increasing the number of attention heads did not help either performance or interpretability. A more detailed analysis on hyperparameters and subsequent trade-offs is left to future work.

Input Representations. For a program comprising N statements $s_1, s_2, ..., s_N$, NEURALPDA takes as input a concatenation of N sequences of M tokens each, $\langle t_1^{(1)}, t_2^{(1)}.., t_M^{(1)} \rangle$, ..., $\langle t_1^{(N)}, t_2^{(N)}.., t_M^{(N)} \rangle$. Next, each token sequence $\langle t_1^{(i)}, t_2^{(i)}.., t_M^{(i)} \rangle$ is input to the 1L-SAN for intra-statement contextualization. Previous works (Radford et al., 2019; Liu et al., 2019) have demonstrated the advantages of a byte-level Byte-Pair Encoding (BPE) scheme for tokenization. Accordingly, we train a byte-level BPE tokenizer for converting a given statement into a sequence of tokens. Here, M is the maximum number of tokens allowed in a statement. For statements with token sequences having $<M$ tokens, a special **[PAD]** token is appended. In contrast, token sequences having $>M$ tokens are truncated to M tokens.

Token Embeddings. For all words in the vocabulary V, we leverage a learnable embedding to learn and store their representations (*i.e.*, $\mathbb{R}^{|V| \times d}$). Using this as a lookup table, token embeddings are retrieved for all the tokens generated by the tokenizer for a given statement.

Token Position Embeddings Attention mechanism in the self-attention layer is invariant to position information. However, this knowledge is key to understanding the sequential



(a) Intra-Statement Context Learning (b) Inter-Statement Context Learning

Figure 3.4: Input representations for: (a) tokens in a statement are the sums of token embeddings, and token position embeddings; (b) statements in a snippet are the sums of statement embeddings, statement-type embeddings, and statement position embeddings.

nature of code tokens in a statement. We enable this via a learnable position encoding scheme, where a vector $\in \mathbb{R}^d$ unique to each position is learned during the training process.

Statement (Output) Representations. Input representations to the 1L-SAN corresponding to the tokens in a given statement are taken as the sums of the *token embeddings*, and their *position embeddings* (as in Figure 3.4a). The 1L-SAN yields intra-statement contextualized token representations as its output, which are then averaged to retrieve the statement representation. Note that the token representations corresponding to the **[PAD]** tokens are not considered for averaging. Such statement representations u_i ($\in \mathbb{R}^d$) for all the statements $s_i \in s_1 ... s_N$ are then passed on for inter-statement contextualization.

- *Inter-Statement Context Learning.* The knowledge of surrounding statements in the context of a given statement helps NEURALPDA model the dependencies between them better. Given the statement representations $u_i \in \mathbb{R}^d$ for the statements s_1, s_2,.., s_N in a code snippet that are local context-aware, the goal of this component is to learn their latent vector representations $v_i \in \mathbb{R}^d$ that model the dependencies between the statements.

  We enable this via a multi-layer Transformer encoder based on the work by (Vaswani et al., 2017) (refer to Chapter 2 for more details on Transformers' model architecture). As shown in Figure 3.3, we denote the number of layers in the Transformer encoder by N_Y, which we

set to 6. We employ 4 attention heads, *i.e.*, h=4 to increase parallelization (since d_h=$\frac{d}{h}$, *i.e.*, d_h=128) and learn different aspects of the syntactic and semantic structure in the statements, while still being interpretable. We set the feed-forward module size to be 4 times that of the size of the input representations d, *i.e.*, 2048.

Statement Input Representations. For all the statements s_i in a program, statement embeddings (*i.e.*, $u_i \in \mathbb{R}^d$) are obtained from the 1L-SAN in the intra-statement contextualization phase. N is the maximum number of statements in a method in the dataset. If a given program has less than N statements, zero vectors ($\in \mathbb{R}^d$) are padded to the inputs.

Statement Position Embeddings. To make our model understand the sequential nature of program statements, as in the intra-statement contextualization phase, we adopt a learnable position encoding scheme to learn unique vectors ($\in \mathbb{R}^d$) for all statement positions.

Statement Types. Most neural network-based dependency parsers leverage parts-of-speech (POS) tags for the words in a sentence for better dependency learning. Thus inspired, we chose to associate with each statement a label indicating the *statement type*, which is essentially the type of the AST node rooted at the sub-AST for the statement. We extract labels such as `METHOD`, `CONTROL_STRUCTURE`, `BLOCK`, etc., which helps augment such syntactic information. We learn unique vectors ($\in \mathbb{R}^d$) for each of the statement types.

Statement (Output) Representations. As shown in Figure 3.4b, the input representations for program statements are taken as the sums of the *statement embeddings*, *statement-type embeddings*, and the *statement position embeddings*. These are further input to Transformer encoder to obtain contextualized statement representations $v_i \in \mathbb{R}^d$ for all $s_i \in s_1 ... s_N$, thus modeling syntactic and semantic knowledge from both within and across the statements.

- *Pairwise Dependence Decoding.* From the sequence of statement representations $v_i \in \mathbb{R}^d$ corresponding to all program statements output by the Transformer encoder in the inter-statement contextualization phase, pairs such as $\langle v_i, v_j \rangle$ (1$\leq i, j \leq$ N) are taken to detect

the presence of CFG/PDG edges between two statements s_i and s_j. We leverage 2-layered multi-layer perceptron networks (each for detecting the CFG and PDG edges, *i.e.*, MLP_{CFG} and MLP_{PDG}, respectively) in this phase, which are scored as follows:

$$score_{\text{rel}}(i,j) = MLP_{\text{rel}}(v_i \circ v_j \circ (v_i * v_j) \circ |v_i - v_j|) \tag{3.1}$$

where $\circ$, $*$ and $|.|$ correspond to concatenation, element-wise product, and absolute element-wise difference operations, respectively; and *rel* represents either the control flow or program dependence relations. Attaining a $score_{\text{rel}}(i,j) > 0.5$ represents the detection of the corresponding CFG/PDG edge from s_i to s_j. The combination of all edges extracted via such an arc-factored approach is realized as the CFG/PDG for the given program.

3.3.3 Training Process

Training NeuralPDA requires knowledge of the ground-truth CFG and PDG edges between the program statements. Thus, it can only be trained on complete programs (at a minimum, which are at a method-level granularity) to be able to leverage program analysis tools to extract them. The CFG and PDG edge information can then be utilized to compute the training objective loss $\mathcal{L}$ for our model as follows:

$$\mathcal{L} = \mathcal{L}_{\text{CFG}} + \mathcal{L}_{\text{PDG}} \tag{3.2}$$

Here, $\mathcal{L}_{\text{CFG}}$ and $\mathcal{L}_{\text{PDG}}$ are the losses associated with decoding CFG and PDG edges, respectively. Each of these components is computed as the sum of binary cross-entropy (BCE) losses over all possible pairs of statements, based on the presence or absence of edges in the corresponding graph. For a program $s_1, s_2, ..., s_N$, let $y_{ij}^{CFG} \in \{0,1\}$ and $y_{ij}^{PDG} \in \{0,1\}$ denote the ground-truth labels indicating whether a directed edge exists from statement s_i to statement s_j in the CFG and PDG, respectively. Let $\hat{y}_{ij}^{CFG}$ and $\hat{y}_{ij}^{PDG} \in (0,1)$ denote the model's predicted probabilities for those edges. Then the losses are computed as:

$$\mathcal{L}_{CFG} = \sum_{i=1}^{n}\sum_{j=1}^{n} \mathbf{1}_{ij}^{\text{valid}} \cdot \text{BCE}(y_{ij}^{rel}, \hat{y}_{ij}^{rel}) \tag{3.3}$$

where *rel* represents either the CFG or PDG, and the binary cross-entropy loss is:

$$\text{BCE}(y, \hat{y}) = -y \cdot \log(\hat{y}) - (1 - y) \cdot \log(1 - \hat{y}) \tag{3.4}$$

and $1_{ij}^{\text{valid}} \in \{0, 1\}$ is an indicator function that masks out padded statement pairs (*i.e.*, it is 0 if s_i or s_j is padding), thus excluding such non-existent edges from loss computations.

The model parameters Θ in NeuralPDA which are optimized to minimize $\mathcal{L}$ include: (a) learnable embeddings for tokens, token positions, statement types, and statement positions, (b) the attention mechanism and feed-forward neural network in Transformer encoder, (c) the multi-layer perceptrons for decoding control flow and program dependence edges (*i.e.*, MLP_{CFG}, and MLP_{PDG}). In total, NeuralPDA comprises ≈39M trainable parameters.

3.3.4 Inference for Dependence Discovery

Despite being trained on only complete code, one can leverage NeuralPDA to extract control flow and program dependence edges from both complete and partial code. However, the following are some important points for consideration:

- *Statement Types*: To extract the syntactic information encoded in *statement types*, one would need the program's AST. In Java, for example, this can be retrieved even if the code is incomplete using tools such as PPA (Dagenais and Hendren, 2008b). However, this is not possible for all programming languages. In such cases, NeuralPDA can be trained without statement types, *i.e.*, by computing the input representation for statements in a program as just the sums of the statement embeddings and their position embeddings. In Section 3.4.1, we demonstrate the practicality of such an alternative.

- *Programs with ≤N statements*: Applying a trained NeuralPDA model to programs in which the number of statements is less than the *maximum statements* allowed in the model is straightforward. In such cases, NeuralPDA predicts the CFG/PDG edges from one statement to another by contextualizing over all the other statements in the program.

Table 3.1: Effectiveness of NeuralPDA on Complete Methods in Java and C/C++

P/L	Graph	Accuracy	Precision	Recall	F1-Score
Java	*CFG*	99.79	98.31	98.58	98.44
	PDG	98.87	89.89	87.53	88.70
	Overall	99.33	94.75	93.83	**94.29**
C/C++	*CFG*	99.54	96.86	96.92	96.89
	PDG	98.63	86.00	88.69	87.33
	Overall	99.08	92.25	93.49	**92.86**

- *Programs with >N statements*: For programs with a total number of statements greater than that allowed in the trained NeuralPDA model, we present the following strategies: (a) train a model with a higher value of N, (b) chunk the program into N-statement code fragments, predict CFG/PDG edges for each of these independently, and finally combine their corresponding CFG/PDG predictions. For example, if a trained model allows a maximum of 16 statements, to predict for a program with 46 statements, one can break it down into code fragments with 16, 16, and 14 statements, respectively. A potential downside to strategy (b), however, is that a statement in a fragment will be contextualized only over the other statements in that fragment, and the control flow and program dependencies across fragments will not be captured. Increasing N could address this issue, albeit with additional computational overhead.

3.4 Empirical Evaluation

3.4.1 Effectiveness on Complete Programs

Data Collection, Procedure, and Evaluation Metrics. To evaluate the effectiveness of NeuralPDA, we first collected and filtered the following programs:

- *Java*. GitHub Java Corpus (Allamanis and Sutton, 2013) is a large-scale collection of Java code containing 10,968 training and 3,817 testing projects. Within each, we retained only

the source code files. We then collapsed the directory structure, randomly selecting 57,600 and 14,400 Java files each from projects in the training and testing sets, respectively.

- *C/C++*. Wang *et al.* (Wang et al., 2023) expanded *Big-Vul* (Fan et al., 2020a), a large C/C++ vulnerability dataset from the Common Vulnerabilities and Exposure database, to span across 2000–2021 and extracted approximately 50K methods from it.

Next, we leveraged the Joern program analysis tool (Joern, 2023) to extract the AST edges (to retrieve syntactic type information), CFG edges, and PDG edges for the files in the Java dataset, and the methods in the C/C++ dataset. We also filtered out the Java and C/C++ methods without any CFG or PDG edges. Finally, we split both Java and C/C++ datasets into: (a) 40,000 Java methods for training, and 4,000 each for validation and testing; (b) 12,767 C/C++ methods for training, and 1,500 each for validation and testing. With an initial learning rate of $5\text{x}10^{-4}$, and by setting the maximum number of tokens in a statement (*i.e.*, M) to 32, and the maximum number of statements in a program (*i.e.*, N) to 8, we trained NeuralPDA on both datasets. The training process was carried out on a machine with an Nvidia Quadro P4000 GPU. The training time per epoch was approximately 4.5 hours and 1.5 hours for Java and C/C++ datasets, respectively.

We model the program dependence decoding step in NeuralPDA as a classification problem. Thus, we adopt the standard evaluation metrics:

$$\begin{aligned} \text{Accuracy} &= \frac{\text{TP} + \text{TN}}{\text{TP} + \text{FP} + \text{FN} + \text{TN}}, \\ \text{Recall} &= \frac{\text{TP}}{\text{TP} + \text{FN}}, \\ \text{Precision} &= \frac{\text{TP}}{\text{TP} + \text{FP}}, \text{and} \\ \text{F1-Score} &= \frac{2 \times \text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}}. \end{aligned} \tag{3.5}$$

Here, TP: True Positives, FP: False Positives, FN: False Negatives, and TN: True Negatives.

Table 3.2: NeuralPDA's performance for different types of CFG and PDG edges

Graph	Edge Type	%C	
		Java	*C/C++*
CFG	*sequential*	99.54	98.91
	if-else	95.52	**
PDG	*data dependence*	82.78	88.21
	control dependence	96.33	94.65

Empirical Results. In Table 3.1, we show the performance of NeuralPDA on complete methods from both Java and C/C++ datasets. Clearly, our model produces competitive results. In particular, NeuralPDA predicts the control flow edges with F1-scores of 98.44% and 96.66%, and program dependence edges with F1-scores of 88.70% and 86.66%. Overall, it approximates the combination, *i.e.*, CFG + PDG generated by the program analysis tool for the complete methods with F1-scores of 94.29% and 92.46%, respectively.

RQ. How accurate is NeuralPDA in generating CFG/PDGs for complete programs?

RA. NeuralPDA approximates the combined CFG+PDG for Java and C/C++ programs with F1-scores of 94.29% and 92.86%, respectively.

Qualitative Evaluation. To better understand the model performance, we analyzed how precisely NeuralPDA can predict different types of control flow and program dependence edges. We enable this by retrieving two kinds of control flow edges: *sequential* and *if-else*. The latter refers to all edges where the control flows from an `if` to an `else`-statement. We also retrieve two kinds of program dependence edges: *data* and *control dependence*.

In Table 3.2, we report the results for our qualitative evaluation, where $\%C$ corresponds to the percentage of correctly predicted edge relations. Overall, NeuralPDA predicts the *sequential*, *if-else*, *data dependence*, and *control dependence* edges with an accuracy of 99.54%, 95.52%, 82.78%, and 96.33%, respectively for Java methods. The relatively lower accuracy in predicting data dependence edges can possibly be attributed to a class-imbalance problem

Table 3.3: Effectiveness of NeuralPDA for Partial Programs in Java and C/C++

N	**Java**			**C/C++**		
	CFG	*PDG*	*Overall*	*CFG*	*PDG*	*Overall*
3	99.70	93.24	97.17	98.39	91.10	96.01
4	99.41	92.51	96.61	98.33	90.18	95.28
5	99.26	91.23	95.96	97.91	89.10	94.37
6	99.04	90.36	95.40	97.14	87.91	93.31
7	98.75	89.45	94.82	96.69	86.45	92.39
8	98.44	88.70	**94.29**	96.66	86.66	**92.46**

since the percentage of the statement pairs having data dependencies is much smaller than that of pairs without them. This can be boosted by: (a) expanding the dataset to contain more data dependence edges, (b) incorporating class-imbalance mitigation strategies.

In contrast, for C/C++ methods, such accuracies are 98.91% for *sequential*, 88.21% for *data dependence*, and 94.65% for *control dependence* edges within the predicted CFG/PDGs. Note that none of the C/C++ methods in the test set have *if-else* control flow edges, as indicated by "**". Compared to Java methods, we observe a notable improvement in the prediction of data dependence edges (*i.e.* from 82.78% → 88.21%), despite the training set for the former being approximately 3.2× larger. This observation further confirms that NeuralPDA's effectiveness in predicting different types of edges is directly dependent on the distribution of those edge relations in the training data.

RQ. How accurately does NeuralPDA predict specific program dependence relations?

RA. NeuralPDA demonstrates high accuracy across both CFG and PDG edge types. For Java, it achieves 99.54% and 95.52% F1-scores for sequential and *if-else* edges, and 82.78% and 96.33% for data and control dependencies. For C/C++, it achieves 98.91% F1-score for sequential edges, 88.21% and 94.65% for data and control dependencies.

3.4.2 Effectiveness on Partial Programs

To show NeuralPDA's flexibility in handling both complete and partial programs, we evaluated our model's performance on consecutive lines of code in a method, starting from the first statement. We refer to this as the Top-N experimental setting. In this experiment setting with N ranging from 3 to 8, if a method has $n \leq 8$ statements, it is treated as a partial method for N from 3→n, and as a complete method for N from n→8. However, in cases where a method has $n > 8$ statements, it is treated as a partial method for all values of N. As seen in Table 3.3, for Java, the overall F1-score decreases gracefully as the size of partial code fragment increases because more edges need to be predicted. Also, F1-score decreases only <3% as N increases from 3 → 8. For C/C++, such a decrease is <4%. Thus, NeuralPDA is effective in handling both partial Java and C/C++ programs.

RQ. How accurate is NeuralPDA in generating CFG/PDGs for partial programs?

RA. NeuralPDA approximates the combined CFG+PDG for Java and C/C++ programs with F1-scores of 94.29%–97.17% and 92.86%–96.01%, respectively.

3.4.3 Ablation Study

Procedure. We performed an ablation over different components in NeuralPDA to better understand their relative importance by removing one of the components. We refer to this setting as Leave-One-Out (LOO)-NeuralPDA, which includes:

- B_1. In Section 3.3.2, we hypothesized that knowledge of statement positions is crucial to learn the sequential nature of source code. We created this baseline to evaluate their importance. Here, the input representations for statements in a code snippet are computed as just the sums of statement embeddings and the statement-type embeddings.

- B_2. To better learn the syntactic nature of a statement, we hypothesized the inclusion of statement types. In this baseline, we assess their importance by computing the output

Table 3.4: Ablation over NeuralPDA model components on Java programs

Baselines	Accuracy	Precision	Recall	F-Score
B_1, *w/o* Statement PE	97.10	82.26	64.61	72.37
B_2, *w/o* Statement Types	99.15	93.16	92.39	92.78
B_3, *w/o* Intra-Statement Context	97.77	89.49	70.29	78.74
B_4, *w/o* Inter-Statement Context	98.53	89.89	84.56	87.14
NeuralPDA	99.33	94.75	93.83	**94.29**

Table 3.5: Qualitative evaluation of LOO-NeuralPDA on Java programs

Baselines	CFG		PDG	
	sequential	*if-else*	*data*	*control*
B_1, *w/o* Statement PE	55.30	68.66	58.39	63.63
B_2, *w/o* Statement Types	98.31	92.54	81.63	95.78
B_3, *w/o* Intra-Statement Context	93.13	31.34	43.60	47.50
B_4, *w/o* Inter-Statement Context	99.02	79.10	84.19	95.41

statement representations for statements in a snippet as just the sums of the statement embeddings and their position embeddings.

- B_3. We created this baseline to better evaluate the importance of *intra-statement context learning*. Instead of using a self-attention network (Figure 3.3) for statement representations, we use Word2Vec (Mikolov et al., 2013) to compute word embeddings for each token in the statement and consider their average as the statement representation.

- B_4. We created this baseline to better evaluate the importance of *inter-statement context learning*. The statement representations generated by this phase (Figure 3.3) for all the statements in a code snippet are directly input to the pairwise-decoder MLPs, instead of being passed to the Transformer encoder for inter-statement contextualization.

We adopt the same evaluation metrics as in Section 3.4.1 to evaluate LOO-NeuralPDA.

Empirical Results. As shown in Table 3.4, our design choices help achieve higher overall accuracy than all LOO-NeuralPDA baselines. In particular, our model improves over the

baselines B_1—B_4 by over 30.29%, 1.63%, 19.75% and 8.21%, respectively. Furthermore, we can be see that NeuralPDA w/o statement types achieves the performance closest to that of NeuralPDA. This is particularly useful when leveraging it to derive the CFG and PDG edges for code snippets in which *extracting such syntactic type information is not possible*.

We also extend the qualitative analysis (as in Table 3.2) to LOO-NeuralPDA baselines to investigate how absence of different model components contributes to the imprecision in predicting control flow and program dependence edges. From Table 3.5, we can see that:

- In the absence of the *position information* of statements, the ability of NeuralPDA to predict all CFG and PDG edges drops significantly, which is justified. For example, without such knowledge, the model might not know that the control should always flow from an `if`-statement to its corresponding `else`-statement, or that the direction of the data dependence should be from a variable definition to its reference in a `def-use` chain, *etc.*
- Statement types like *call*, *return*, *etc.* help the model capture the control dependencies better. Thus, in the absence of such syntactic information, the drop in F1-score in predicting the control flow and control-dependent edges is expected.
- In Key Idea 3 (see Section 3.3.1), we establish the importance of intra-statement context learning for identifying data dependencies between statements. The 89.86% drop in F1-score in predicting data dependencies without such contextualization supports that claim.

RQ. How do different components in NeuralPDA contribute to its performance?

RA. All model components in NeuralPDA directly contribute to its ability in predicting different types of control flow and program dependence relations.

3.4.4 Usefulness in Detecting Vulnerabilities in Complete Programs

Deep learning (DL)-based approaches that utilize PDGs for vulnerability detection (VD) can tolerate a low level of errors in program dependencies, wherein the imprecision acts as

noise and aids in regularizing the model. VulCNN (Wu et al., 2022) is a state-of-the-art method-level VD tool that takes as input a program semantics-capturing image extracted from PDGs. In this experiment, we seek to determine how the PDGs predicted by NeuralPDA (say, PDG^{*}) for *complete* methods affect the performance of downstream tasks.

Here, we use VulCNN with both PDG^{*} and the PDG derived from Joern (say, $PDG^{\#}$) for these methods as input, to see how closely PDG^{*} mimics $PDG^{\#}$ and approximates the performance of the VD model. Mathematically, we can formulate this task as follows:

$$0 < VD\{PDG^{*}\} \leq VD\{PDG^{\#}\} \tag{3.6}$$

where $VD\{.\}$ indicates the performance of the automated VD model. Notably, if $VD\{PDG^{*}\} \lesssim VD\{PDG^{\#}\}$, we can establish the efficacy of NeuralPDA for downstream SE tasks.

Data Collection. For this study, we used the VD dataset from prior work comprising complete Java methods collected from eight large open-source Java projects (Li et al., 2019). First, we filtered by projects, dedicating *avro*, *camel*, *hbase*, *hive*, *lucene-solr*, and *pig* for training purpose; *flink* and *cloudstack* for validation and testing, respectively. We retain all methods that have between 3 and 10 statements in each of these splits. Finally, we randomly selected an equal number of data samples from the vulnerable and non-vulnerable subsets in both splits, obtaining about 8K methods for training and 1K each for testing and validation.

Procedure. For all Java methods in the VD dataset, we extracted program dependencies (*i.e.*, data and control dependence edges) with Joern program analysis tool (Joern, 2023). Next, using NeuralPDA trained on the Java dataset in Section 3.4.1 (Table 3.1), we generated PDGs (*i.e.*, PDG^{*}) for all complete methods in the VD dataset. We then passed $PDG^{\#}$ and PDG^{*} to VulCNN for detecting vulnerabilities. VulCNN adopts centrality analysis to transform PDGs into program semantics-capturing images. Accordingly, we generate two image datasets corresponding to $PDG^{\#}$ and PDG^{*}, each of which are input to a convolutional neural network (CNN) for detecting the presence of vulnerabilities in complete code.

Table 3.6: Comparison of PDG$^{\#}$ (generated by Joern) and PDG* (predicted by NeuralPDA) for method-level vulnerability detection with VulCNN (Wu et al., 2022).

Methodology	TPR	TNR	F1-Score
PDG$^{\#}$ + VulCNN	74.03	74.03	74.01
PDG* + VulCNN	73.27	73.27	73.26

Evaluation Metrics. We adopt the same metrics used by Wu *et al.* (Wu et al., 2022) to evaluate VulCNN, *i.e.*, *true positive rate* (TPR) (also referred to as *Recall*), *true negative rate* (TNR), and *F1-score*. Here, the positive label corresponds to the presence of a vulnerability in the method under study, while the negative label is given to a non-vulnerable method.

Empirical Results. In Table 3.6, we report the performance of VulCNN from both experiment settings, *i.e.*, by using PDG$^{\#}$ and PDG*. We can observe that NeuralPDA predicts PDG* with an overall F1-score of 91.13%. This further establishes the *generalizability of* NeuralPDA, more so, because the Java dataset that NeuralPDA was trained on, and the VD dataset comprising Java methods for which PDG*s were derived come from two *entirely distinct code corpora*. Moreover, VulCNN achieves an F1-score of 73.26% using PDG*, which is a close approximate of the F1-score of VulCNN using PDG$^{\#}$, 74.01%.

RQ. How useful is NeuralPDA in approximating traditional tools in the extrinsic task of detecting vulnerabilities in complete programs?

RA. The PDGs predicted by NeuralPDA approximates the performance of those generated by Joern for detecting vulnerabilities in complete programs by 98.98%.

3.4.5 Usefulness in Detecting Vulnerabilities in Partial Programs

Prior work manually inspected and reported 99 commonly used C/C++ code snippets from StackOverflow (S/O) answers as vulnerable (Verdi et al., 2022). However, due to their incomplete nature, these can not be analyzed automatically for vulnerabilities. In this

experiment, we design an "in-the-wild" evaluation by: (a) first, leveraging NeuralPDA trained on complete C/C++ methods in Section 3.4.1 to predict PDGs for the incomplete C/C++ code snippets from S/O; (b) next, training VulCNN (Wu et al., 2022) to detect vulnerabilities in C/C++ code; (c) finally, making use of the predicted PDGs and trained VulCNN to check for vulnerabilities in the S/O code snippets. Such a design is useful for developers to detect early the vulnerabilities in the code fragments on online forums.

Data Collection. (1) In Section 3.4.1, we describe the expanded C/C++ dataset comprising about 50K methods, 26.3% of which are vulnerable (Wang et al., 2023). We split this according to a 80%-10%-10% ratio to train VulCNN. (2) Of the 99 vulnerable code fragments collected by Verdi *et al.*, VulCNN fails to process 33 of them. We filter these out, and leverage NeuralPDA to predict PDGs for the remaining 66 code snippets.

Procedure. First, for all C/C++ methods in the VD dataset, we extract PDGs with Joern (Joern, 2023). We then train VulCNN on this dataset, achieving an F1-score of 65.66% on the test set. Next, we leveraged the trained NeuralPDA on C/C++ dataset to predict PDGs for the 66 code snippets from StackOverflow. Finally, we generate corresponding program semantics-capturing images to input to the trained VulCNN model and discover the number of code snippets in which it can correctly identify vulnerabilities.

Empirical Results. We observed that VulCNN correctly identifies 14 out of the 66 code snippets as vulnerable. This is encouraging, more so because the C/C++ data that was used to train VulCNN (F-score=65.66%) does not cover all the vulnerabilities prevalent in the manually inspected S/O code snippets. The 52 mis-detected code snippets could be due to the inaccuracy of VulCNN. Besides, as in Section 3.4.4, since the C/C++ dataset that was used to train NeuralPDA and the StackOverflow C/C++ code snippets do not possess any similarities, we can establish our tool's generalizability.

```
1   private boolean isValidUntil(Until annotation) {
2     if (annotation != null) {
3       double annotationVersion = annotation.value();
4       if (annotationVersion <= version) {
5         return false; }
6     }
7     return true;
8   }
```

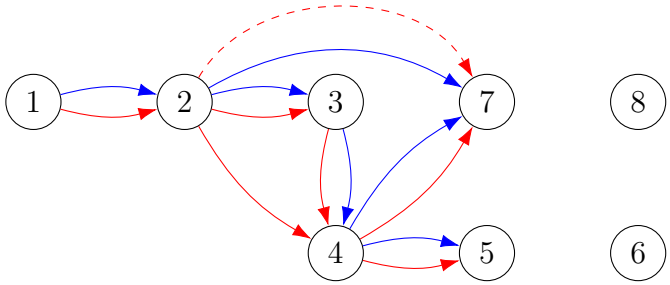


Figure 3.5: Case study: (*top*) a Java program, and (*bottom*) its CFG+PDG. Here, → and → denote CFG and PDG edges. Dashed arrow indicate edges missed by NeuralPDA.

RQ. How useful is NeuralPDA in the extrinsic task of discovering vulnerabilites in StackOverflow code snippets?

RA. The PDGs predicted by NeuralPDA helps VulCNN (Wu et al., 2022) discover real-world vulnerabilities in 14 code fragments from StackOverflow.

3.4.6 What does NeuralPDA Learn? A Case Study

Consider the source code example illustrated in Figure 3.5 (*top*), which was retrieved from the test-split in the Java dataset (see Section 3.4.1). In Figure 3.5 (*bottom*), we present the PDG predicted for this Java method with NeuralPDA. We can see that NeuralPDA predicts all CFG/PDG edges accurately and misses one control-dependence edge $S_2 \rightarrow S_7$. However, given that the baseline correctly identifies the control flow edge $S_2 \rightarrow S_7$, it is possible that it could not conclude whether S_2 determines the execution of S_7, resulting in the miss.

Previous works (Vaswani et al., 2017; Clark et al., 2019) have shown that the attention heads in multi-layer SAN-based models tend to capture specific syntactic and semantic

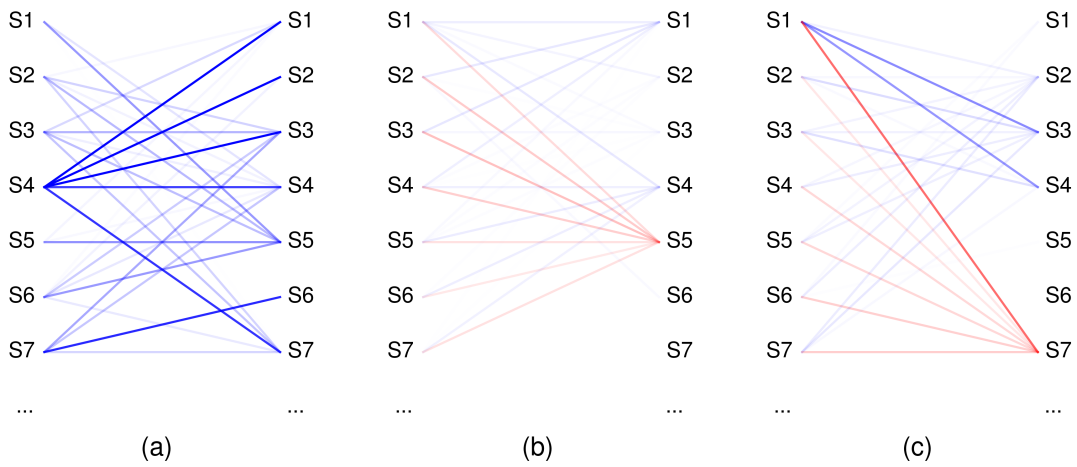


Figure 3.6: Attention heads from NeuralPDA for the Java program in Figure 3.5: (a) L6-H4, (b) L4-H1, (c) L3-H3. Here, Ln and Hm denote n-th layer and m-th attention head.

properties. To better interpret what NeuralPDA learns in this example, we plotted the attention maps for all heads across six layers in our model's inter-statement contextualization phase, some of which we illustrate in Figure 3.6. In Figure 3.6(a)–(c), a higher dependence of S_i on S_j is indicated by a darker edge. For example, in Figure 3.6(a), we can see that in the 4^{th} attention head on the 6^{th} layer (L6-H4), S_4 attends to S_1, S_2, S_3, S_4, and S_7. More interestingly, in Figure 3.6(b), *i.e.*, in L4-H1, all the statements attend to S_5 but not S_7 (indicated in red). Similarly, in Figure 3.6(c), *i.e.*, in L3-H3, we can see that all the statements attend to S_7 but not S_5. NeuralPDA possibly learns the sense of an execution trace on these attention heads, because if S_5 is executed, S_7 cannot be, and vice versa.

3.5 Concluding Remarks

In this chapter, we introduced the first neural network-based approach for program dependence analysis (in particular, data and control dependencies) that extends such analyses of complete programs to partial programs. We do so by modeling this task as a statement-pair dependence decoding problem, aided by intra-statement and inter-statement contextualization. Our empirical evaluation demonstrated that NeuralPDA achieved high accuracy in generating

CFG/PDGs for both complete and partial programs with significant time efficiency, being 380× faster. We also demonstrated NeuralPDA's effectiveness in detecting vulnerabilities in both complete and partial programs. Other software engineering tasks that can tolerate some level of inaccuracies can also benefit from NeuralPDA.

CHAPTER 4

REASONING ABOUT VARIABLE-STATEMENT DEPENDENCIES IN LATENT SPACE: LEARNING TO SLICE PROGRAMS

4.1 Overview

Program slicing is a fundamental program analysis technique that identifies the set of program statements that affect or are affected by a variable at a designated location, referred to as the *slicing criterion*. The set of statements that may influence the slicing criterion forms the *backward slice*, while those that may be influenced by it, the *forward slice*. Slicing is widely useful in both manual and automated debugging, as it helps developers focus on a minimal subprogram relevant to understanding or fixing program faults (Francel and Rugaber, 2001).[1]

There are two primary forms of slicing:

- **Static slicing**, which analyzes program structure to conservatively approximate all statements that *may* influence or be influenced by the criterion across *all* program executions.
- **Dynamic slicing**, which analyzes program execution traces to identify statements that *actually* influence or are influenced by the criterion in a *specific* program execution.

While they differ in whether they rely on program execution, both static and dynamic slicing ultimately involve reasoning about variable-statement dependencies to determine the backward or forward slice. In practice, however, both approaches face significant limitations.

[1]The content presented in this chapter is based on the following publications (Yadavally et al., 024a)(Yadavally et al., 024b):

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

Aashish Yadavally, Yi Li, and Tien N. Nguyen. 2024. "Predictive Program Slicing via Execution Knowledge-Guided Dynamic Dependence Learning". *In Proceedings of the ACM on Software Engineering 1, FSE (2024)*, 271–292.

Traditional compiler-based static slicing requires access to the entire source code of a system to build a precise model of dependencies, an assumption that often fails in real-world scenarios where parts of the codebase, such as third-party libraries or external modules are not available. Dynamic slicing, in contrast, requires the collection and storage of runtime information, which introduces computational overhead. These challenges are further exacerbated in security-sensitive or resource-constrained environments, where enabling dynamic instrumentation or executing programs for specific inputs may be infeasible and time-consuming.

Given these constraints, a natural question arises:

Can we learn to approximate static and dynamic slices directly from source code, by reasoning about the variable-statement dependencies in latent space?

To this end, we explore two complementary formulations:

- NS-Slicer (see Section 4.2): Given a program and a slicing criterion, the goal is to predict all statements that may semantically influence or be influenced by the criterion across any possible execution path. The model is trained to approximate these variable-statement dependencies by leveraging pre-training on program structure and semantics.

- ND-Slicer (see Section 4.3): Given a program, a slicing criterion, and a concrete test input, the goal is to predict the sequence of statements that *actually* influence the criterion during that specific execution. To enable such a behavioral approximation, the model is trained to identify such dynamic dependencies by leveraging execution-aware pre-training.

Together, these models enable the construction of approximate slices that are both scalable and effective. While NS-Slicer prioritizes over-approximation and generalization to incomplete or unseen code, ND-Slicer learns runtime behaviors from past executions, generalizing slicing across programs and diverse test inputs without actual execution. Notably, both are inherently language-agnostic and extensible to multilingual software systems.

```
1   if(codec != null)
2     in = new LineReader (codec.createInputStream(fileIn), job);
3     end = Long.MAX_VALUE;
4   } else {
5     if(start != 0)
6        skipFirstLine = true;
7        --start;
8        fileIn.seek(start);
9     }
10    in = new LineReader(fileIn, job);
11  }
12  if(skipFirstLine)
13    start += in.readLine (new Text(), 0,
14         (int) Math.min((long) Integer.MAX_VALUE, end - start));
15  }
```

Figure 4.1: Motivating example for NS-Slicer: StackOverflow post #16180130.

4.2 NS-Slicer: Learning Static Program Slices

4.2.1 Motivation and Key Ideas

Illustrating Example. Whether detecting vulnerabilities manually or with automated tools, it is useful to only focus on the statements that can influence or be influenced by a variable of interest, *e.g.*, `in` at line 13 in Figure 4.1. In this case, variable `in` serves as the *slicing criterion*, and its backward slice includes lines 12, 10, 8, 7, 6, 5, 2, and 1. Notably, this slice also captures both the null pointer exception (NPE)-inducing statements, *i.e.*, lines 2 and 10, illustrating how slicing can help isolate root causes of program faults.

Existing approaches to *partial program slicing* are limited. *First*, traditional slicing tools, such as JavaSlicer (Galindo et al., 2022), extract program slices by building a system dependence graph (SDG), which requires access to complete code along with all of their dependencies, an assumption that breaks down for incomplete code snippets from online forums (*e.g.*, StackOverflow). Alternatively, combining a program dependence graph (PDG) from an automated tool (*e.g.*, Joern (Joern, 2023)) with data/control-flow analysis to extract the program slice does not work either, due to the under-representative PDGs resulting from:

(a) missing declared data types, (b) missing variable declarations, (c) certain data types are unresolved due to the missing imports for external libraries, (d) references to undeclared libraries, (e) inability to process templates (see Section 3.2.1).

Finally, one could attempt to build a PDG using a learning-based approach, such as NeuralPDA (Yadavally et al., 2023) (Chapter 3), which is effective for code snippets. However, by design, NeuralPDA operates at the statement level and exhibits two key limitations: (a) it does not distinguish between data and control dependence edges, and (b) it does not identify the specific variable or argument involved in a given data dependency. This level of granularity is essential for performing subsequent data/control-flow analysis to extract program slices. Thus, NeuralPDA is not suitable for variable-specific slicing tasks.

These limitations motivate our proposed approach, NS-Slicer, a neural network-based *partial program slicing* technique that learns to approximate static slices for any given criterion. It is particularly useful in scenarios involving (in)complete code where a low level of errors and imprecision is acceptable as a trade-off for improved scalability and computational efficiency. In designing NS-Slicer, we have the following key ideas:

Key Idea 4.1. *Neural Network-Based Partial Program Slicing*

A learning-based approach for static program slicing learns to capture fine-grained dependencies among the variables within and across statements by training on complete programs from open-source projects (*e.g.*, GitHub). This enables the model to generalize to incomplete code snippets, making it suitable for real-world settings. Our intuition for such a formulation is driven by: (a) traditional program slicing (Galindo et al., 2022) techniques work well for complete code, allowing us to build the training data containing $\langle$*Program*, *Slicing Criterion*, *Backward/Forward Slices*$\rangle$ tuples; (b) pre-trained NLMs operate over token sequences and are inherently independent of program completeness, enabling them to process both complete and partial programs effectively.

In this chapter, we aim to approximate the process of building a static program slice from program dependence graphs. Traditionally, for a specific slicing criterion: (*Step I*) one would build the PDG; (*Step II*) apply data-flow analysis to determine which variables influence or are influenced by the criterion; (*Step III*) traverse the PDG and identify corresponding statements that influence or are influenced by the criterion as backward and forward slices.

Key Idea 4.2. *Pre-Trained Language Model for Variable-Statement Dependence Learning*

Previous studies (Hernández López et al., 2023; Guo et al., 2021) have demonstrated the ability of pre-trained NLMs (in short, PLMs) to learn both syntactic and semantic properties of source code. Furthermore, NeuralPDA (Yadavally et al., 2023) reinforces the potential of attention-based models to learn program dependencies at the statement level (*i.e.*, among statements). On this basis, we posit that PLMs can discern the *variable-statement dependencies* for program slicing. Given its data flow-specific learning objectives, we opted for GraphCodeBERT (Guo et al., 2021) as the PLM in NS-Slicer. We expect that with this knowledge, GraphCodeBERT will help NS-Slicer in Step II described above. In Section 4.4.7, we present a case study to probe and validate such a dependency learning.

Key Idea 4.3. *Mimic Program Slice Construction with Dedicated Multi-Pass Classifiers*

The PLM in NS-Slicer contextualizes the source code tokens to produce syntax and semantics-aware token representations, which encoding knowledge about how *variables* across statements influence or are influenced by the slicing criterion. We pool together token representations of the individual tokens that make up the variable at the criterion as well as all program statements to obtain their corresponding criterion and statement representations.

Next, we mimic Step III of traditional PDG-based static slicing by designing dedicated multi-pass classifiers that, for each program statement, use its pooled statement representation to ascertain whether it belongs to the backward or forward slice, respectively. Two fundamental factors drive the design of such distinct classifiers. *First*, backward and forward slices are

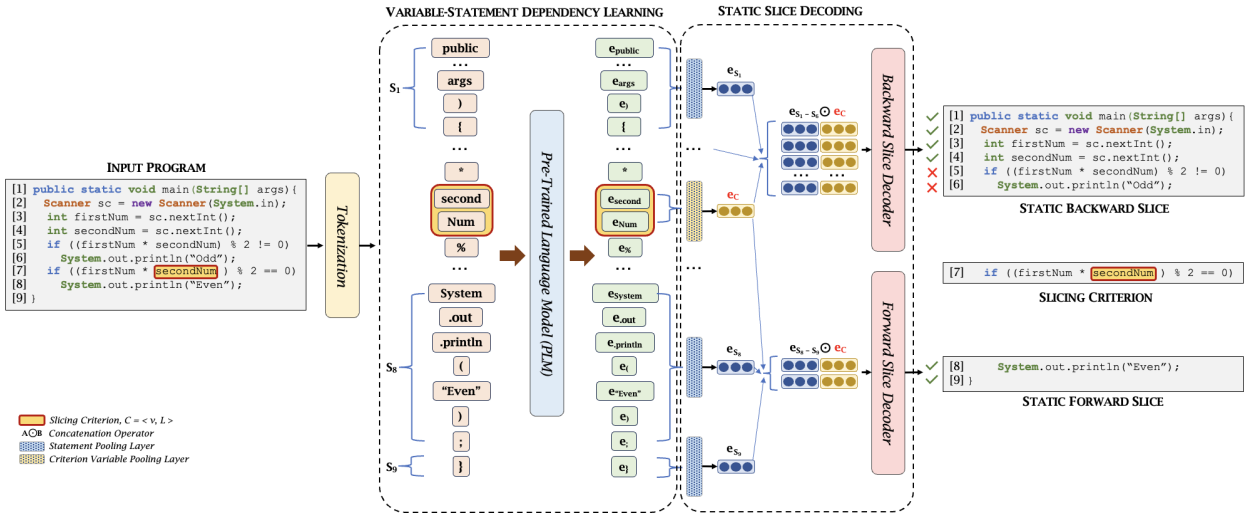


Figure 4.2: Model architecture for NS-Slicer

inherently different, as each comprises statements that either affect or are affected by a slicing criterion. We need distinct slicing classifiers to capture this nuance effectively. *Second*, static slicing considers all possible paths to/from the slicing criterion. To probe all such paths, we need to inspect the set of all statements that either precede or follow the specified criterion. However, the cardinality of these statement sets is not constant and is determined by the location of the criterion within the program. Therefore, we need a multi-pass classifier, where each pass involves considering the criterion variable and a statement from the backward/forward statement sets, to determine whether that statement belongs to the corresponding slice or not. Note that such a design is not suitable for dynamic slicing, in which case one would explore only one path to/from the slicing criterion (see Section 4.3).

4.2.2 Approach Overview

In Figure 4.2, we present a general overview of NS-Slicer, a learning-based approach to static program slicing which primarily has two phases: *training* and *inference*. The training phase yields a neural model that, given a complete/partial program and a slicing criterion, predicts the corresponding backward and forward slices. To this end, we train the model

on ⟨*Program*, *Slicing Criterion*, *Backward/Forward Slice*⟩ tuples, where the ground-truth program slices are extracted using a traditional static program slicing tool. Thus, the training phase requires that the collected programs are complete. During inference/prediction, possibly incomplete code snippets can be input to NS-Slicer.

For both training and inference, we: *first*, split the given program into a sequence of tokens (*i.e.*, tokenization); *second*, identify variable-statement dependencies that encode the knowledge of the variables across statements that influence or are influenced by the criterion; *third*, construct static program slices by discarding irrelevant statements in the program by identifying the sets of statements that affect or are affected by the criterion.

NS-Slicer has the following essential components:

- *Variable-Statement Dependence Learning.* Previous studies have shown that PLMs are capable of capturing both syntax and semantics of programming languages (Guo et al., 2021; Hernández López et al., 2023). Building on this, we incorporate PLMs in NS-Slicer. Through the training process, we expect that this component will leverage such knowledge of fine-grained dependencies in source code to learn to apply the flow analyses and identify variable-statement dependencies. Moreover, the inherent sequential nature of PLMs ensures NS-Slicer can operate on both complete and partial code, thus overcoming the limitations of traditional program slicing approaches, which work only on complete code.

  Consider a program $P = \langle s_1, s_2, \cdots, s_N \rangle$, where each statement s_i consists of a sequence of code tokens $\langle c^i_1, c^i_2, \cdots, c^i_{M_i} \rangle$ on the i-th line (i.e., a total of M_i tokens). Accordingly, we input the sequence of code tokens $\langle c^1_1, c^1_2, \cdots, c^1_{M_1}, \cdots, c^N_1, c^N_2, \cdots, c^N_{M_N} \rangle$ to the PLM. For each of the tokens c^i_m in P, the PLM yields a rich and contextualized token representation $e_{c^i_m} \in \mathbb{R}^d$ (where d is the input size), that is both syntax and semantics-aware.

- *Pooling Layers.* A pooling operation is useful to aggregate sequences of token representations. In NS-Slicer, we have: (a) *variable pooling layer*, that computes the representation for

the variable at the slicing criterion (*i.e.*, *slicing variable embedding*) from the token representations of the comprising individual tokens in the criterion variable; and (b) *statement pooling layer*, that computes the representation for a program statement (*i.e.*, *statement embedding*) from the token representations of the comprising individual tokens. We use *mean-pooling* as the default aggregation strategy within both pooling layers.

Let the variable corresponding to the slicing criterion be denoted as $v = \langle c^x_{a+1}, \cdots, c^x_{a+b} \rangle$, comprising b tokens on the x-th line. Let s_i be the statement, whose membership in the static program slice needs to be determined. We compute the pooled representation of the slicing criterion as:

$$e_v = \frac{1}{b} \sum_{j=a+1}^{a+b} e_{c^x_j} \tag{4.1}$$

and that of the statement as:

$$e_{s_i} = \frac{1}{M_i} \sum_{j=1}^{M_i} e_{c^i_j} \tag{4.2}$$

where M_i is the number of tokens in s_i, and $e_{c^i_j} \in \mathbb{R}^d$ is the contextualized token representation of c^i_j that is syntax and semantics-aware, as produced by the PLM.

- *Static Slice Decoding.* The static slice decoding phase has two components: *backward slice decoder*, and *forward slice decoder*. Both components construct the static backward and forward slices by deleting the parts of the program that do not affect, or are not affected by the variable at the slicing criterion, respectively. This is akin to traversing the PDG in the backward and forward directions to compute the corresponding program slices, except that, the knowledge of the *variable-statement dependencies* required for deriving the program slices is encoded in the statement representations e_{s_i}.

  We capture the nuanced difference between static slice decoding components by leveraging two distinct 2-layered multi-layered perceptrons (*i.e.*, MLP_b for backward slicing and MLP_f for forward slicing). To inspect whether a statement s_i in a given program P is involved in

the computation of the variable v on the x-th line or not, we score the MLPs as follows:

$$\text{score}_{slice}(s_i, v_x) = \text{MLP}_{slice}(e_{s_i} \circ e_{v_x}) \tag{4.3}$$

where "$\circ$" corresponds to the concatenation operation, and $slice \in \{b, f\}$.

If a statement s_i attains a $\text{score}_{slice}(s_i, v_x) > 0.5$, it indicates that s_i belongs to the static backward/forward slice of the variable v on line x in the given program P. The combination of all such statements represents the predicted static program slice, wherein, the backward and forward slices are:

$$\hat{B} = \{s_i \mid s_i \in P \wedge \text{score}_b(s_i, v_x) > 0.5, 1 < i < x\} \tag{4.4}$$

$$\hat{F} = \{s_i \mid s_i \in P \wedge \text{score}_f(s_i, v_x) > 0.5, x < i < N\} \tag{4.5}$$

Note that MLP_b is applied to all statements preceding the slicing criterion ($1 < i < x$) and MLP_f, to the ones following it ($x <$ i $< N$). Thus, both sets $\hat{B}$ and $\hat{F}$ are disjoint.

4.2.3 Training Process

For a given program $P = \langle s_1, s_2, \cdots, s_N \rangle$ and variable v on the x-th line as the slicing criterion, let the ground-truth backward and forward slices be $B = \langle y_1, y_2, \cdots, y_{x-1} \rangle$ and $F = \langle y_{x+1}, y_{x+2}, \cdots, y_N \rangle$, respectively. Notably, the backward slicing classifier is applied to all statements preceding the x-th line, and forward slicing classifier is applied to all that follow it, with y_i meaning that it belongs to either of them or not. Due to its formulation as binary classification, the ground-truth backward slices B_j for all complete programs P_j in the training data D can be utilized to compute the training loss for backward slicing (*i.e.*, $\mathcal{L}_b$) as:

$$\mathcal{L}_b(\theta) = \sum_{P_j \in D} \sum_{i=1}^{x_j - 1} \{y_i \log(p^b(s_i, y_i) + (1 - y_i) \log(1 - p^b(s_i, y_i))\} \tag{4.6}$$

```
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])
```

Figure 4.3: Motivating example for ND-Slicer: Python code example.

```
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)
```

Figure 4.4: Execution trace and program slices for motivating example in Figure 4.3.

Similarly, the ground-truth forward slices F_j for all the complete programs P_j in D can be utilized to compute the training loss corresponding to forward slicing (*i.e.*, $\mathcal{L}_f$) as:

$$\mathcal{L}_f(\theta) = \sum_{P_j \in D} \sum_{i=x_j+1}^{N} \left\{ y_i \log(p^f(s_i, y_i) + (1 - y_i) \log(1 - p^f(s_i, y_i)) \right\} \tag{4.7}$$

The final joint training loss for NS-Slicer can be computed as follows:

$$\mathcal{L}(\theta) = \min_\theta \left\{ \mathcal{L}_b(\theta) + \mathcal{L}_f(\theta) \right\} \tag{4.8}$$

4.3 ND-Slicer: Learning Dynamic Program Slices

4.3.1 Motivation and Key Ideas

Illustrating Example. Consider the Python program in Figure 4.3 inspired from an instance in CodeNetMut dataset (Liu et al., 2023). It 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 Figure 4.4, we illustrate its execution trace. The bug occurs on line 6 in the code listing, 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 aid developers in debugging this fault, one can use program slicing tools. The static backward slice of the variable `s` on line 10 in the running example includes the lines 10, 9, 7, 6, 5, 4, 3, 2, and 1. We can see that this covers all statements that can potentially affect the variable `s` on any program execution rather than focusing on a particular execution. This is not very helpful. For instance, in evaluating which statements impact the value of `s` on line 10, one such slicing technique must examine both branches of the `if`-statement on line 6, even if only one of them is executed. 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 limiting its effectiveness.

Alternatively, one can employ dynamic slicing to identify the relevant statements. The dynamic slice includes statements that affect the value of `s` on line 10 during the second iteration of the `for`-loop. We denote the slicing criterion for the variable `s` as "10(2)", where the parentheses indicates the statement's occurrence in the execution trace. The resulting dynamic backward slice, shown on line 3 in Figure 4.4, is longer than its static counterpart but offers greater precision by accounting for the specific execution path, including loop iterations. This enables developers to trace back to the critical statements, ultimately identifying the root cause on line 6. The bug can potentially be fixed by adding the verification: `v - r >= 0`. However, in certain development scenarios as discussed above, execution the programs may not be feasible, thus limiting the applicability of runtime behavior analyses.

To address these challenges, we present ND-Slicer, a learning-based approach that predicts dynamic backward slice for a given slicing criterion (referred to as *predictive slice*), without executing the program. In designing ND-Slicer, we have 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) a given program and its test input, (b) slicing criterion, and (c) the corresponding dynamic backward slice. In the running example, this translates to the following: (a) program in Figure 4.3 and its test 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 Figure 4.4.

Key Idea 4.4. *Dynamic Dependence Learning (DDL)*

Previous work has highlighted the effectiveness of sequence-based models in learning program dependencies (Guo et al., 2021; Yadavally et al., 2023). These approaches benefit from their ability to work with incomplete code. Along similar lines, we adopted a sequence-to-sequence framework for predictive slicing. Here, the encoder aims to learn the nuances of source code from the training instances, 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 the DDL component, here, 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). Similarly, 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 to predict the backward slice.

Second, a criterion must be defined at a precise point in 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. Thus, ND-Slicer needs to understand and distinguish between the execution of line 10 in different iterations of the loop.

Key Idea 4.5. *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 to learn 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. For the running example, combining Key Ideas 4.3 and 4.4 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 $10(2) \rightarrow 7(1) \rightarrow 6(2)$ and $10(1) \rightarrow 9(1) \rightarrow 6(1)$, 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.

Key Idea 4.6. *Attention-Based Decoders for Slice Generation*

We leverage the Transformer (Vaswani et al., 2017) decoder (see Section 4.4.2 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. Based on this, Transformer decoder learns the conditional probability of generating a sequence containing these discrete line

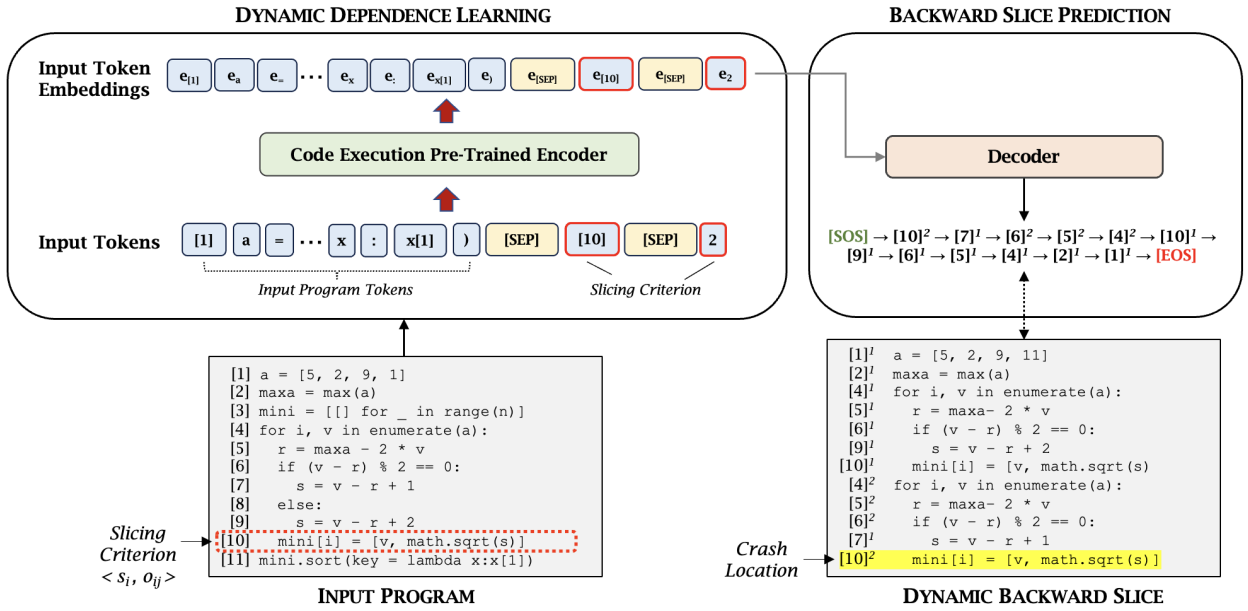


Figure 4.5: Model architecture for ND-SLICER

numbers, representing the predictive slice. In the running example, it learns to correctly predict the output sequence 10(2) → 7(1) → 6(2) → 5(2)... (line 3 in Figure 4.4).

4.3.2 Approach Overview

In Figure 4.5, we illustrate the model architecture for ND-SLICER. Broadly, it adopts the sequence-to-sequence framework for predictive slicing with the following essential components:

- *Dynamic Dependence Learning.* In recent times, specialized source code models have highlighted the effectiveness of learning-based approaches in understanding data and control dependencies (Guo et al., 2021; Yadavally et al., 2023). However, their learning objectives do not focus on run-time behavior of code, limiting their applicability to predictive slicing. To address this challenge, we opted for CodeExecutor (Liu et al., 2023), a model pre-trained on code execution, as the encoder in our framework. This execution-aware pre-training equips ND-SLICER with the capability to *learn dynamic program dependencies*. In addition,

treating source code as a sequence of tokens, PLMs can operate on both complete and partial code, thereby overcoming the practical limitations of dynamic analyses.

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:

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

For each token t in this input representation, the DDL module generates a contextualized 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 execution-flow based on the input tokens $t_x \in I_P$; (3) it learns dynamic data and control dependencies between $t_j \in P$ and $t^{(c)}$ based on the occurrence of the criterion denoted by $[c]$ and o, respectively.

- *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 *attention* in propagating information from the input sequence to the decoder within the sequence-to-sequence framework, we adopted the Transformer decoder in ND-Slicer (see Section 4.4.2 for all variants).

  The encoder in the DDL component maps the input sequence into a series 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, where each y_k corresponds to the special

line identifier token $[i]$ from the input. This process is auto-regressive, where the previously generated symbols are consumed as the additional input for generating the next.

4.3.3 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:

$$P(\mathbf{y}|\mathbf{x}) = P(y_1, y_2, ..., y_k|\mathbf{x}) = \prod_{i=1}^{k} p(y_i|y_{<i}, \mathbf{x}) \tag{4.9}$$

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:

$$\mathcal{L}_{step}(\theta) = -p^* log(p) = -\sum_{j=1}^{|V|} p_j^* log(p_j) \tag{4.10}$$

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:

$$\mathcal{L}_{instance}(\theta) = -\frac{1}{k}\sum_{i=1}^{k} \log(p(y_i|y_{<i}, x)) \tag{4.11}$$

During inference, we start with the **[SOS]** token and follow the simple transduction process by decoding from left-to-right, picking 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.

4.4 Empirical Evaluation

4.4.1 Effectiveness of NS-Slicer on Static Slicing Benchmark

Data Collection. For evaluating NS-Slicer on complete programs, we considered IBM's Project CodeNet dataset (Puri et al., 2021), which includes 4,053 programming challenges for several languages. In particular, we focus on Java language as the main resource, which covers 250 problems and a total of 75,000 solutions. We parsed all such Java programs using Eclipse JDT to identify the locations of different variables in the programs, collecting the corresponding static program slices using JavaSlicer (Galindo et al., 2022).

Since we model the program slice decoding task as binary classification, it is imperative to ensure that the labels for positive and negative samples are balanced. Thus, we only retain those instances in which the ratio (r) of the statements belonging to the backward/forward slice, to the ones not belonging to the corresponding slice lies between 0.3 and 0.7, *i.e.*, $0.3 < r < 0.7$. r is the range for the ratio between the number of positive and negative samples (statements) that are input to the slicing classifiers. Note that this ratio does not correspond to the size of the slice over the program's size. By not enforcing a form of balance of the positive and negative samples in this manner, the classifiers would not capture well the notion of belonging, or not belonging to the slice. Accordingly, we retain approximately 43,000 data instances corresponding to the Java programs, each containing between 5–69 statements per program. Finally, we split them at the problem-level in a 80%–10%–10% ratio, *i.e.*, dedicating the Java programs and their slices across 200 problems for training, 25 for validation, and 25 for testing, respectively. Such problem-level splitting avoids data corruption across the dataset splits, thus better representing a realistic scenario.

Procedure. Pre-trained language models (PLMs) on source code benefit from pre-training tasks by learning to encode the syntax (Hernández López et al., 2023) and semantics (Guo et al., 2021) in programming languages. In particular, the data flow-specific pre-training

in GraphCodeBERT (Guo et al., 2021) encodes in it the relation of where the value in a variable comes from, making it suitable for our task of static program slicing. Thus, we used GraphCodeBERT as the PLM in NS-Slicer. During the training phase, we fine-tuned it besides training the MLP heads corresponding to backward and forward slicing, respectively.

This section assesses how closely NS-Slicer *mimics* the traditional program slicing tool, *i.e.*, JavaSlicer in terms of the resulting slices. Accordingly, we picked multiple baselines to compare against our model. First, we chose pre-trained version of GraphCodeBERT off-the-shelf. In this case, during the training phase of NS-Slicer, the parameters within the PLM were fixed, and the token/statement representations were used as is to train the backward and forward slicing MLP heads. Next, we chose the state-of-the-art CodeBERT model (Feng et al., 2020a), considering both the pre-trained and fine-tuned versions as above.

The backbone of PLMs in NS-Slicer is the standard RoBERTa-base architecture (Liu et al., 2019) which has 12 Transformer-encoder layers, each with 12 attention heads. We used the byte-pair encoding (BPE) (Liu et al., 2019) scheme in the pre-trained `RobertaTokenizer` for splitting the given source code into sub-tokens, each initialized as per the NS-Slicer variant. The dimension size for token representations thus produced by these PLMs is 768. All our experiments were conducted on an NVIDIA RTX A6000 GPU. With a batch size of 64, we initialized all variants with learning rates of 0.0001 and 0.0005 during training, reporting the best-performing models. Overall, NS-Slicer (with both CodeBERT and GraphCodeBERT) has about 128M parameters, and took about 32 minutes per epoch for training.

Evaluation Metrics. We model the program slice decoding step in NS-Slicer as a binary classification problem at each statement. Thus, we adopt the standard evaluation metrics:

$$\begin{aligned} \text{Accuracy-S} &= \frac{\text{TP} + \text{TN}}{\text{TP} + \text{FP} + \text{FN} + \text{TN}}, \\ \text{Recall} &= \frac{\text{TP}}{\text{TP} + \text{FN}}, \\ \text{Precision} &= \frac{\text{TP}}{\text{TP} + \text{FP}}, \text{and} \\ \text{F1-Score} &= \frac{2 \times \text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}}. \end{aligned} \tag{4.12}$$

Here, TP: True Positives, FP: False Positives, FN: False Negatives, and TN: True Negatives.

Next, to capture the model performance at the slice-level, we report a stricter Exact-Match Accuracy (*i.e.*, *Accuracy-EM*) that determines the number of times NS-Slicer predicts the backward and forward slices exactly the same as the ground-truth slices from JavaSlicer. Finally, we report Dependence Accuracy (*i.e.*, *Accuracy-D*) to assess how accurately inter-statement dependencies are predicted, causing a particular statement to be included in the slice. We define *Accuracy-D* for a particular program as the ratio of the correctly predicted dependencies to the actual dependencies across all slicing criteria for that program, finally reporting the mean across all programs in the dataset.

Empirical Results. In Table 4.1, we report the performance of NS-Slicer on complete Java programs, comparing different PLMs for the static program slicing task. We can see that using GraphCodeBERT (rows 10–12) produces the most competitive results, predicting the backward and forward slices with an F1-score of 97.41% and 95.82%, respectively. Overall, it best predicts the complete static slice with an F1-score of 96.77%. Furthermore, these match ground-truth program slices constructed by JavaSlicer exactly, in 85.20% of the cases. Note that the ground-truth program slices are executable, *i.e.*, are syntactically valid and can be executed independently of the main program. Thus, 85.20% of program slices predicted by NS-Slicer are also executable, demonstrating its utility in debugging real-world programs.

Table 4.1: Effectiveness evaluation of NS-Slicer on complete programs

Approach		Slicing Criterion	Evaluation Metrics (in %)					
			A-EM	*A-D*	*A-S*	*P*	*R*	*F1*
Pre-Trained	C/BERT	Backward	43.21	92.24	84.40	88.52	88.30	88.41
		Forward	40.84	94.80	85.76	83.62	87.20	85.37
		Overall	**42.03**	**92.46**	85.06	86.51	87.86	**87.18**
	GC/BERT	Backward	47.53	92.54	86.91	91.75	88.54	90.12
		Forward	49.75	95.76	88.07	86.03	89.49	87.72
		Overall	**48.64**	**93.15**	87.47	89.36	88.92	**89.14**
Fine-Tuned	C/BERT	Backward	81.72	97.72	95.65	97.01	96.52	96.76
		Forward	83.47	97.88	95.59	95.31	95.45	95.38
		Overall	**82.59**	**97.56**	95.62	96.33	96.09	**96.21**
	GC/BERT	Backward	85.77	98.21	96.51	97.60	97.21	97.41
		Forward	84.62	98.49	95.99	95.20	96.45	95.82
		Overall	**85.20**	**98.09**	96.26	96.63	96.91	**96.77**

CodeBERT is pre-trained on the masked-language modeling (MLM) and replaced-token detection (RTD) learning objectives, compared to GraphCodeBERT, which leverages code structure and data-flow for pre-training. As a result, using GraphCodeBERT in place of CodeBERT as PLM in NS-Slicer results in a relative improvement in overall F1-score by 0.58%, and exact-match accuracy by 3.16%. This improvement can possibly be attributed to the data-flow knowledge in GraphCodeBERT, which ensures both statements containing the variable belong to the slice. We conducted t-test between CodeBERT and GraphCodeBERT PLMs in NS-Slicer, and the results show the improvements are significant with $p < 0.01$.

We also report the performance of NS-Slicer using CodeBERT and GraphCodeBERT off-the-shelf (rows 1–3 and 4–6 in Table 4.1). Here, only the backward and forward slicing MLPs are trained. The results demonstrate a decrease in overall F1-score by 11% and 8.56%, respectively. Moreover, there is a notable decline in the exact-match accuracy, with a drop of 102.71% for CodeBERT and 75.16% for GraphCodeBERT. This *reinforces the complexity of the program slicing task* and underscores the rationale behind NS-Slicer's design. Furthermore, NS-Slicer also achieves very high Dependence Accuracies (*Accuracy-*

D) across all the models from 92.46%–98.09%. This validates our hypothesis on the ability of PLMs in learning to reason about variable-statement dependencies.

RQ. How accurate is NS-Slicer in predicting static slices for complete Java programs?

RA. NS-Slicer predicts static slices for complete Java programs with an overall F1-score of 96.77%, exactly matching the ground-truth in 85.2% of cases.

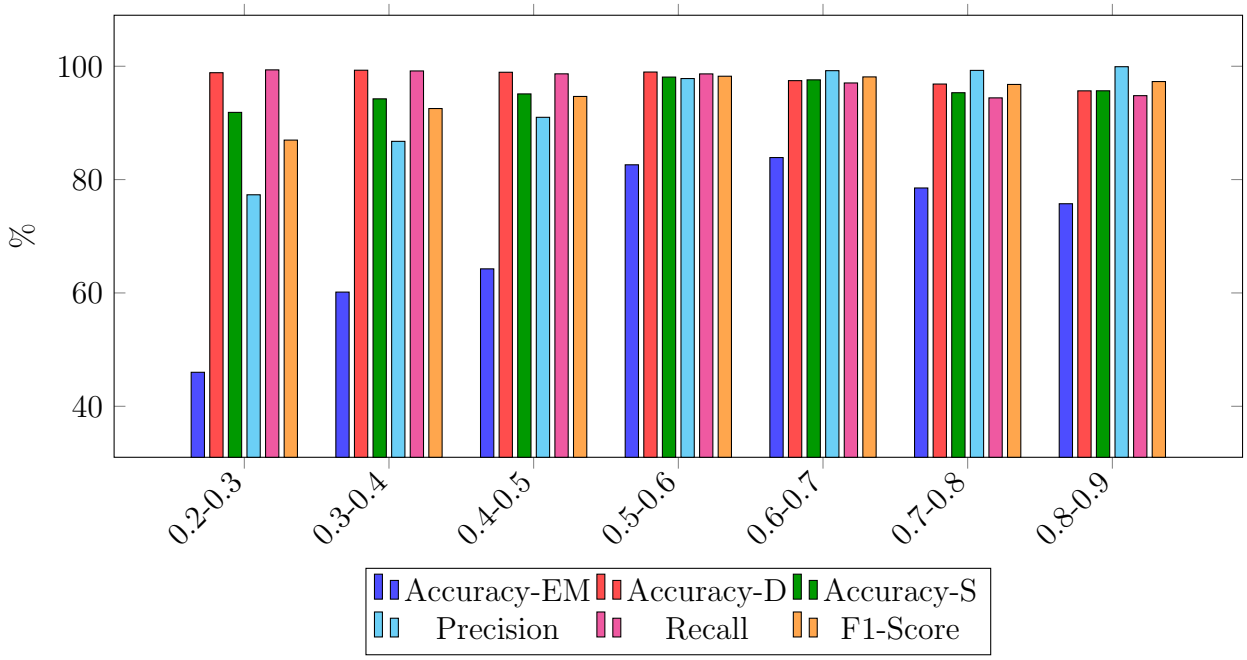


Figure 4.6: Sensitivity of NS-Slicer to the static slice sizes as a fraction of entire program

Sensitivity to the Size of Static Program Slices. In Figure 4.6, we report the stratified performance of NS-Slicer with fine-tuned GraphCodeBERT (best-performing model variant) based on the ratio (r) of the sizes of static program slices to the corresponding lengths of the programs. Here, note that r in the given ranges (*i.e.*, 0.2–0.3 to 0.8–0.9) corresponds to 1.1%, 10.7%, 17.8%, 23.5%, 21.8%, 19.1%, and 5.8% of the test set, respectively.

A lower value of r indicates fewer variable-statement dependencies in the program. Notably, when $0.2 < r < 0.5$, we can see that dependence accuracy and recall are among the highest. This showcases the effectiveness of NS-Slicer in identifying the presence of such

dependencies. However, based on the lower precision in these cases, we can conclude that it does not identify the statements that do not belong to the static program slice as well. As a result, the exact-match accuracy and F1 scores are also lower (compared to when $r > 0.5$). We can attribute this skewedness to the fewer number of examples with a low value of r during training. In addition, the nature of static program slicing diminishes the consequences of including such non-dependent statements in the slice.

4.4.2 Effectiveness of ND-Slicer on Dynamic Slicing Benchmark

Data Collection. For evaluating ND-Slicer on complete Python programs, we used the CodeNetMut dataset (Liu et al., 2023). This comprises executable Python programs, which are essentially mutation variations of Project CodeNet (Puri et al., 2021). Notably, these programs do not rely on specific external resources such as file contents, external modules, or third-party packages. Moreover, each program includes the input values at the beginning of the program in the form of assignment statements, assigning values to input variables.

For each program in the dataset, execution trace is provided which consists of: order in which the statements execute; 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.

(Agrawal and Horgan, 1990) present several approaches to computing dynamic slices, from which we adopt Algorithm 2. For a given program, first, we build the static program dependence graph (PDG) using *python-graphs* (pyt, 2022) 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 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., 2023), there are a total of 19,541 Python programs along with their execution traces. Among these, we skipped the ones for which *python-graphs*

fails to build 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 extracted a total of 168,289 $\langle P, [c], o, S \rangle$ tuples for predictive slicing. Here, each denotes the input program, the statement in the slicing criterion, the occurrence of the 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.

Procedure. First, we compared our approach with a naive baseline (B_0) to establish a reference for 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 earlier to extract the dynamic slices for corresponding slicing criteria.

Next, instead of arbitrary traces (as in B_0), we used *pre-trained CodeExecutor* (Liu et al., 2023) to first predict the execution traces for all programs in our evaluation dataset, further applying the Dynamic Slicing Algorithm to the predicted traces. Note that this baseline B_1 requires not 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 as a sequence-to-sequence (Seq2Seq) framework, it gives us the flexibility to plug in different encoders and decoders. We considered two encoders: GraphCodeBERT (Guo et al., 2021) and CodeExecutor (Liu et al., 2023). The rationale behind picking GraphCodeBERT is its data flow-specific pre-training objective,

which encodes the relation of where the value of a variable comes from, that is required for program slicing. In essence, Seq2Seq framework with GraphCodeBERT as encoder learns to pick the specific execution path corresponding to the inputs from among all possible program paths. With CodeExecutor, the knowledge from code execution-specific pre-training makes it suitable for learning dynamic data and control dependencies for different program inputs.

The backbone of both encoders is the standard RoBERTa-base (Liu et al., 2019) architecture which has 12 Transformer-encoder layers, each with 12 attention heads. We used 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 Seq2Seq framework, we employed two decoders: the standard Transformer decoder (Vaswani et al., 2017) and the Pointer-Transformer (Prabhu et al., 2020). Transformer decoder comprises a stack of 6 layers, while the Pointer-Transformer combines Transformer decoder and the Pointer-Generator Network (See et al., 2017). The general idea for the Pointer-Transformer is that it copies from the input sequence to generate the output sequence, which helps achieve state-of-the-art performance in solving combinatorial/ordering problems over a finite set of points (*e.g.*, Traveling Salesman Problem). This aligns with the task of detecting sequence of program statements as belonging to the dynamic slice, which is just a sequence of *line identifiers* selected from the input sequence. We facilitate this process by expanding the vocabulary with special position markers that point to input positions. Finally, the model 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 encoders and decoders mentioned above as baselines: GraphCodeBERT + Pointer-Transformer (B_3), GraphCodeBERT + Transformer (B_4), CodeExecutor + Pointer-Transformer (B_5), and CodeExecutor + Transformer (B_6). Overall, ND-Slicer has about 183M parameters. We conducted all our experiments on an NVIDIA

Table 4.2: Effectiveness evaluation of ND-Slicer on executable Python programs

Approach		Evaluation Metrics (in %)			
		$R\text{-}L_P$	$R\text{-}L_R$	$R\text{-}L_F$	*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**

RTX A6000 GPU, training all variants for 10 epochs. We initialized them with learning rates of 0.0001 and 0.0005, reporting the best-performing models.

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:

$$\begin{aligned} \text{ROUGE-L Precision } (R\text{-}L_P) &= \frac{LCS(S,\hat{S})}{L_S}, \\ \text{ROUGE-L Recall } (R\text{-}L_R) &= \frac{LCS(S,\hat{S})}{L_{\hat{S}}}, \text{and} \\ \text{ROUGE-L F1-Score } (R\text{-}L_F) &= \frac{2.R\text{-}L_P.R\text{-}L_R}{R\text{-}L_P + R\text{-}L_R} \end{aligned} \tag{4.13}$$

Empirical Results. In Table 4.2, 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 using pre-trained CodeExecutor (B_1) to predict execution traces and subsequently derive dynamic backward slices, we observed the least favorable results. A possible 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 $\langle$*variable*, *value*$\rangle$ pairs for each statement, thus introducing a cascading error that affects its performance. Conversely, predicting 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 4.2) 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 $\langle$*encoder*, *decoder*$\rangle$-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 used CodeExecutor as the encoder, pairing it with Pointer Transformer and Transformer as 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 best-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.

RQ. How accurate is ND-Slicer in predicting dynamic slices for Python programs?

RA. ND-Slicer predicts dynamic slices for executable Python programs with an overall ROUGE-L F1-score of 95.4%, exactly matching ground-truth in 81.3% of cases.

4.4.3 Applicability of NS-Slicer to Partial Programs

As discussed earlier, traditional approaches can not extract static program slices for partial programs. In contrast, NS-Slicer is not limited by the program's completeness. In this experiment, we exhibit its ability to predict static slices for partial Java programs.

Table 4.3: Effectiveness evaluation of NS-Slicer on partial programs (extracted from Section 4.4.1), where $P\%$ is omitted at both start and end of the given program.

P	**Slicing Criterion**	**Evaluation Metrics (in %)**					
		A-EM	*A-D*	*A-S*	*P*	*R*	*F1*
5%	Backward	85.98	98.26	96.67	97.57	97.40	97.49
	Forward	80.99	97.17	95.59	95.25	95.56	95.40
	Overall	**83.48**	**97.76**	96.14	96.60	96.64	**96.62**
10%	Backward	83.89	98.10	96.17	97.08	96.91	96.99
	Forward	72.91	93.79	94.12	95.45	92.68	94.05
	Overall	**78.40**	**96.13**	95.14	96.37	95.03	**95.70**
15%	Backward	84.54	97.31	95.91	97.32	96.23	96.77
	Forward	62.02	89.89	91.83	96.00	88.57	92.14
	Overall	**73.28**	**93.92**	93.85	96.73	92.68	**94.66**

Procedure. Extracting ground-truth static slices for partial programs is not possible. As a result, for the test Java programs in Section 4.4.1, we strip $P\%$ of the statements at the beginning and at the end of the code example to mimic incompleteness. To evaluate the performance of NS-Slicer on partial programs thus obtained, we extract ground-truth slices corresponding to the retained code from corresponding complete variants. In this process, we omit any instances which end up with empty backward or forward slices. The difficulty in dealing with partial programs obtained in such a manner is exacerbated by the possible deletion of various variable declarations, which further eliminates the *def-use* relationships. Moreover, the higher the value of P, more number of such relationships shall be defied.

Empirical Results. In Table 4.3, we report NS-Slicer's results on partial Java programs obtained by omitting 5%, 10%, and 15% of the complete variants, both at the beginning and end. We record an overall F1-score of 94.66%–96.62%, and an exact-match accuracy of 73.28%–83.48%. Overall, when stripping programs in this manner, an average of 2, 3, and 5 statements per program are omitted, respectively. Notably, in 14.89%, 51.97%, and 59.46% of the cases, the excluded statements contain variable declarations that are referenced in the remaining partial program, thus disrupting the *def-use* chain. Thus, we can explain the

decrease in performance by 2.07% in overall F1-score and 13.92% in exact-match accuracy, as resulting from the deletion of 5% → 15% of the program.

Similar to the the dependence accuracies reported for complete programs in Table 4.1, the corresponding accuracies for partial programs in Table 4.3 range from 93.92% to 97.76%. These results further validate our hypothesis that PLMs can learn variable-statement dependencies.

RQ. How accurate is NS-Slicer in predicting static slices for partial Java programs?

RA. NS-Slicer predicts static slices in partial Java programs with an overall F1-score of 94.66%–96.62%, exactly matching the ground-truth in 73.28%–83.48% of cases.

4.4.4 Applicability of ND-Slicer to Non-Executable Programs

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. In comparison, ND-Slicer is not limited by the completeness of the program. We set up this experiment to exhibit its ability in predicting dynamic program slices for non-executable Python programs.

Procedure. Extracting ground-truth dynamic backward slices for partial programs is not possible without manual intervention. Thus, aiming to mimic a non-executable program, we picked Python programs from the test set in Section 4.4.2 and omitted the `import`-statements for modules whose utilities are being used in the code. 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 evaluated the performance of the best-performing variant of ND-Slicer, *i.e.*, CodeExecutor + Transformer on this non-executable Python program dataset, adopting the same metrics as in Section 4.4.2 to assess its performance.

Table 4.4: Effectiveness evaluation of ND-Slicer on non-executable Python programs

Approach	Statement Type	Evaluation Metrics (in %)			
		$R\text{-}L_P$	$R\text{-}L_R$	$R\text{-}L_F$	*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**

Empirical Results. In Table 4.4, 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 ground-truth dynamic backward slices, improving over the naive baseline by 71.6% and 109.8%, respectively. In particular, 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 statements referencing methods belonging to 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 from external libraries.

RQ. How accurate is ND-Slicer for non-executable Python programs?

RA. ND-Slicer predicts dynamic slices for non-executable Python programs with an overall ROUGE-L F1-score of 81.6%, exactly matching ground-truth in 57.5% of cases.

4.4.5 Usefulness of NS-Slicer in Vulnerability Detection

In this experiment, we explore the applicability of static slices predicted by NS-Slicer for the downstream task of detecting vulnerabilities in Java programs. VulDeePecker (Li et al., 2018) is a popular automated approach that utilizes program slices for this purpose, representing programs as *code gadgets* that are composed of a number of semantically related program statements. Each of these focuses on a *key point* hinting at the existence of a vulnerability. During the training phase, VulDeePecker utilizes a BiLSTM model to learn to detect the vulnerability patterns from code gadgets. In the detection phase, it breaks up a given program into multiple code gadgets, using the trained BiLSTM to predict vulnerabilities.

Next, we present the pipeline for extracting such code gadgets in VulDeePecker. In simpler terms, a code gadget is a static program slice from the arguments of an API method call. In Figure 4.7 (*top*), it: (a) extracts all library/API call arguments, (b) builds PDGs using Joern and applies data flow analysis to extract corresponding gadgets, (c) retrieves *vulnerability labels* for the code gadgets. However, its dependence on traditional PA approaches to build static program slices *limits VulDeePecker to complete code*. Alternatively, NS-Slicer can be plugged into the second step of the process to extract code gadgets, as in Figure 4.7 (*bottom*). We used the predicted slices from NS-Slicer in both the training and prediction phases of VulDeePecker. Such an integration extends its utility to both complete and partial code.

Data Collection, Procedure, and Evaluation Metrics. Most benchmark vulnerability datasets cover C/C++ (*e.g.*, BigVul (Fan et al., 2020b)), and those for Java are limited. CrossVul (Nikitopoulos et al., 2021) is a cross-language vulnerability dataset that includes vulnerable files written in more than 40 programming languages, covering 563 commits across Java files. We parse them using Eclipse JDT and extract a total of 14,139 methods with external libraries. Note that NS-Slicer can work on all these methods regardless of whether they are complete (or compilable) or not. We follow the methodology in BigVul to label

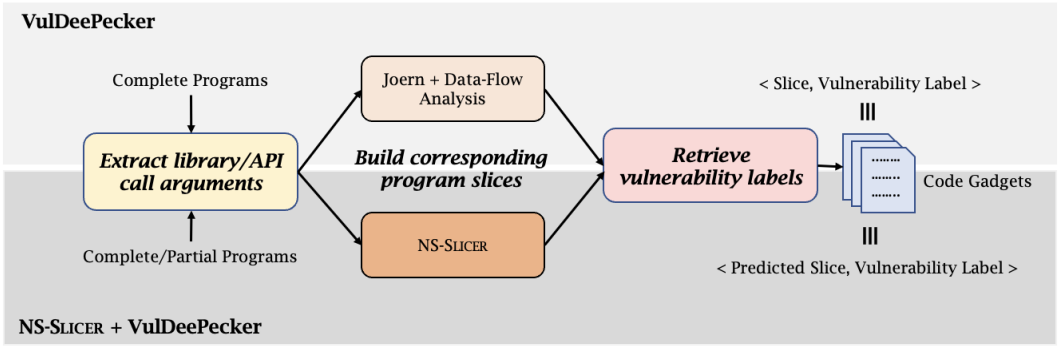


Figure 4.7: Usefulness of NS-Slicer in vulnerability detection: Contrasting *code gadget*-building pipelines in (*top)* VulDeePecker, (*bottom*) NS-Slicer+VulDeePecker.

them as *vulnerable* or *non-vulnerable*, retrieving a total of 574 and 13,565 methods for both, respectively. To ensure a balanced dataset, we only select files that are independent of the ones possessing the vulnerable methods and randomly pick 574 non-vulnerable ones. Finally, we split them in a 80%–10%–10% train–validation–test ratio at the file-level.

Similar to (Li et al., 2018), we picked the library/API function call arguments within a method as the focus area. Next, we collected all library/API call arguments for each method, determining their *vulnerability labels* based on whether the API call belongs to the changed code or not. As discussed earlier, Joern fails to produce quality PDGs in the absence of external libraries, subsequently generating program slices with a low accuracy. Thus, we leverage NS-Slicer to extract the program slices corresponding to the API call arguments, and produce the code gadgets.

Overall, we obtained 1,476 code gadgets for training, 184 for validation, and 187 for testing. Among these, 1,020 are vulnerable and 827 are non-vulnerable. To avoid underfitting of the BiLSTM in VulDeePecker due to limited size of the Java vulnerability dataset, we use the one trained on BigVul as the initialization point. Such a strategy makes use of knowledge gained from predicting vulnerabilities in C/C++ code, leading to a better generalization for Java programs. For evaluation, we adopt the same metrics as in prior vulnerability detection literature (Li et al., 2018; Wu et al., 2022), *i.e*, *Accuracy*, *Precision*, *Recall*, and *F1-Score*.

Table 4.5: Effectiveness evaluation of NS-Slicer in vulnerability detection: Results of NS-Slicer + VulDeePecker, where M corresponds to the pre-trained language model.

NS-Slicer {M} + VulDeePecker	**Evaluation Metrics (in %)**			
	A	*P*	*R*	*F1*
Pre-trained GraphCodeBERT	59.46	64.71	62.86	63.77
Fine-tuned GraphCodeBERT	60.00	58.96	97.14	**73.38**

Empirical Results. In Table 4.5, we present the performance of VulDeePecker in predicting vulnerabilities using program slices, predicted with pre-trained GraphCodeBERT as the PLM in NS-Slicer (row 1), and the fine-tuned version of GraphCodeBERT from Section 4.4.1 (row 2). We can see that using predicted static program slices from NS-Slicer helps predict vulnerabilities in such Java code with an F1-score of 73.38%, while outperforming the baseline by 15.1%. This further reiterates the role of learning *variable-statement dependencies* as opposed to only relying on data-flow knowledge of pre-trained GraphCodeBERT.

Furthermore, as Zhou *et al.* (Zhou and Sharma, 2017) note, real-world vulnerabilities exist in a 9:1 non-vulnerable to vulnerable ratio in code repositories. To test the performance of NS-Slicer + VulDeePecker in a real-world setting, we replicated this ratio across our test set and averaged the performance across 10 samples, achieving an F1-score of 98.67%. However, due to the significantly fewer number of vulnerable code gadgets in such samples, we note that the metrics observed in this experiment setting are skewed.

RQ. How useful are static slices predicted by NS-Slicer in detecting vulnerabilities?

RA. The static program slices predicted by NS-Slicer helps VulDeePecker detect vulnerabilities in partial Java programs with an F1-score of 73.38%.

4.4.6 Usefulness of ND-Slicer in Crash Detection

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

```
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)
```

Figure 4.8: A Python Code Example with an Injected Crash Fault

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.

Procedure. We enabled this experiment by injecting synthetic `ZeroDivisionError` faults in test Python programs from Section 4.4.2. These often occur when a division or modulo operation is attempted with a denominator or divisor of zero. To facilitate such a crash injection, we first identified 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 $n-1$. Finally, we applied the Dynamic Slicing Algorithm considering the location of the crash fault (*i.e.*, line $n-1$) as the slicing criterion to compute the ground-truth dynamic backward slices.

Consider the Python program in Figure 4.8 to elucidate the procedure for crash injection. Here, the value of `cnt` is initialized as `0` on line 4, which changes to `1` at line 7. We pick line 6

as the location to inject the crash fault. Considering first occurrence of line 8 as the criterion, *i.e.*, 8(1), its ground-truth dynamic backward slice would be: {8, 5, 4, 2, 1}. This includes the additional statements not dependent on `cnt`, *i.e.*, 2 and 1 due to variable `s` on line 6.

Overall, we obtained 236 such crash-injected Python programs. Next, we leveraged the best-performing model in Section 4.4.2, *i.e.*, fine-tuned CodeExecutor+Transformer, to predict 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 Figure 4.8). 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.

Empirical Results. Our results indicate ND-Slicer can correctly identify crash locations in 63.9% of faulty programs and predict the dynamic backward slice exactly the same in 47.5% of 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.

RQ. How useful are dynamic slices predicted by ND-Slicer in debugging crashes?

RA. The dynamic slices predicted by ND-Slicer correctly identifies crash locations in 63.9% of faulty programs, exactly matching ground truth in 47.5% of cases.

Table 4.6: Probing PLMs in NS-Slicer for variable aliasing: Comparison of forward slicing performance on complete Java programs. Here, (M_{x}) denotes programs with synthetic variable aliases, while (M^*_{x}) denotes the version without synthetic variable aliases.

Approach	**Evaluation Metrics (in %)**					
	A-EM	*A-D*	*A-S*	*P*	*R*	*F1*
M_{CodeBERT}	34.82	74.33	81.31	90.12	78.95	84.17
M^*_{CodeBERT}	83.47	97.88	95.59	95.31	95.45	95.38
$M_{\text{GraphCodeBERT}}$	38.74	74.23	82.49	90.27	80.89	85.32
$M^*_{\text{GraphCodeBERT}}$	84.62	98.49	95.99	95.20	96.45	95.82

4.4.7 What does NS-Slicer Learn? A Case Study

Variable aliasing is an intrinsic code property in which multiple references when used to refer to the same object, still point to the same location in memory. In practice, aliasing is not straightforward to debug, as the changes caused on one variable also reflects on the other. We set up this experiment to assess how well the PLMs understand variable aliasing, and the effect such aliases have on the performance of NS-Slicer in static program slicing.

Procedure. We enable this experiment by introducing synthetic variable aliases in the test Java programs in Section 4.4.1, by inserting a variable assignment in the statement following the one containing the variable of the requested criterion. Next, in all subsequent statements that contain the variable, we replace the original variable with the alias. For example, let us consider a Java program in which we need to predict the program slice for the variable `x` of type `int`, on statement 5. Let the forward slice for this contain statements 8, 9, 13, and 15, among which statements 9 and 13 contain `x`. In this case, we introduce a variable assignment "`int aliasingVar = x`" on statement 6, and replace `x` on statements 9 and 13 by `aliasingVar`. The new forward slice contains statements 6, 9, 10, 14, and 16, while the backward slice remains unaffected. For the test programs thus obtained, we compared performance of CodeBERT and GraphCodeBERT in NS-Slicer, with and without aliasing.

Empirical Results. In Table 4.6, we report the forward slicing performance of both CodeBERT and GraphCodeBERT PLMs, on test programs with and without synthetic variable aliases. We observed a drop in the forward slicing F1-score, with reductions of only 13.32% and 12.31%, respectively. This shows that NS-Slicer is able to track the dependencies across the variable aliases (illustrated in Figure 4.9). In contrast, forward slicing exact-match accuracy shows substantial declines, with a drop of 139.72% for CodeBERT and 118.43% for GraphCodeBERT. The lower exact-match accuracy despite having a high F1-score can be attributed to the strictness of the former metric. This suggests that NS-Slicer may have mis-predicted only a few statements per program.

We performed an in-depth analysis of the test Java programs to gain more insights about this behavior. On average, 42.86% of the statements in the forward slice for each program contains references to the synthetic variable `aliasingVar`. Upon stratifying the test set based on such references, compared to the forward slices, we did not observe any direct correlation between the number of references and our tool's predictive capabilities.

While the predicted forward slices still hold value, PLMs may encounter challenges in generating *exact-match slices* in such complex yet uncommon aliasing scenarios. This can be attributed to the lack of understanding of source code at the memory level, as the current pre-training tasks mainly focus on lexical aspects of source code. Thus, advancements in source code pre-training could further improve the performance of PLMs in program slicing.

RQ. How does injection of Java aliases in programs affect NS-Slicer's performance?

RA. The PLMs in NS-Slicer predict forward slices for programs containing variable aliases with an F1-score of 84.17%–85.32%, and exact match accuracy of 34.82%–38.74%.

Case Study I. The self-attention (Vaswani et al., 2017) mechanism in PLMs facilitates contextualization by assigning attention scores to each token based on its relationship with

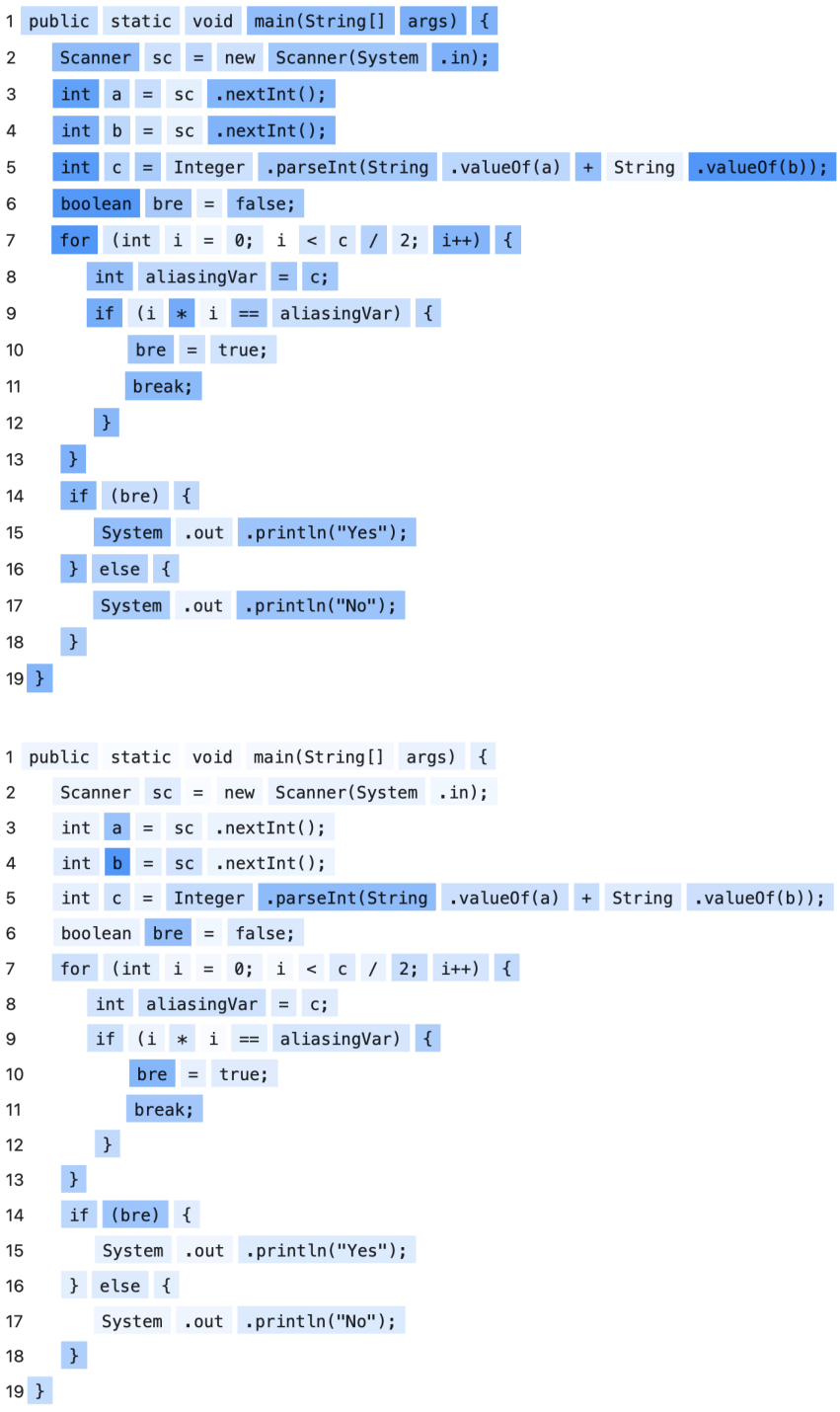


Figure 4.9: What does NS-Slicer learn? Visualization of attention score heatmaps from *pre-trained* (*top*) and *fine-tuned* (*bottom*) GraphCodeBERT PLMs within NS-Slicer, for all words in a Java program, sliced with respect to variable `c` on line 7.

all the other tokens within its context. Thus, for an input containing N tokens, we obtain an $N \times N$ attention map, where each attention score numerically signifies the importance of one token on the other. In this section, we use attention maps to investigate the GraphCodeBERT PLM's understanding of *variable-statement dependencies* within NS-Slicer.

In Figure 4.9, we present a test Java program, for which the static program slices need to be derived with respect to variable `c` on line 7. The ground-truth backward and forward slices for this slicing criterion include the lines $\{1, 2, 3, 4, 5\}$ and $\{8, 9, 11, 12, 13, 19\}$, respectively. To predict the static program slices, we utilize two versions of NS-Slicer from Section 4.4.1, utilizing the *pre-trained* GraphCodeBERT (row 2 in Table 4.1), and *fine-tuned* GraphCodeBERT (row 4 in Table 4.1). In Figure 4.9 (*top*) and (*bottom*), we visualize both models' attention via heatmaps, where each word (delimited by whitespace or periods) reflects its attention score with respect to the variable `c` on line 7. To compute these scores, we follow prior work (Clark et al., 2019) and take the mean of all such scores over its sub-tokens. In general, a darker colour in the heatmaps indicate a stronger relationship with variable `c`, in the form of *variable-statement dependencies*. Moreover, given that NS-Slicer is geared to predict whether a statement belongs to the program slice with respect to a variable in a requested criterion or not, the attention scores here suggest such decision making.

When using *pre-trained* GraphCodeBERT as in Figure 4.9 (*top*), we can see that the model does not pay attention to data and control dependent variables. As a result, the program slice computed by plugging pre-trained GraphCodeBERT model in NS-Slicer is not capable of predicting accurate backward and forward slices. Therefore, we can see that PLMs cannot directly be used for the program slicing task, reiterating the need to formulate program slicing task with distinct slicing classifiers as in NS-Slicer.

In contrast, *fine-tuned* GraphCodeBERT in NS-Slicer correctly predicts the backward and forward slice for this program. In Figure 4.9 (*bottom*), we can see that it pays more attention to variables `a` and `b` on line 5, and subsequently, also to corresponding variable

```
1    public static void main(String[] args) throws Exception {                ✓
2      Scanner sc = new Scanner(System.in);                                   ✓
3      int n = sc.nextInt();                                                  ✓
4      String ans = "Yes";                                                    ✗
5      int p_t = 0;                                                           ✓
6      int p_x = 0;                                                           ✗
7      int p_y = 0;                                                           ✗
8      for(int i = 0; i < n; i++){                                            ✓
9        int t = sc.nextInt();                                                ✓
10       int x = sc.nextInt();                                                ✓
11       int y = sc.nextInt();                                                ✓
12       int diff = Math.abs(x - p_x) + Math.abs(y - p_y);                    ✓
13       int aliasingVar = t;                                                 ✓
14       if(diff > aliasingVar - p_t || Math.abs(aliasingVar - p_t - diff) %
     2 == 1){                                                                 ✓
15         ans = "No";                                                        ✗
16         break;                                                             ✓
17       }                                                                    ✓
18       p_t = aliasingVar;                                                   ✓
19       p_x = x;                                                             ✗
20       p_y = y;                                                             ✗
21     }                                                                      ✓
22     System.out.println(ans);                                               ✗
23   }                                                                        ✓
```

Figure 4.10: What does NS-Slicer learn? Java program sliced with respect to variable `t` on line 9: ✓ denotes statements correctly identified by NS-Slicer as part of the slice, ✗ indicates statements erroneously classified as not belonging to the slice.

declarations on lines 3 and 4, respectively. Thus, it predicts the lines $\{3, 4, 5\}$ to belong to the backward slice. Applying a similar reasoning, we can also explain the prediction of line $\{2\}$ belonging to the backward slice. On line 8, the synthetic variable alias `aliasingVar` is introduced. As a result, lines $\{8, 9\}$ are predicted to belong to the forward slice. While lines 10 and 11 are control-dependent on line 9, we can see that variable `bre` does not depend on either `c` or the alias `aliasingVar` anywhere else. Thus, line $\{6\}$ and lines $\{14, 15, 16, 17\}$ are omitted from the backward and forward slices, respectively. Due to the control dependency described earlier, lines $\{11, 12\}$; and due to the nature of executable slices produced by JavaSlicer in ground truth, lines $\{13, 19\}$, are predicted to belong to forward slice. Therefore, we can see that NS-Slicer exhibits an understanding of the *variable-statement dependencies*.

Case Study II. Next, we present a test Java program in Figure 4.10 for which ground-truth static backward and forward slices with respect to variable `t` on line 9 are $\{1, 2, 3, 5, 6, 7, 8\}$ and $\{10, 11, 12, 13, 14, 16, 17, 18, 19, 20, 21, 23\}$, respectively. We can see that the variable `ans` depends on the conditions related to the variables `t`, `p_t`, `diff`, `x`, and `y`. However, it does not directly contribute to the computation of `t` on line 9. Thus, *fine-tuned* GraphCodeBERT (row 4 in Table 4.1) omits lines $\{4\}$ and $\{15, 22\}$ from the static backward and forward slices.

Furthermore, the `if`-condition on line 14 depends on the variable `diff`, which itself depends on the variables `p_x` and `p_y`. As a result, lines $\{6, 7\}$ and $\{19, 20\}$ should be included in the static backward and forward slices, respectively. However, with the introduction of the alias `aliasingVar` on line 13, NS-Slicer seems to miss this indirect dependency, causing it to omit these lines from its predicted slices. Notably, when the program is tested without the aliasing statement, NS-Slicer correctly identifies the relevant statements in the static slice, highlighting its current limitations in handling variable aliasing.

4.4.8 In-Depth Study of ND-Slicer's Performance: Statement Types

Analyzing 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 it understands the variables involved in conditions and how they influence branch selection or operations within branches. Likewise, with a loop, we can determine its ability to understand data/iteration dependencies within the loop, as well as the loop's exit conditions.

Procedure. To facilitate this experiment, we selected Python programs from Section 4.4.2, 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

Table 4.7: Evaluation of ND-Slicer on Python programs with different statement types

Approach	Statement Types	Evaluation Metrics (in %)			
		$R\text{-}L_P$	$R\text{-}L_R$	$R\text{-}L_F$	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

independent of loops. Moreover, in both (b) and (c), we ensured that slicing criterion is within the scope of these control structures. Here, we assess performance of the best-performing CodeExecutor+Transformer (from Section 4.4.2) while adopting the same evaluation metrics.

Empirical Results. In Table 4.7, we report the performance of ND-Slicer on Python programs with different statement types. We can see that in the case of branch-free programs, predictive slices with different 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 4.2), this is an improvement by 6.9%, reiterating ND-Slicer's capabilities in understanding *dynamic data dependencies*.

Next, consider instances with conditional statements. Here, 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 programs with conditional statements are more complex than without them. Overall, a high performance for these programs underscores ND-Slicer's proficiency in understanding dynamic control dependencies.

In the case of instances 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, when programs have `if-else` blocks within the 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. Therefore, the drop in performance by 5.4% relative to the overall performance of ND-Slicer can be attributed to program complexity.

4.4.9 In-Depth Study of ND-Slicer's Performance: Execution Iterations

The iterative and potentially complex nature of loops makes dynamic slicing particularly challenging, as it necessitates a precise understanding of control flow and iteration-dependent behaviors. Loops can introduce dependencies across execution iterations, tracking which is crucial for accurately predicting the dynamic slice. To this end, ND-Slicer encodes the execution iteration of the loop into its input representation. In this experiment, we assess its ability to predict dynamic slices for statements within loops, across execution iterations.

Procedure. For this experiment, we identified instances in our test set that contain a `for`-loop (denoted as D_{loop}). We then 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 groups, recording the same evaluation metrics as in Section 4.4.2. 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.

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 ground-truth dynamic backward slice for all occurrences

Table 4.8: Evaluation of ND-Slicer across execution iterations on Python programs: with `if-else` blocks (D_b), without `if-else` blocks ($D_{b'}$), and all examples (D_{loop}). Here, *EM* denotes Exact Match, and *R-L*$_x$ denotes ROUGE-LCS scores.

Split (→)	**w/ `if-else`** (D_b)				**w/o `if-else`** ($D_{b'}$)				**Overall** (D_{loop})			
Iteration (↓)	*R-L*$_P$	*R-L*$_R$	*R-L*$_F$	*EM*	*R-L*$_P$	*R-L*$_R$	*R-L*$_F$	*EM*	*R-L*$_P$	*R-L*$_R$	*R-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**

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, resulting in 1,090 instances in this experiment setting.

Empirical Results. In Table 4.8, we report results for these experiment settings. Overall, we observe a consistent decline in Exact Match Accuracy as the execution iteration o_i increases, reflecting ND-Slicer's increasing difficulty in maintaining accuracy as the iterations progress: from 81.7% to 70.0% as o_i goes from $1 \rightarrow 5$, and a further decline to 58.2% when $o_i \in [6, 10]$. We also observe a notable disparity in ND-Slicer's performance on loop examples with and without `if-else` blocks. This suggests that branching constructs introduce complexities, occasionally leading to incorrect decisions in the predicted dynamic slice, particularly for lower values of o_i. Interestingly, for a higher value of o_i ($o_i \in [6, 10]$), ND-Slicer performs slightly better on D_b than on $D_{b'}$. This hints that the errors in $D_{b'}$ may stem from difficulties in understanding intricate loop-exit conditions, which are worse for higher execution iterations.

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 nuances of execution iterations, `if-else` blocks, and ND-Slicer's predictive capabilities, offering valuable directions for future research and optimization.

4.4.10 In-Depth Study of ND-Slicer's Performance: Inter-Procedural Slicing

In this experiment, we delve into the performance of ND-Slicer in inter-procedural program slicing. Inter-procedural dynamic analysis, even for traditional techniques, poses a substantial challenge as it requires the comprehension of control flows that span multiple methods and depend on complex dynamic instrumentations. In 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 relationships in the context of inter-procedural dynamic slicing.

Procedure. To conduct this experiment, we curated Python programs from the original CodeNetMut dataset featuring inter-procedural calls. Notably, none of these were included in our intrinsic evaluation (in Section 4.4.2), as the PDG building tool, *python-graphs*, failed for these instances. As a result, ND-Slicer was exclusively trained in intra-procedural slicing scenarios, making this investigation into inter-procedural slicing particularly intriguing.

Consider a program P with methods M_1 and M_2, where M_1 calls M_2 at line x^{M_1}. Since the static PDG-building tool fails for P as a whole, we instead build a static PDG for M_1 in isolation. Based on execution history, we identify the set of lines $y_i^{M_1}$ in M_1 that exhibit dynamic data dependencies with x^{M_1}. We then use ND-Slicer to predict dynamic backward slices, treating the combined $\{x^{M_1}\} \cup \{y_i^{M_1}\}$ as the slicing criteria. Let S denote the resulting predictive slice. In total, we identified 579 such instances involving 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:

$$SLA = \frac{\sum_{P \in D} \begin{cases} 1 & |S_P \cap \{z | z \in M_2\}| \neq 0 \\ 0 & otherwise \end{cases}}{|D|} \tag{4.14}$$

$$HLA = \frac{\sum_{P \in D} \begin{cases} 1 & |S_P \cap \{r | r \in M_2, r = M_2^{return}\}| \neq 0 \\ 0 & otherwise \end{cases}}{|D|} \tag{4.15}$$

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

Empirical Results. In this experiment, we leveraged two best-performing models from Section 4.4.2 within the ND-Slicer framework to predict 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%, while 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 particularly promising, because both GraphCodeBERT (off-the-shelf and during fine-tuning) and CodeExecutor (during fine-tuning) were trained only on individual methods. This suggests that the models have acquired some abilities to generalize to inter-procedural execution.

While the lack of ground-truth backward slices poses a challenge for directly assessing inter-procedural predictive slices, an alternative strategy is to apply ND-Slicer independently for

M_1 and M_2 in P, with l and r as slicing criteria, respectively. The resulting slices (denoted by $S_P^{M_1}$ and $S_P^{M_2}$) can then be merged by inserting $S_P^{M_2}$ into $S_P^{M_1}$ following l, thereby forming a composite inter-procedural slice. Further empirical investigation is needed to assess this strategy. Nonetheless, ND-Slicer can scale to an inter-procedural setting, showing its ability to reason about complex control and data flows across method boundaries.

4.5 Concluding Remarks

In this chapter, we presented NS-Slicer and ND-Slicer, neural network-based static and dynamic slicing tools, respectively. These tools advance the capabilities of neural language models in understanding and reasoning about variable-statement dependencies in the context of both static and dynamic slicing, *i.e.*, settings involving, and not involving program inputs. We used pre-trained language models, each taking advantage of pre-training learning objectives such as masked language modeling toward understanding program semantics, and code execution prediction to internalize dynamic program behavior. These further enable NS-Slicer and ND-Slicer to produce highly accurate static and dynamic slices for complete programs, as well as extend such analyses to incomplete or non-executable programs. As we demonstrate, such generalization is valuable for tasks like vulnerability detection, and crash fault localization. Overall, our contributions not only demonstrate the promise of neural language models in fine-grained program analysis tasks but also introduce these tasks for evaluating the internal semantic knowledge representations of these models.

CHAPTER 5

TOWARD COMPREHENSIVE DEPENDENCE ANALYSIS OF PARTIAL PROGRAMS WITH LARGE LANGUAGE MODELS

5.1 Overview

As noted in Chapter 3, dependence analysis (DA) examines the relationships between different program elements. However, in many real-world scenarios, only partial programs (formally defined in Section 5.2.3) are available–whether due to modular development, privacy constraints, or being sourced from online forums like StackOverflow. In such cases, it is not possible for compiler-based dependence analysis approaches to completely disambiguate the syntactic constructs. As a result, they are rendered *ineffective* for partial programs.[1]

Chapters 3 and 4 explored the use of neural language models (NLMs) to directly learn to predict dependencies between program components at different levels of granularity. In this chapter, we present LλMDA, a *partial program dependence analysis* framework that combines large language models (LLMs) with traditional DA tools to construct a more precise and complete model of dependencies for partial programs. In doing so, we address the **Precision–Recall Conundrum** (illustrated in Figure 5.1), which we define as follows:

(♦) When applied to complete programs, classical DA tools such as (Joern, 2023) correctly identify most or all dependencies, achieving a *high precision and recall*.

(♦) On partial programs, DA tools prioritize caution (*i.e.*, soundness) and avoid making any assumptions. As a result, they often *miss* dependencies due to unresolved references or incomplete context, resulting in *high precision but lower recall*.

[1]The content presented in this chapter is based on the following research article:

Aashish Yadavally[*], Xiaokai Rong[*], and Tien N. Nguyen. 2025. "Large Language Model-Aided Partial Program Dependence Analysis". (* indicates equal contribution)

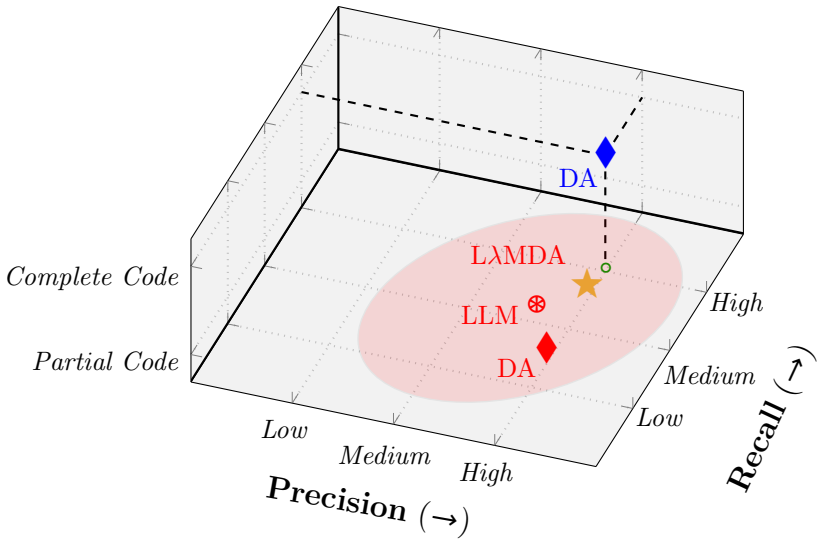


Figure 5.1: A theoretical framework representing the efficacy of program dependence analysis approaches for partial and complete code (highlighted in red and blue, respectively): without LLM (♦), and with LLM (⊛, ★). A high precision denotes the *correct* identification of dependence, and a high recall, the identification of *all dependencies* between program elements.

(⊛) Large language models can directly "predict" the dependencies between program elements (see Chapters 3 and 4). These rely on two key assumptions: (a) that missing type and context information in partial programs can be *implicitly* recovered in latent space through pre-training, and (b) that some degree of imprecision is acceptable. While such methods typically yield a *higher recall*, the lack of correctness guarantees leads to *lower precision*, making them less suitable in settings that require high reliability.

(★) To overcome the limitations of both standalone DA tools and LLM-only approaches, LλMDA combines the strengths of each: it uses LLMs as *context augmenters* to fill in missing code elements in a partial program, and then applies classical DA tools on the augmented code to recover dependencies. This two-phase design allows LλMDA to preserve the reliability of static analysis while improving its coverage, achieving a better balance between precision and recall. With necessary information, ★ would theoretically converge to ◦, the spatial counterpart to applying DA tools on complete code.

```
1   if (codec != null) {
2     in = new LineReader(codec.createInputStream(fileIn),job);
3     end = Long.MAX_VALUE;
4   } else {
5     if ( start != 0 ) {
6       skipFirstLine = true;
7       --start;
8       fileIn.seek ( start );
9     }
10    in = new LineReader(fileIn,job);
11  }
12  if (skipFirstLine) {
13    start += in.readLine (new Text(),0,(int) Math.min((long) Integer.
      MAX_VALUE, end-start));
14  }
```

Figure 5.2: Motivating example for LλMDA: Incomplete code snippet from StackOverflow post #16180130 copied from the `LineRecordReader` class in the Hadoop project on GitHub.

In addition to validating the theoretical framework illustrated in Figure 5.1 on partial program benchmarks (Phan et al., 2018; Saifullah et al., 2019), we also demonstrate LλMDA's generality and utility through two downstream applications, discussed later in the chapter.

5.2 LλMDA: Large Language Model-Aided Program Dependence Analysis

5.2.1 Motivation

Developers frequently access online forums such as StackOverflow (S/O) for quick solutions to coding tasks, often using those online code examples. While such code reuse can accelerate development, it also introduces potential risks. For instance, these examples might be outdated (Ragkhitwetsagul et al., 2021), or possess vulnerabilities (Verdi et al., 2022), and may inadvertently migrate to a codebase. Thus, a thorough analysis of such "toxic" code snippets is crucial for maintaining the integrity and robustness of the target codebase.

In Figure 5.2, we illustrate one such code example from an answer for S/O post #16180130 (White, 2013). This was originally copied from the `LineRecordReader` class in the Hadoop project on Github, intending to explain the usage of the `FileSplit` object, which

requires an offset of `-1` in some use cases and not in others. Notably, this has subsequently been modified to better handle such offsets in the Hadoop project itself (not shown here), significantly impacting its behavior and usage requirements. Although this change was introduced several years ago, the code in the S/O post remains outdated. Alarmingly, this outdated snippet has since been adopted by multiple new projects (Ragkhitwetsagul et al., 2021), which may now encounter unexpected issues in their implementations.

Consistent with our findings in Section 3.2.1, we observed that (Joern, 2023) fails to effectively capture dependencies in the potentially vulnerability-inducing code snippet shown in Figure 5.2. To assess its correctness, we ran Joern on the corresponding complete program from the Hadoop project listed in Figure 5.4, and then pruned the resulting dependencies to retain only those pertaining to the original incomplete code snippet. Figure 5.3 illustrates the resulting PDG, marking the correctly identified edges with a $\rightarrow$, and the missed ones with `-↛`. While Joern achieves *high precision*, it exhibits *low recall*, missing edges involving variables `codec`, `filein`, `in` (missing variable declarations), as well as `LineReader` (unresolved APIs).

The missed dependencies illustrate the limitations of the traditional DA tools in effectively examining incomplete code snippets. In many cases, the original version of the project from which a code snippet was copied may no longer be available either, further depriving developers of the necessary context. As a result, their ability to accurately assess and mitigate risks associated with integrating potentially vulnerable code snippets is significantly hindered.

5.2.2 Key Ideas

In this context, we introduce LλMDA to provide better correctness and completeness guarantees in the dependence analysis of partial programs. A natural idea, based on this example, is to disambiguate the code snippet by completing the missing pieces around it. However, this is not straightforward. A naive approach might involve inserting stubs for unresolved symbols and addressing the syntactic errors in a best-effort, rule-based manner. While this

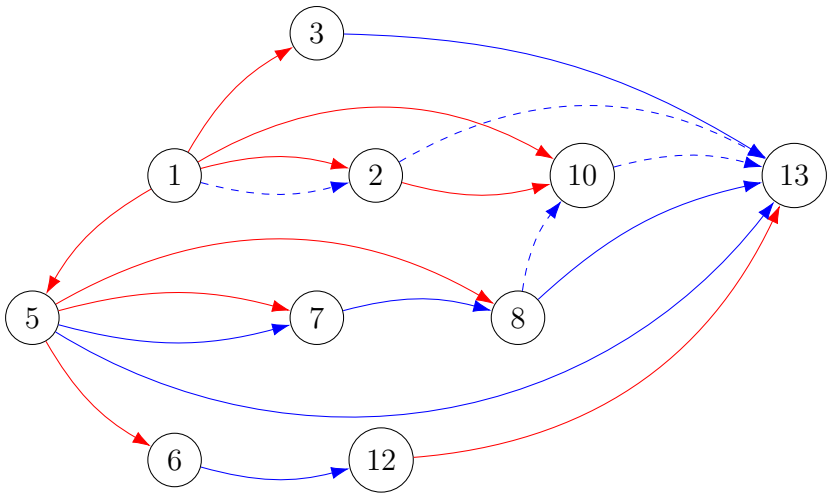


Figure 5.3: Augmenting a partial program with relevant context helps compiler-based dependence analysis tools retrieve missing dependencies (⇢ to →, for example in Figure 5.2). Here, control and data dependencies are highlighted in blue and red, respectively.

can potentially make an incomplete program compilable, lack of understanding of intended behaviors can introduce imprecision, potentially leading to false dependencies.

Instead of relying on hand-crafted heuristics, LLMs can leverage typical usage patterns and identifier cues to generate contextually-appropriate completions for unresolved symbols in partial programs. By inferring structures, types, and dependencies, LLMs can approximate the intended semantics, producing an *approximately-complete* program. These completions, however, can be highly project-specific with multiple valid implementations. For instance, the complete variant from Hadoop project (Figure 5.4) is just one candidate for the partial program in Figure 5.2. While LLMs can incorporate user intent in completion, *our focus is on the minimal contextual information necessary to enable effective dependence analysis.*

For illustration, LLMs can expand the code snippet in Figure 5.2 in multiple ways, including: (1) identifying possible data types for `fileIn`, `codec`, `job`, `start`, `end`, *etc.* in the partial program; (2) initializing these variables; (3) adding necessary import statements. Figure 5.5 shows an approximately-complete version, generated by GPT-4o, corresponding to the incomplete code snippet in Figure 5.2. While this version may not align exactly with a human-completed version, it provides sufficient structure for subsequent dependence analyses.

```
1  import java.io.IOException;
2  import java.io.InputStream;
3  import org.apache.hadoop.conf.Configuration;
4  import org.apache.hadoop.fs.FSDataInputStream;
5  import org.apache.hadoop.fs.FileSystem;
6  import org.apache.hadoop.fs.Path;
7  ...
8  public class LineRecordReader implements ... {
9    ...
10    public LineRecordReader(Configuration job, FileSplit split) throws
     IOException {
11      start = split.getStart();
12      end = start + split.getLength();
13      final Path file = split.getPath();
14      compressionCodecs = new CompressionCodecFactory(job);
15      final CompressionCodec codec = compressionCodecs.getCodec(file);
16      FileSystem fs = file.getFileSystem(job);
17      FSDataInputStream fileIn = fs.open(split.getPath());
18      boolean skipFirstLine = false;
19      if (codec != null) { // copied to StackOverflow (as shown in Figure 5.2)
20        in = new LineReader(codec.createInputStream(fileIn),job);
21        end = Long.MAX_VALUE;
22      } else {
23        if (start != 0) {
24          skipFirstLine = true;
25          --start;
26          fileIn.seek(start);
27        }
28        in = new LineReader(fileIn,job);
29      }
30      if (skipFirstLine) {
31        start += in.readLine(new Text(),0,(int) Math.min((long) Integer.
     MAX_VALUE,end-start));
32      }
33      this.pos = start;
```

Figure 5.4: A snapshot of `LineRecordReader` class in Hadoop project on GitHub, from where the code snippet in Figure 5.2 was copied.

Despite their ability in generating approximately-complete programs, LLMs do not guarantee syntactic or semantic correctness. This can result in errors due to incorrect types or APIs from external libraries. To address this issue, prior research has proposed using semantic verifiers, such as compilers, to ensure the correctness of generated code (Yasunaga and Liang, 2020; Chen et al., 2024). Similarly, we adopt a compiler for semantic verification,

```
1   import java.io.IOException;
2   import java.io.InputStream;
3   import org.apache.hadoop.conf.Configuration;
4   import org.apache.hadoop.fs.FSDataInputStream;
5   import org.apache.hadoop.fs.Path;
6   import org.apache.hadoop.io.Text;
7   import org.apache.hadoop.io.compress.CompressionCodec;
8   import org.apache.hadoop...CompressionCodecFactory;...
9   public class FileProcessing implements ... {
10    public processFile (Configuration job, String fName, long start,
      long end, boolean skipFirstLine, CompressionCodec codec) throws
      IOException {
11      FSDataInputStream fileIn = new FSDataInputStream(fName);
12      LineReader in;
13      if (codec != null) {
14        in = new LineReader(codec.createInputStream(fileIn),job);
15        end = Long.MAX_VALUE;
16      } else {
17      if (start != 0) {
18         skipFirstLine = true;
19          --start;
20         fileIn.seek(start);
21      }
22      in = new LineReader(fileIn,job);
23    }
24    if (skipFirstLine) {
25      start += in.readLine(new Text(),0,(int) Math.min((long) Integer.
      MAX_VALUE,end-start));
26    }
27    this.pos = start;
```

Figure 5.5: Complete code predicted by GPT-4o in LλMDA for incomplete code in Figure 5.2.

providing feedback signals to the LLM to iteratively refine approximately-complete programs (see Section 5.2.4). Alternative analyses may involve model checkers to verify properties, symbolic execution engines to explore multiple paths and identify potential runtime errors, runtime verification, test case execution, *etc.* to enforce domain-specific correctness.

Key Idea 5.1 (LLMs for Approximating Partial Programs)**.** *Leverage programming patterns learned during the expansive pre-training of LLMs to create a syntactically and semantically-valid, approximately-complete variant of the partial program.*

Figure 5.5 illustrates the approximately-complete variant generated by the LLM in LλMDA for the partial program in Figure 5.2. As seen, the LLM adds formal parameters for the undeclared variables `job`, `start`, `end`, `skipFirstLine`, and `codec`, along with the relevant `import` statements. Notably, it also declares `fileName` as a formal parameter and creates an instantiation of `FSDataInputStream` that takes `fileName` as an argument is initialized to `fileIn` at line 11. This statement is necessary to set up the API call `createInputStream` at line 14. In Figure 5.3, we show the PDG for the incomplete program (Figure 5.2). The dashed edges represent dependencies that were missed in the incomplete program but are present in the PDG for the complete version (Figure 5.4). However, when applied to the approximately-complete variant, Joern successfully recovers all relevant control and data dependencies, including those that were previously missed. Therefore, we can see that augmenting partial programs with missing context with LLMs helps achieve a *high recall*, while applying the DA tool to determine the dependencies helps achieve a *high precision*, even if the LLM's completions do not exactly match the human-written code.

Key Idea 5.2 (DA $\vdash$ High Precision, LLM $\vdash$ High Recall)**.** *Analyzing the syntactically and semantically valid, complete variant generated by an LLM for a partial program can help the* **precise** *retrieval of* **more** *(i.e., missed) dependencies.*

5.2.3 Important Concepts

Definition 5.1 (Partial Program)**.** *A partial program P is a syntactically valid, non-empty subset of an otherwise complete program (i.e., $P \subset P_C$). The incompleteness of P arises from the presence of unknown symbols S within P (such as fields, methods, type expressions) that are originally defined in P_C (i.e., $S \in P_C$).*

The disambiguation of P entails augmenting it with additional context necessary for semantic completeness. This includes variable declarations, method signatures, class and interface definitions, import statements, try-catch blocks, type casts, assignment of appropriate data types for local variables, method return types, and class-level fields, as well as code hardening constructs that enable resolution of unknown symbols in P.

Definition 5.2 (Context)**.** *The context C for a partial program P comprises the additional program elements required to resolve all unknown symbols S within P and disambiguate it.*

Definition 5.3 (Approximately-Complete Program)**.** *An approximately-complete program P_{AC} is obtained by augmenting a partial program P with context C generated by an LLM in LλMDA, such that $P_{AC} = P + C$. This augmentation resolves all unknown symbols S within P, ensuring that P_{AC} is both syntactically and semantically valid.*

Note that P_{AC} (as obtained from LλMDA) may not perfectly match the human-completed version, as it is designed to be functionally complete, rather than to reflect the developer's original intent or constraints. Instead, P_{AC} provides sufficient context for the DA tool (*e.g.*, Joern) to recover the relevant program dependencies that would otherwise be missing in P.

5.2.4 Approach Overview

Figure 5.6 illustrates LλMDA for enabling an effective dependence analysis of partial programs with LLMs. In the first phase of LλMDA, an LLM is tasked with disambiguating a given partial program by augmenting the necessary context required to retrieve syntactically and semantically valid, complete variant (referred to as *approximately-complete programs*). The augmented context can include: (1) variable declarations, (2) type information, (3) method signatures and class definitions, (4) import statements, among other program constructs. To ensure its correctness, we guide and validate the LLM in iterative cycles of

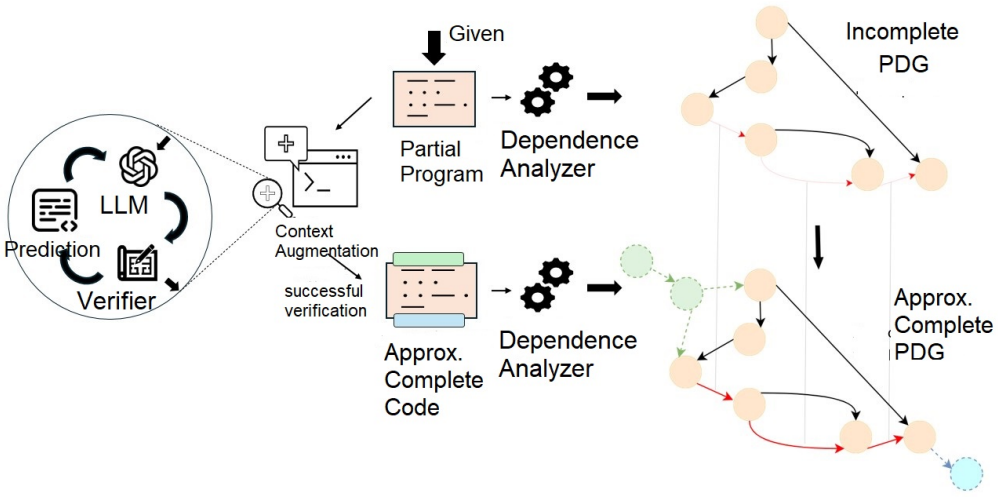


Figure 5.6: An overview of predictive dependence analysis framework with LλMDA. Here, augmenting the partial program with relevant context helps retrieve missing dependencies (⇢ to →) with a dependence analysis tool such as Joern (Joern, 2023).

self-correction, providing a feedback signal from a compiler. This iterative process continues until the approximately-complete program is compilable. If the code remains uncompilable after a specified number of iterations, manual intervention may be necessary. Note that depending on the requirements for the analysis, the compiler could be replaced with other semantic verifiers, *e.g.*, model checkers, type checkers, specification verifiers.

In the second phase of LλMDA, traditional dependence analysis (DA) tools are applied to the approximately-complete programs to retrieve dependencies for all program statements. Finally, these dependencies are pruned to retain only those relevant to the program statements present in the original partial program. With the additional context, it is possible for DA tools to identify dependencies between program elements that were originally missed due to ambiguities associated with the unknown symbols in the partial program (*i.e.*, *high recall*). Furthermore, by design, DA tools provide soundness guarantees (*i.e.*, *high precision*). As a result, LλMDA improves over applying traditional DA tools directly to partial programs in recall; and with other rule-driven or learning-based approaches in precision.

Prompt for Approximating Partial Programs

Task. Given a partial Java code snippet, compiler output/errors, and the expected outcome, your task is to generate the necessary code to complete the snippet. The additional code should address the issues indicated by the compiler output and achieve the desired outcome. Focus on enhancing the code header to make the snippet a compilable Java unit, without modifying the original code body.

Partial Code Snippet (Do Not Modify):
<code>

Compiler Output/Errors:
<errors>

Expected Outcome:
Using the information provided by the compiler, the model should fill in the missing information of the partial code snippet, focusing solely on enhancing the code header. The goal is to make the partial code snippet a compilable Java unit without modifying any part of the existing code body.

Your first task is to analyze the provided code, identify what are missing and complete the Java snippet. Do not modify the original code. Additionally, you do not need to write a method to solve missing API calls. After completing the code, your second task is to fill in type information.

Here is an example procedure of how you should construct answer. Suppose you are given a partial code snippet: <sample code snippet>

Here is what your response is supposed to look like:

Code Approximation:
```java
... // LLM approximated code
```

Type Information:
... // LLM proposed type information

Disclaimer:
Do not include any explanation or comments.

Figure 5.7: Prompt to LLM in LλMDA for approximating the partial program.

5.2.5 Context Augmentation: $P \rightarrow P_{AC}$

For a given partial program P, the objective is to generate a compilable variant P_{AC} such that no program elements within P are modified. To facilitate this process, we leverage an LLM $\mathcal{M}$ and a compiler $\mathcal{V}$ (as a semantic verifier), which work in tandem iteratively to populate the necessary missing information in P.

Approximating Partial Programs with LLM. We first check if the partial program P is compilable, collecting all its errors $\mathcal{V}(P)$ if it is not. Next, in the first pass to the LLM

(prompt shown in Figure 5.7), we provide the partial program P along with *all* compiler errors, and a set of instructions describing the task.

In the interest of making P compilable, the two main objectives of the LLM in this phase are (a) code approximation, and (b) type analysis. The first task requires the LLM to fill in essential program elements including necessary headers, import statements, method signatures, *etc.*, which are crucial for the input program's functionality. To drive this process, we instruct the LLM to inspect the compiler's error message, diagnose, and subsequently attempt to rectify these errors. This involves fixing syntax errors as well as disambiguating all unknown symbols. To further facilitate this process, we include the second task that requires the LLM to accurately resolve types and enumerate all variables and their associated types, *e.g.*, ⟨`fileIn`, `FSDataInputStream`⟩. We expect this analysis will help the LLM capture type dependencies intrinsically, which could facilitate a better identification of fully-qualified names for the unknown symbols. Finally, it yields a candidate approximately-complete program P^c_{AC} and the extracted type information $\mathcal{T}$. We adopt the few-shot setting with an exemplar, which serves as a guiding reference for the LLM by providing task-specific insights.

Refinement/Validation via Semantic Verifier. In this phase, we leverage a compiler as a semantic verifier due to its efficiency and cost-effectiveness. We begin by checking the compilability of the candidate approximately-complete program P^c_{AC}. If P^c_{AC} does not compile, a refinement process is initiated. Here, the LLM enters a corrective cycle, iteratively adjusting the program based on compiler feedback until it compiles successfully or reaches a predefined iteration threshold θ (in our experiments, we chose a maximum of 15 cycles).

First, we collect all errors generated during the failed compilation attempt. These might cover a range of syntactic and semantic issues. Some errors, however, are tied to external dependencies, such as "`package does not exist`" and "`class name and file name do not match`". The LLM fails to resolve these, so we discard them. Moreover, if a "`symbol not found`" error

Prompt for Incorporating Compiler Feedback for Self-Correction

Task. After incorporating the suggested code enhancements into the original code snippet, the code was recompiled, resulting in new compiler output/errors. Your task is to analyze these new errors, understand the context of both the original and modified code, and apply further modifications to correct the errors without altering the core functionality or logic of the original code.

Partial Code Snippet (Do Not Modify):
<original_code>

Modified Code with Errors:
<modified_code>

New Compiler Output/Errors:
<new_errors> (*i.e.*, after filtering the compiler's outputs)

Expected Outcome:
Code Approximation:
```java
... // LLM approximated code here.
```
Type Information:
... // LLM proposed type information here.

Here is an example procedure of how you should construct answer.
Same as in Figure 5.7

Disclaimer:
Do not include any explanation or comments.

Figure 5.8: Prompt to LLM in LλMDA for providing feedback for self-correction.

does not correspond to an API, such as in the case of an undefined constant, we also discard it. *Next*, we construct a feedback loop prompt (as illustrated in Figure 5.8) to guide the LLM in generating new P^c_{AC} candidates that address the errors identified by the compiler at the end of each iteration. The tasks and objectives of the LLM in this prompt are the same as earlier (Figure 5.7). In addition, we include the following key elements:

[1] original partial code snippet P

[2] LLM-generated P^c_{AC} candidate at the end of i-th iteration, *i.e.*, $P^{c(i)}_{AC}$

[3] filtered error messages from compiler

Here, it is important to include both the original and current candidates, especially for longer code snippets, as it helps the LLM avoid getting stuck in a cycle of repeated errors

and facilitates a more accurate understanding of the necessary corrections. *Finally*, the approximately-complete program P_{AC} is $P_{AC}^{c(i)}$ if there are no more compilation errors. In contrast, if threshold θ is reached with not all compilation errors getting resolved, it means that the LLM failed to disambiguate P, and might require manual intervention.

5.2.6 Problem Formulation

For a partial program P, let P_H represent a manually completed variant from human developers for on a specific use case, and let P_{AC} represent the approximately-complete variant generated by the LLM in LλMDA. Given a dependence analysis tool T, we define the dependence graphs produced for any program variant P_i as $G_i = T(P_i)$. Let $\tilde{G}_H$ and $\tilde{G}_{AC}$ denote the pruned versions of G_H and G_{AC}, respectively, such that they retain only the program elements originally present in P. We aim to show that LλMDA's design ensures:

$$Precision(\tilde{G}_{AC}, \tilde{G}_H) \text{ is } \uparrow\uparrow \tag{5.1}$$

$$Recall(\tilde{G}_{AC}, \tilde{G}_H) > Recall(G, \tilde{G}_H) \tag{5.2}$$

5.3 Empirical Evaluation

5.3.1 Effectiveness of LλMDA in Partial Program Dependence Analysis

In this experiment, we aim to evaluate how the filled-in information from LLMs in LλMDA helps the DA tool in better capturing dependencies in the original, incomplete code snippet.

Datasets. We selected two benchmark datasets from prior work, StatType-SO (Phan et al., 2018) and COSTER-SO (Saifullah et al., 2019). Both cover six Java libraries: `android`, `gwt`, `hibernate`, `joda-time`, `jdk`, and `xstream`. The authors of both benchmarks made these partial code snippets compilable by copying them into Eclipse, adding all required libraries in an iterative fashion and manually filling in the missing information. Finally, they used Eclipse

JDT compiler to validate the compilability of the code snippets. In the *Context Augmentation* phase in LλMDA, in simple terms, we aim to automate this process. Thus, we evaluated on the incomplete S/O code snippets, using the manually filled-in ones as the ground-truth. Overall, we collected 172 and 274 incomplete code snippets in those two datasets, respectively, each containing between 2–45 and 4–90 statements.

Procedure. We utilized the state-of-the-art Joern (Joern, 2023) as our DA tool. Our evaluation involved several baseline comparisons and approaches for generating PDGs from the incomplete code snippets ($D_{\text{incomplete}}$). *First*, we set up a traditional baseline (⧫) by creating a pseudo-syntactically valid variant for each incomplete code snippet by wrapping around them a dummy method signature (D_{dummy}). This modification was necessary as Joern requires syntactically valid inputs to construct PDGs. *Second*, we established learning-based baselines, NeuralPDA (Yadavally et al., 2023) (⊛.A) and PDBERT (Liu et al., 2024) (⊛.B), which were trained to "predict" the dependencies between program elements directly. *Third*, within the framework of LλMDA, we employed multiple LLMs including GPT-3.5, Claude-2.1, Claude-3.5 Sonnet, and GPT-4o (★.A–★.D) to obtain approximately-complete programs from the incomplete code snippets (D_{approx}). Finally, we used the manually-completed versions of these code examples (D_{complete}) to get the oracle dependencies.

For each dataset D_x ($x \in \{\text{dummy}, \text{approx}, \text{complete}\}$), we constructed the corresponding PDGs using Joern (*i.e.*, D_x+ Joern $\rightarrow G_x$). Next we sliced all G_x to select the sub-PDGs (G'_x) corresponding to the program statements in $D_{\text{incomplete}}$. By comparing such sub-PDGs from the partial and complete code snippets, *i.e.*, G'_{dummy} with G'_{complete}, we assess the impact of code incompleteness on Joern's ability to build the PDGs. Similarly, we quantify the impact of code approximation in LλMDA by the LLMs by comparing the sub-PDGs from the corresponding approximately-complete and complete code snippets, *i.e.*, G'_{approx} with G'_{complete}. For the learning-based approaches, we aggregated the predicted dependencies to construct the PDGs (G_{learning}) and compared with G'_{complete}.

Table 5.1: Effectiveness of LλMDA in partial program dependence analysis

Dataset (→) / **Approach (↓)**	**StatType-SO**			**COSTER-SO**		
	Evaluation Metrics (in %)					
	Precision	*Recall*	*F1-Score*	*Precision*	*Recall*	*F1-Score*
D_{dummy} + Joern (⧫)	99.0	69.8	81.9	96.2	48.5	64.5
NeuralPDA (⊛.A)	14.5	92.4	25.1	11.9	89.5	21.0
PDBERT (⊛.B)	80.5	95.4	87.3	74.9	81.1	78.1
LλMDA *w/* GPT-3.5 (★.A)	94.9	81.8	87.8	96.9	71.2	82.1
w/ Claude-2 (★.B)	96.1	84.2	89.7	98.3	61.9	75.9
w/ Claude 3.5 (★.C)	89.9	93.2	91.6	89.0	81.5	85.1
w/ GPT-4o (★.D)	95.5	88.1	**91.7**	99.2	83.1	**90.5**

Evaluation Metrics. We use Precision, Recall, and F1-Score to measure the quality of the PDGs. A True Positive (TP) occurs when an edge in $G \in \{G'_{\text{dummy}}, G'_{\text{approx}}, G_{\text{learning}}\}$ along with the associated nodes matches exactly with those in G'_{complete}. False Positives (FP) occur when: an edge in G between two nodes is determined, but does not exist in G'_{complete}; an edge in G between two nodes is determined, but one of the nodes does not exist in G'_{complete}. The latter is prevalent in G'_{approx}, as the LLM sometimes tends to modify some of the original program statements to harden the code. False Negatives (FN) occur when an edge is absent in G, but is present in the corresponding G'_{complete}. Formally, these metrics are defined as:

$$\begin{aligned} \text{Precision} &= \frac{\text{TP}}{\text{TP} + \text{FP}}, \\ \text{Recall} &= \frac{\text{TP}}{\text{TP} + \text{FN}}, \text{and} \\ \text{F1-Score} &= \frac{2 \times \text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}}. \end{aligned} \tag{5.3}$$

Empirical Results. In Table 5.1, we compare the PDGs constructed for the partial Java programs using LλMDA, with multiple LLMs (★.A–★.D), against traditional and learning-based approaches. In the case of the former, *i.e.*, when compared to PDGs constructed for partial programs wrapped around in dummy method signatures (⧫), we saw that LλMDA

improves in recall by 17.2%–33.5% and 27.6%–71.3% for the StatType-SO and COSTER-SO benchmarks, respectively. Interestingly, while there is a slight drop in precision by 3%–10.1% for the StatType-SO benchmark, in the case of COSTER-SO, it still goes up by 0.7%–3.1% (except for Claude 3.5 Sonnet). This difference in precision can possibly be attributed to the difference in lengths of the partial programs in StatType-SO and COSTER-SO, because with larger lengths, the LLMs contextualize the programs in COSTER-SO better towards the correct identification of unknown symbols, eventually resulting in better approximation. Moreover, the *precision for both benchmarks is high while achieving an improvement in recall*. Overall, LλMDA improves over the traditional approaches in F1-score by 7.2%–12% and 17.7%–40.3% for both benchmarks.

Next, among the learning-based approaches, we can see that NeuralPDA (⊛.A) shows a low performance in predicting program dependencies. This can be attributed to its lack of pretraining on a broader code corpus, relying instead on specialized training with a limited dataset. As a result, it lacks generalization capabilities. The notably high recall and low precision suggests a bias toward over-predicting dependencies between program elements. In contrast, PDBERT (⊛.B) demonstrates the advantages of pretraining, achieving the highest recall among all approaches. However, its moderate precision suggests a slight tendency toward over-predicting dependencies, likely a trade-off to maintain high recall. While the precision-recall tradeoff factors in with the smaller programs in StatType-SO, with slightly more context, the LLM-based framework in LλMDA outperforms PDBERT in both precision and recall. Overall, LλMDA improves over the learning-based approaches by 5%–265.3% and 15.9%–331% in F1-score for StatType-SO and COSTER-SO, respectively.

Among the LLMs employed in LλMDA, GPT-3.5 and Claude-2.1 demonstrate similar performance, as do Claude-3.5 Sonnet and GPT-4o. We can see that Claude-2.1 excels at contextualizing smaller programs, yielding a high precision and recall on StatType-SO; while GPT-3.5 is better at disambiguating unknown symbols in larger programs, as seen in its higher recall on COSTER-SO. We observe similar trends for GPT-4o and Claude-3.5 Sonnet.

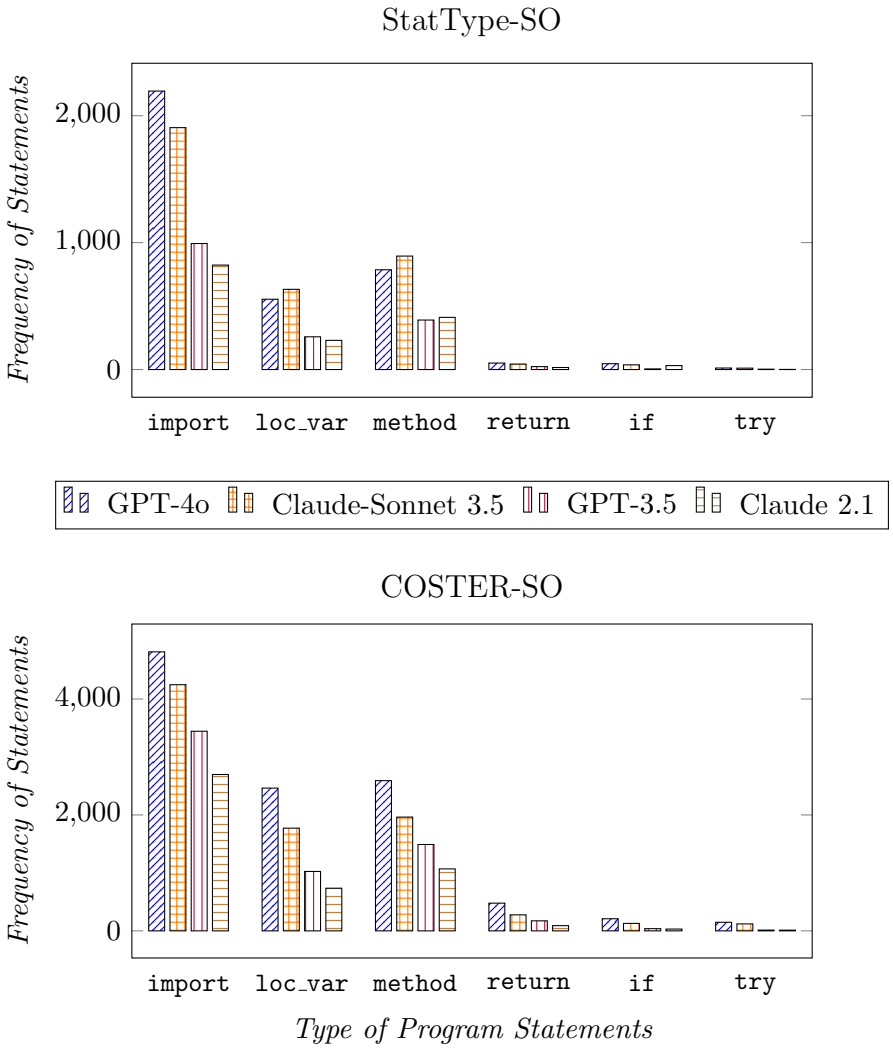


Figure 5.9: Distribution of different types of program statements filled-in by LLMs in LλMDA for disambiguating partial programs in StatType-SO and COSTER-SO benchmarks.

RQ. Can LλMDA enhance dependence analysis for partial programs?

RA. Our findings validate the framework presented in Figure 5.1, *i.e.*, LλMDA balances precision and recall by combining the strengths of DA tools and LLMs.

Qualitative Evaluation. In this experiment, we aim to assess the quality of code approximation by the LLM in the Context Augmentation phase in LλMDA. To this end, we analyzed the types of program statements filled-in for approximation, across iterations. In Figure 5.9, we

show the frequencies of these statements. Note that our code approximation prompt does not explicitly refer to such types when filling in the missing information. The LLM rather uses the feedback from the compiler in localizing and successfully fixing compilation errors. For example, missing import statements, local variable declarations typically trigger ‘‘`cannot find symbol`’’ errors at compile-time. These statements constitute 76.9% and 68.05% of the different types of statements populated by GPT-4o in the two datasets, respectively.

While filling-in the `import` declarations, we observed that LLMs identified appropriate fully-qualified names for the unknown symbols even though the code snippets lacked external libraries that could offer hints for such associations. This is different from when populating local variable declarations, which requires an understanding of more local type-specific dependencies. When coupled with compiler feedback containing unidentified symbols and their respective locations, *GPT-4o is able to gauge both external and local type dependencies.*

The LLM filled-in statements include *method declarations* in 20.7% and 22.5% of the partial programs in both datasets. Apart from the method signatures, these also include dummy methods that were added to resolve unresolved method calls. Introducing dummy methods within the same class does not affect the data dependencies within the method, but only helps in achieving syntactic validity. We also observed that GPT-4o attempted to harden code examples at various levels: *inserting a conditional that checks for corner cases* in 104 instances across both datasets (*e.g.*, protecting code from invalid input data such as `null` or an out-of-bound index), and *handling missing exceptions* in 21 cases.

GPT-4o and Claude 3.5 successfully generated approximately-complete code that passed the compiler for both datasets. In contrast, LλMDA with GPT-3.5 failed to approximate 10 programs in the COSTER-SO dataset, and Claude-2.1 failed 33 times. This difference may stem from the relatively worse coding abilities of GPT-3.5 and Claude-2.1, which likely limits their ability to resolve ambiguous or unresolved types. This aligns with our earlier finding that, while GPT-3.5 and Claude-2.1 handle smaller programs effectively, they struggle with longer code snippets, which may require more complex type resolution.

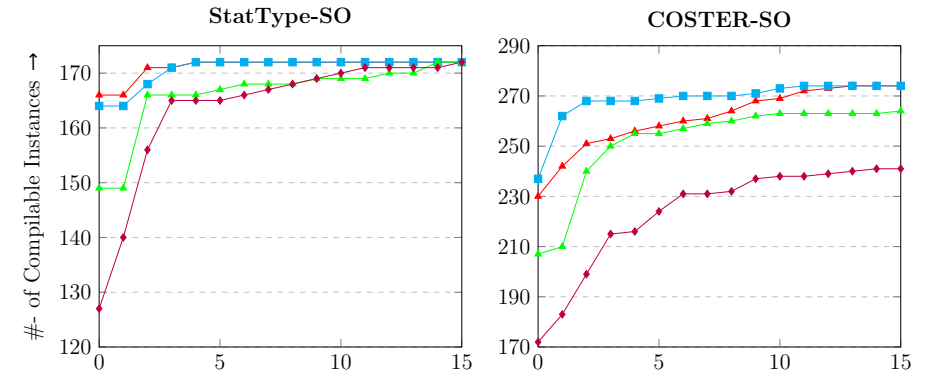
5.3.2 Sensitivity Analysis

In this section, we study how the approximately-complete code produced in LλMDA through iterative refinements, helps Joern recover more missing data dependencies.

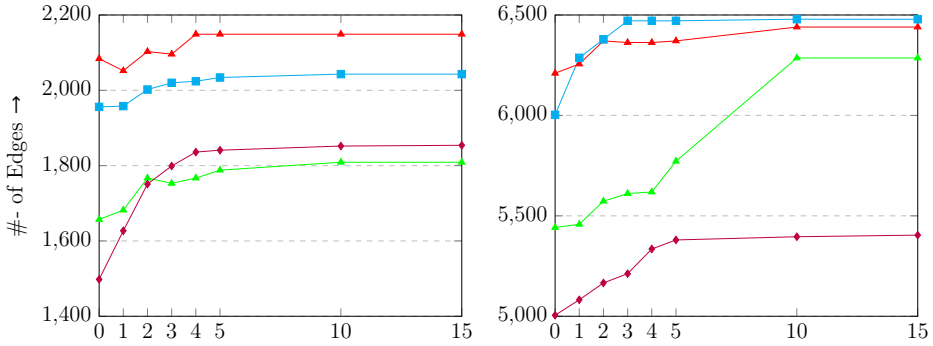
Procedure. Here, we utilized the approximately-complete programs from all LLMs within the LλMDA framework for both StatType-SO and COSTER-SO benchmarks. First, we quantified the interactions between the LLMs and the compiler by measuring *the number of iterations* required to transform the incomplete code into a valid, compilable version. Second, we evaluated the percentage of *correctly recovered data dependencies* from the ground truth after each refinement iteration in LλMDA. This process of recovering edges is illustrated in Figure 5.3. Third, we also measured the semantic correctness of the approximated code after each iteration, comparing the approximated version (D_{approx}) with the manually completed ones (D_{complete}) for the incomplete code snippets in both benchmarks. We used CodeBERT (Feng et al., 2020b) to obtain semantic representations of both versions and computed their cosine similarity, where higher scores indicate greater semantic equivalence.

Empirical Results.

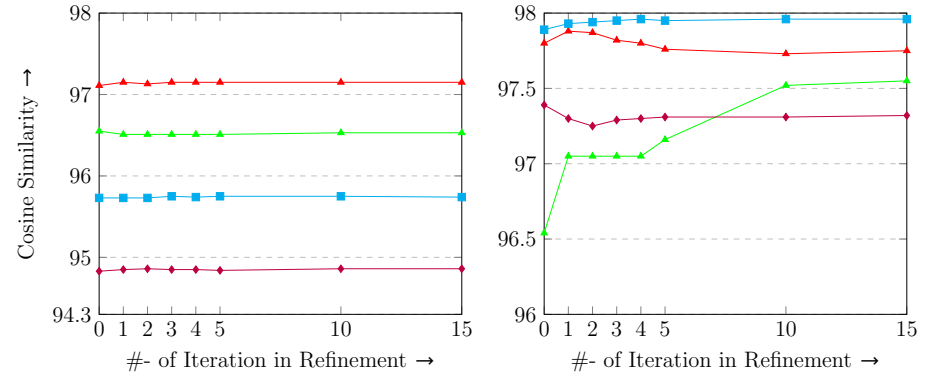
Sensitivity to Number of Refinement Iterations. Figure 5.10(a) shows the total number of instances in which the approximated code from LLMs becomes compilable by the end of a given feedback cycle. In the first iteration based on just the initial compiler feedback, LλMDA enables LLMs to generate compilable code snippets for **73.8%** of the StatType-SO instances (127 with Claude-2.1) and up to **96.5%** (166 with GPT-4o). For COSTER-SO, the first-iteration success rates range from **62.8%** (172 with Claude-2.1) to **83.9%** (230 with GPT-4o). This shows that LλMDA can produce compilable code using only the code approximation prompt and initial compiler feedback. Furthermore, we can see that additional refinement iterations guided by compiler feedback progressively reduce compilation errors.



(a) Successful compilation across refinement iterations



(b) Edges recovered across refinement iterations



(c) Semantic similarities across refinement iterations

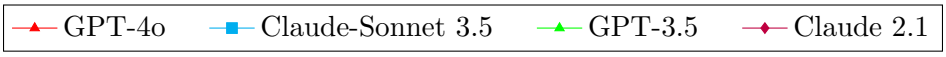


Figure 5.10: (a) Compilation rate, (b) Number of data dependence edges recovered, and (c) Semantic similarity of approximately-complete code with manually completed version, measured across refinement iterations within the compiler feedback-guided LλMDA framework.

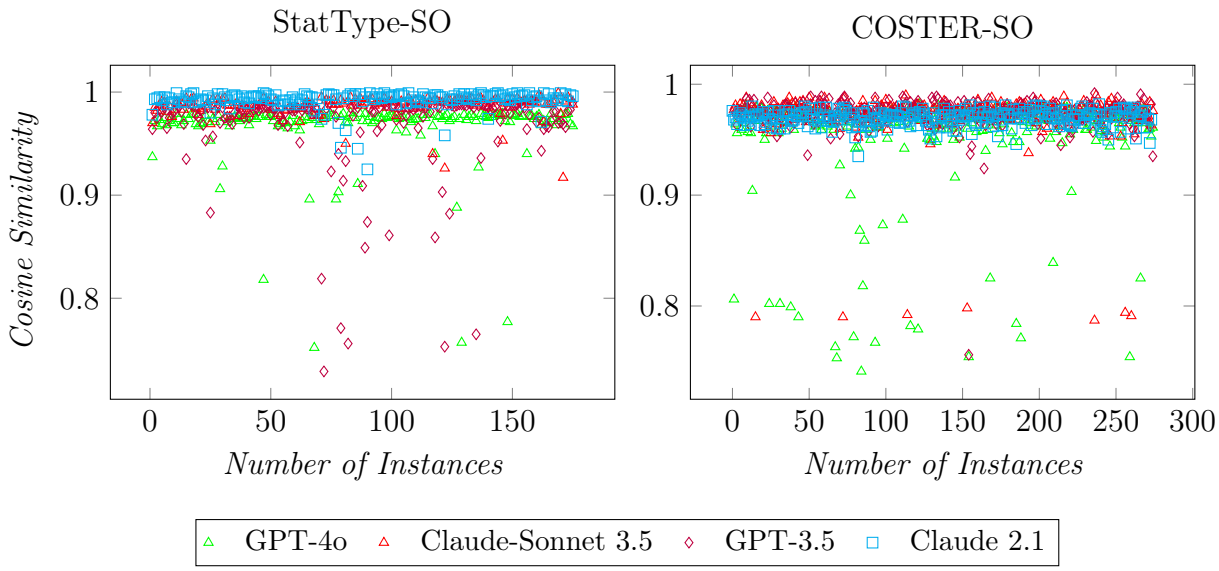


Figure 5.11: Cosine similarity between approximately-complete programs generated by LLMs and human-completed ground-truth for two benchmarks: StatType-SO and COSTER-SO.

Notably, for the StatType-SO benchmark, *nearly 98%* of code snippets become compilable after just two iterations with GPT-4o. Similarly, *97%* of code snippets in the COSTER-SO dataset compile successfully after only three iterations with Claude-3.5 Sonnet.

Recovering Missing Data Dependence Edges. Figure 5.10(b) shows the correlation between compilability and the number of data dependence edges recovered. As noted in Section 5.2.3, the approximately-complete programs from LLMs in LλMDA may not exactly match that from developers. However, *the data dependence edge recovery is tied to syntactic and semantic completeness*, as is reflected by *the progressively increasing number of data dependence edges recovered as the number of compilable approximately-complete programs increase*. With four iterations on the StatType-SO and two iterations on the COSTER-SO benchmarks, GPT-4o successfully recovered **96.5%** and **84%** of the missing edges, respectively. These findings reinforce our Key Ideas 5.1 and 5.2 that LLMs are useful to fill in missing statements to disambiguate partial programs and improve the recovery of data dependencies.

Semantic Correctness of Approximately-Complete Code. In Figure 5.10(c), we plot the mean cosine similarity scores between the approximately-complete programs generated by the

Table 5.2: Usefulness of LλMDA in exception flow analysis

Dataset	D_x + **Joern,** x =	**Evaluation Metrics (in %)**		
		Precision	*Recall*	*F1-Score*
StatType-SO	*approx-naive*	75.6	37.1	49.8
	approx (LλMDA)	**98.4**	**72.0**	**83.2**
COSTER-SO	*approx-naive*	73.9	29.3	42.0
	approx (LλMDA)	**100**	**86.6**	**88.3**

LLMs and their manually-completed counterparts in both the StatType-SO and COSTER-SO benchmarks across all refinement iterations. These scores are generally high, ranging between **0.94–0.98** across datasets, indicating strong semantic alignment. To provide finer-grained insights, Figure 5.11 presents a scatter plot of these similarity measures for individual instances. We observe that the outputs of GPT-4o and Claude 3.5 Sonnet are consistently close to the human-written completions, with cosine similarity approaching 1.

There are a few outliers when using GPT-3.5 and Claude-2.1, with lower cosine similarity measures. This correlates with instances where these models failed to produce compilable code, leading to greater divergence from the manually completed versions. However, *even these lower-matching completions still contribute to data dependence edge recovery, as indicated in Figure 5.10(b).* This result underscores that **LλMDA need not generate an exact match to human-written code to be effective**: as long as the added context is semantically meaningful, it can enable static analyzers to recover dependencies even in partial code. Overall, these findings reinforce Key Idea 5.1: semantic approximation, rather than exact reproduction, is sufficient for enabling precise dependence analysis in partial programs.

5.3.3 Usefulness of LλMDA in Exception-Flow Analysis

In this experiment, we aim to evaluate how the approximately-complete code from LλMDA helps Joern (Joern, 2023) in better capturing exception-flow edges in an Exception-Flow Graph (EFG). An EFG is part of the control flow graph (CFG) that connects to the exception

handling block(s). We used the same S/O datasets as in Section 5.3.1, focusing on instances containing `try/catch` blocks. We exclude those with only general exceptions (*e.g.*, `Throwable e`, `Exception e`), as these do not aid in identifying exception types. Overall, this yielded 23 and 38 code snippets from StatType-SO and COSTER-SO, respectively.

For our baseline, we prompted the LLM to make each code snippet compilable in a single step, without leveraging the iterative feedback loop used in LλMDA. We refer to this approach as *naive approximation*. Using this setup, each incomplete snippet from the filtered benchmark datasets was completed by the LLM to include exception-handling constructs, such as appropriate exception types and corresponding `catch` blocks. To evaluate exception-flow recovery, we used Joern to construct Exception Flow Graphs (EFGs). This involved first generating Control Flow Graphs (CFGs) for the approximated programs and then performing static slicing from the entry point of the code to the exception-handling blocks to retrieve the EFG edges. For the ground truth, we applied the same procedure to the manually-completed versions of the code snippets, yielding reference EFGs. We use the same evaluation procedure as in Section 5.3.1, substituting data dependence edges with exception-flow edges. As seen in Table 5.2, LλMDA helps construct the most comparable EFGs to the ones corresponding to manually completed code, achieving an F1-score of **83.2%** and **88.3%** on both datasets. Similar trends were observed as in the dependence analysis in Section 5.3.1: with LLM and refinement via compiler-guided feedback, LλMDA enhances both precision and recall.

5.3.4 Usefulness of LλMDA in Exception Handling Recommendations

Dataset, Procedure, and Evaluation Metrics. In this experiment, we assess whether LλMDA offers utility beyond dependence analysis. Specifically, we evaluate its effectiveness in the context of exception handling. To this end, we replace the dependence analysis component with one that identifies potential exceptions in incomplete code snippets. This allows us to examine whether the approximately-complete programs generated by LλMDA

Table 5.3: Usefulness of LλMDA in exception handling recommendations

Approach	Evaluation Metrics (in %)		
	Precision	*Recall*	*F1-Score*
PA-Neurex + $D_{\text{incomplete}}$	10.8	31.0	16.1
PA-Neurex + D_{approx} (LλMDA)	**15.7**	**36.7**	**22.0**
GPT-4o + $D_{\text{incomplete}}$	57.5	29.1	38.6
GPT-4o + D_{approx} (LλMDA)	**63.1**	**78.0**	**69.7**

can improve the accuracy of exception-handling suggestions. We use the same benchmark dataset introduced in Section 5.3.3 for consistency and comparability.

To replace the DA tool within LλMDA, we adopted the exception-handling recommendation tool developed as part of NeurEx (Cai et al., 2024), which we refer to as PA-NeurEx. Given a code snippet, PA-NeurEx recommends a list of exception types that should be handled; this list may be empty if no candidates are identified. The tool operates by examining each API element and the corresponding `import` statements in the snippet, attempting to resolve their types and map them to relevant exception types based on a database derived from libraries' documentation. As another baseline, we used GPT-4o (OpenAI, 2023b) with one-shot prompting as an alternative exception-handling recommendation tool. In this setup, we provide GPT-4o with either the incomplete or approximately-complete version of a code snippet and ask it to wrap potentially exception-throwing API calls in a `try-catch` block. We then extract the exception types from the generated code. We adopt the same evaluation metrics as used in NeurEx (Cai et al., 2024) for this task: Precision, Recall, and F1-score.

Empirical Results. Table 5.3 illustrates GPT-4o's performance in recommending exception types. When using LλMDA's approximated code instead of the original incomplete snippet, GPT-4o's recall improves by **168%**. This increase arises from the added type information in the approximated code, which enables GPT-4o to better determine which base Java packages to use. This substantially reduces incorrect exception predictions, improving the

precision by **9.7%**. Overall, as reflected by the F1-score, GPT-4o's performance in handling exceptions across both datasets improves by **80.6%**. In contrast, PA-NeurEx tends to favor completeness, which results in lower precision. With LλMDA's approximated code, PA-NeurEx benefits from the filled-in fully qualified API names, improving its precision by **45.4%** and recall by **18.4%**, leading to a **36.6%** improvement in F1-score.

5.4 Concluding Remarks

In this chapter, we introduced an alternative framework for partial program dependence analysis (compared to Chapters 3 and 4 that intrinsically learn the program dependencies) by integrating large language models with traditional semantic verification tools. In this case, the former is used to synthesize the plausible context required to disambiguate the unknown symbols and infer the missing information in incomplete programs. We demonstrated that LλMDA mitigates the precision-recall tradeoff in traditional DA tools by improving the recovery of dependence edges across multiple benchmarks comprising incomplete programs from StackOverflow. Furthermore, we also illustrated it's effectiveness in downstream applications such as exception-flow analysis and exception-handling, underscoring its practical value. Overall, LλMDA establishes a new framework to enable a range of static and dynamic analyses that require reasoning over under-specified partial programs.

CHAPTER 6

CHAIN-OF-THOUGHT REASONING ABOUT INTER-CONSTRAINT DEPENDENCIES IN PROGRAM PATH CONSTRAINTS

6.1 Overview

While Chapters 3—5 addressed limitations in traditional program analyses, such as their dependence on complete codebases, lack of generalizability, limited scalability, and reliance on runtime data collection and storage, this chapter turns to a different bottleneck: the challenge of scaling constraint-based analyses due to **state space explosion**.[1]

Several program analysis tasks are modeled as *constraint satisfaction problems*, including symbolic execution and automated test-case generation (Cadar et al., 2008; Godefroid et al., 2012), type checking, program verification (Barnett et al., 2005; Griggio, 2009; D'Silva et al., 2008), security (Backes et al., 2020), and optimization (Schkufza et al., 2016). In these settings, program states and their transitions are expressed as logical formulae, and Satisfiability Modulo Theories (SMT) solvers such as Z3 (de Moura and Bjørner, 2008), CVC4/5 (Liang et al., 2016; Barbosa et al., 2022), *etc.* are used as the back-end reasoning engines. For example, in symbolic execution, a program path is represented as a logical constraint, aiding a systematic exploration of all possible execution paths. In model checking, specifications of certain properties (*e.g.*, no `null` pointer is ever dereferenced) are expressed in a temporal logic, and the unreachability of the error state is verified.

To mitigate the computational cost of solving large constraint systems, machine learning (ML) has increasingly been used to either augment SMT solvers (Lagniez and Biere, 2013;

[1]The content presented in this chapter is based on the following publication (Yadavally et al., 2025):

> **Aashish Yadavally**, Xiaokai Rong, Phat Nguyen, and Tien N. Nguyen. 2025. "Large Language Models for Safe Minimization". In *47th IEEE/ACM International Conference on Software Engineering*, ICSE 2025, Ontario, Canada, April 27-May 3, 2025. IEEE, 2501–2513.

Wang et al., 2021; Selsam and Bjørner, 2019; Shi et al., 2023) or replace them altogether with neural solvers (Li et al., 2023; Selsam et al., 2019). With the reasoning capabilities of large language models (LLMs), it is natural to study to what extent they can aid SMT solvers.

In this chapter, we investigate their ability in *safe minimization* (formally defined in Section 6.2.1), which involves removing non-conflict causing constraints from an unsatisfiable constraint system, such that its size reduces from $n \rightarrow m$ ($m << n$) while preserving its unsatisfiability, subsequently reducing the constraint search space from $O(2^n) \rightarrow O(2^m)$. We refer to such subsets of the original constraint system as *smaller unsatisfiable subsets* (SUSes). A special case of SUSes is a minimal unsatisfiable subset (MUS), which is defined as an SUS in which the removal of any constraint makes it satisfiable.

Safe minimization to SUSes or MUSes is crucial in several tasks. For instance, it could enhance the analysis of inconsistencies in a program, which could stem from bugs, incorrect assumptions, *etc.*, and may lead to unexpected behaviors. Localizing the source of such inconsistencies can aid software engineers to effectively tackle these challenges. These subsets can be viewed as representing the *minimal reasons* or *explanations* for infeasible program paths, thus aiding program comprehension and debugging.

In brief, this chapter explores the central question:

Is it possible to safely minimize unsatisfiable constraint systems using LLMs, under the condition that no inconsistency-causing constraints are eliminated?

6.2 Safe Minimization of Unsatisfiable Constraint Systems

6.2.1 Illustrations and Concepts

Infeasible Constraint Systems. In Figure 6.1, we present the conjunctive normal form (CNF) of a string formula extracted from the source code for an interview question related to the *partition* problem LeetCode (lee, 2024), as documented in the SMT-LIB benchmarks (smt,

```
1   At(s, 1) == At(s, 5)
2   Not(At(s, 1) == At(s, 4))
3   Length(s) <= 8
4   Length(s) == 8
5   At(s, 5) == At(s, 6)
6   At(s, 4) == At(s, 7)
7   Not(At(s, 4) == At(s, 6))
8   Not(At(s, 4) == At(s, 5))
9   Not(At(s, 5) == At(s, 7))
10  Not(Length(s) <= 7)
11  Not(Length(s) == 7)
12  Not(At(s, 6) == At(s, 7))
13  Not(Length(s) <= 6)
14  Not(Length(s) == 6)
15  Not(Length(s) <= 5)
16  Not(Length(s) == 5)
17  Not(Length(s) <= 4)
18  Not(Length(s) == 4)
19  At(s, 1) == At(s, 3)
20  Not(At(s, 3) == At(s, 7))
21  Not(At(s, 3) == At(s, 6))
22  Not(At(s, 3) == At(s, 5))
23  Not(At(s, 3) == At(s, 4))
24  Not(Length(s) <= 3)
25  Not(Length(s) == 3)
26  At(s, 1) == At(s, 2)
27  Not(At(s, 2) == At(s, 7))
28  Not(At(s, 2) == At(s, 6))
29  Not(At(s, 2) == At(s, 5))
30  Not(At(s, 2) == At(s, 4))
31  At(s, 2) == At(s, 3)
32  Not(Length(s) <= 2)
33  Not(Length(s) == 2)
34  Not(Length(s) <= 1)
35  Not(Length(s) == 1)
36  Not(Length(s) <= 0)
37  Not(Length(s) == 0)
```

Figure 6.1: Minimizing unsatisfiable string constraint systems: A motivating example.

2024). This constraint system is unsatisfiable, and the constraint set $\{C_1, C_5, C_{19}, C_{21}\}$ (highlighted in orange) indicates one of its *minimal unsatisfiable subsets* (MUSes). The inconsistency is related to the string s, in which constraints C_1, C_5, and C_{19} assert that the characters at indices 1, 3, 5, and 6 are all the same, and C_{21} asserts that those at indices 3

and 6 can not be the same. Note that removing any of the constraints in the MUS will no longer preserve such a contradiction (thus, *minimal*).

For a constraint system $C = \{C_1, C_2, ..., C_n\}$ over a set of variables, if we can find an assignment to all variables such that each C_i is satisfiable, *i.e.*, all restrictions of every C_i are met by the corresponding assignments, we say that C is *satisfiable*. Otherwise, it is considered to be *unsatisfiable*, or *infeasible*. In this chapter, we focus on analyzing infeasible string constraint systems on the basis of the following concepts:

Definition 6.1 (Smaller Unsatisfiable Subset—SUS)**.** *A smaller unsatisfiable subset of an unsatisfiable constraint system C is a subset $S \subseteq C$ such that S is still unsatisfiable.*

Definition 6.2 (Safe Minimization)**.** *Given an unsatisfiable constraint system C, the identification of a subset S such that it is still unsatisfiable (i.e., S is an SUS) is referred to as safe minimization. Otherwise, if S is satisfiable, it is unsafe.*

Definition 6.3 (Minimal Unsatisfiable Subset—MUS)**.** *A minimal unsatisfiable subset of an unsatisfiable constraint system C is a subset $M \subseteq C$ such that M is still unsatisfiable and $\forall C_j \in M$, $M \backslash \{C_j\}$ is satisfiable. Note that the minimality in MUS refers to set minimality, and not to minimum cardinality.*

Consider the running example in Figure 6.1 to illustrate safe minimization. In this unsatisfiable constraint system, consider constraint C_4, which asserts that the length of the string `s` should be equal to 8. Next, consider pairs of constraints $\{C_{10}, C_{11}\}$, $\{C_{13}, C_{14}\}$, $\{C_{15}, C_{16}\}$, $\{C_{17}, C_{18}\}$, $\{C_{24}, C_{25}\}$, $\{C_{32}, C_{33}\}$, $\{C_{34}, C_{35}\}$, and $\{C_{36}, C_{37}\}$. Each of these resolves to `Length(s)>7`, `Length(s)>6`, `Length(s)>5`, `Length(s)>4`, `Length(s)>3`, `Length(s)>2`, `Length(s)>1`, and `Length(s)>0`, respectively. Furthermore, C_3 resolves to `Length(s)==8` in conjunction with C_4. Thus, including C_4 and eliminating the constraints C_3, $C_{10} - C_{11}$, $C_{13} - C_{18}$, $C_{24} - C_{25}$, and $C_{32} - C_{37}$ would not affect the unsatisfiability of the resulting set.

Consider constraint C_1 which asserts that the character at index 1 in string `s` is the same as that at index 5. Accordingly, constraint C_2, which originally asserts that the character at index 1 in `s` is not the same as at index 4, can be interpreted as the character at index 5 in `s` not being the same as that at index 4. We can see that this is the same as constraint C_8, which is repetitive. Thus, C_8 can be eliminated without affecting the unsatisfiability of the resulting set. Using a similar analogy between constraint sets $\{C_5, C_9, C_{12}\}$, $\{C_5, C_{21}, C_{22}\}$, $\{C_5, C_{28}, C_{29}\}$, $\{C_6, C_{20}, C_{23}\}$, $\{C_6, C_{27}, C_{30}\}$, and $\{C_{21}, C_{31}, C_{28}\}$, the constraints C_{12}, C_{22}, C_{29}, C_{23}, C_{30}, and C_{28}, respectively, can safely be eliminated as well. We refer to such reasoning on sub-groups of constraints in a constraint system as *macro-reasoning*. We posit that by exploiting *inter-constraint relationships* through macro-reasoning, we can identify redundant constraints from the input formula while preserving the inconsistency in the SUS.

Definition 6.4 (Macro-Reasoning)**.** *Macro-Reasoning is the reasoning on sub-groups of an unsatisfiable constraint system that exploits the inter-constraint relationships to identify and eliminate redundant constraints and enable safe minimization.*

6.2.2 Motivation and Key Ideas

Recently, LLMs have shown emergent behaviors with the abilities to reason in various domains, *e.g.*, arithmetic (Lewkowycz et al., 2022), formal logic (Morishita et al., 2023), source code (Touvron et al., 2023; Bubeck et al., 2023), or producing proof steps towards proof generation (First et al., 2023). In this chapter, we *aim to evaluate whether LLMs can be leveraged to decompose a constraint system and macro-reason on groups of constraints via tractable steps of thoughts towards identifying (inferred) redundant constraints.* That is, we model this as a *soft-search* problem, to see whether the LLM can be leveraged to explore the constraint space to approximately identify the constraints that do not contribute to an inconsistency, the remaining which, can be encapsulated as *the SUS*. This contrasts with traditional exhaustive search on the entire constraint space (Liffiton and Malik, 2013).

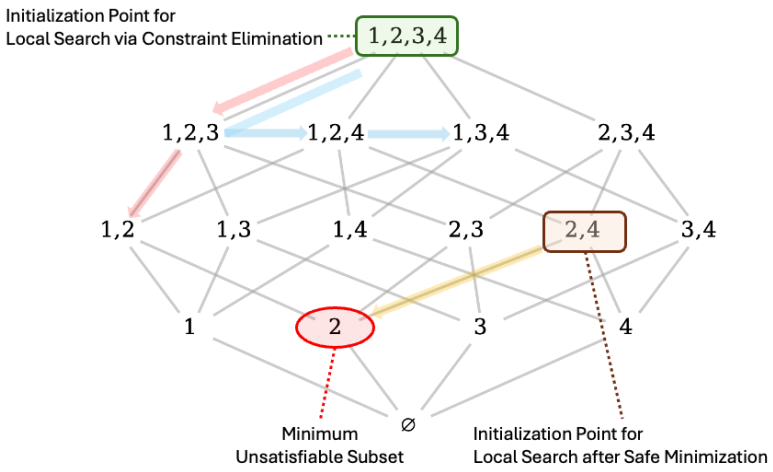


Figure 6.2: Hasse diagram of the power set lattice for a generic set of four constraints $C = \{1, 2, 3, 4\}$. Starting from $\{1, 2, 3, 4\}$, one can explore the constraint space via local search strategies such as depth-first (→) and breadth-first (→) search; compared to SafeMin's macro-reasoning driven approach.

Figure 6.2 illustrates the Hasse diagram for the power set lattice for a set of constraints $C = \{1, 2, 3, 4\}$, where the edges represent containment relations. Starting from $\{1, 2, 3, 4\}$, both breadth-first (BFS, →) and depth-first search (DFS, →) strategies exhaustively explore the state space by eliminating constraints from C and checking the satisfiability of each subset. In contrast, SafeMin reasons about the *inter-constraint relations* to get to $\{2, 4\}$ (*i.e.*, an SUS) without exhaustive search, which helps converge faster to $\{2\}$ (*i.e.*, MUS).

Exploring Diverse Reasoning Paths for a Parallelized, Partial Enumeration of Multiple Minimal Unsatistiable Subsets. While computing MUSes is an application of SUSes produced by our framework, a constraint system might have multiple MUSes. An MUS for the above constraint system is highlighted in Figure 6.1. However, this string formula has multiple MUSes, including constraint sets $\{C_1, C_5, C_{26}, C_{28}\}$, $\{C_1, C_{26}, C_{29}\}$, $\{C_1, C_5, C_{21}, C_{26}, C_{31}\}$, *etc.* Finding *all* MUSes is typically intractable with respect to completion, and a formula with n constraints can have an order of $O(2^n)$ MUSes. Thus, applications of MUSes

tend to relax the completeness criterion by not focusing on finding all of them, but depend on the number produced within a certain time limit (Zhao and Liffiton, 2016).

Complex reasoning tasks often have multiple reasoning paths. Wang *et al.* (Wang et al., 2022) make use of this diversity via a "sample-and-marginalize" decoding strategy in the LLM, where the optimal answer for a question is decided by marginalizing out the sampled reasoning paths and finding the most consistent one. Due to the nature of our task of minimizing constraint systems, we formulate it as a *soft search* problem. Thus, we propose a contrasting "sample-and-enumerate" decoding strategy. Here, the sampled reasoning paths can lead to multiple SUS candidates pertaining to non-unique MUSes. Such a design establishes a partial enumeration of MUSes. We call this strategy **self-exploration**, which encompasses the varied macro-reasoning perspectives of the LLM within a specific constraint system.

In theory, increasing the size of the sample during self-exploration results in a more complete enumeration of MUSes. Moreover, combining all candidate SUSes with independent SMT-verifiers (see Section 6.2.3) helps parallelize partial MUS enumeration with LLMs.

6.2.3 Our Approach

Our preliminary empirical findings (Section 6.3.1) revealed that LLMs exhibit macro-reasoning capabilities for the safe minimization of unsatisfiable constraint systems. However, their effectiveness remains limited by hallucinations and inability to support partial enumeration of multiple MUSes. To address these limitations and enhance the use of LLMs in safe minimization, we propose SafeMin, a learning-aided solving framework that employs an LLM and an SMT solver in tandem, to safely minimize a constraint system into its unsatisfiable subsets (SUSes). This section presents the SafeMin framework (Figure 6.3).

An infeasible constraint system $C = \{C_1, C_2, ..., C_n\}$ contains constraints, some of which contribute to its unsatisfiability, while others do not. The goal of safe minimization is to find subsets of C that are both *unsatisfiable* and have *fewer* constraints than the input formulae.

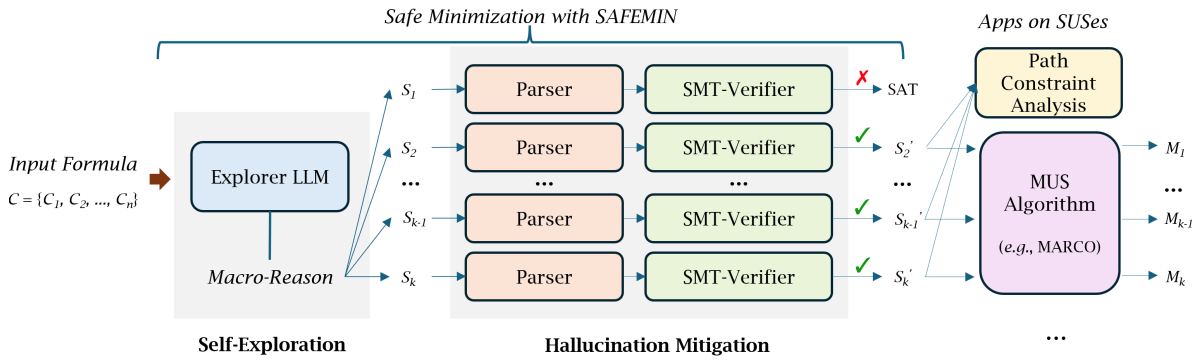


Figure 6.3: Overview of SafeMin framework

These subsets are referred to as the smaller unsatisfiable subsets (SUSes, $\mathcal{S}$). An SUS contains all constraints that lead to unsatisfiability in an MUS, and may additionally include a few that do not. We posit that this approximation is desirable, as the *imprecision* helps maintain *better correctness guarantees* in the form of unsatisfiability of the candidate SUSes.

SafeMin is designed to have the following key phases:

Self-Exploration. The goal, here, is to explore the constraint space of a given constraint system and identify candidate SUSes. Accordingly, we refer to this process as *self-exploration*, wherein large language models (LLMs) are leveraged along two dimensions:

[1] Large Language Model as a Macro-Reasoning Agent. Inter-constraint relationships within MUSes encapsulate the underlying conflicts that lead to the unsatisfiability of a given constraint system. The primary role that the LLM serves is to capture such constraint interactions and decompose the constraint system into sub-groups of constraints. We surmize that reasoning on such sub-groups facilitates the identification of the root causes for inconsistencies, and subsequently, the SUSes. Furthermore, this process establishes tractable steps of thoughts for minimizing the constraint space.

In Figure 6.4, we present the prompt given to the LLM for this purpose, which primarily illustrates the *Instructions* for it to adopt a Chain-of-Thought (CoT) (Wei et al., 2022)

Prompt to Explorer LLM for Safe Minimization

You are playing the role of an SMT solver on strings.

Task. Given a list of constraints or clauses, your goal is to identify a subset of these such that it is still unsatisfiable, *i.e.*, contains conflicts resulting from inconsistencies or contradictions in the logical state space.

Follow these thought steps to find the conflicts or contradictions:
Step 1: Read and understand the given logical string formula to identify the variables, operators, and logical connectives used in the formula.
Step 2: Analyze dependencies both within, and between constraints. These can arise from string operations such as concatenation, length, position, substring extraction, etc., as well as the variables they share.
Step 3: Macro-reason on these constraints to find pairs or groups that can possibly be combined and resolved. It is okay for there to be overlap among the different groups of constraints. The motivation for this grouping, however, should be to find redundant constraints in the input, which can be inferred from combining parts of these groups. Note that such redundant constraints CAN be eliminated.
Step 4: Next, try to identify constraints or groups of constraints which also satisfy another constraint or group of constraints. In such a case, the latter constraint or group CAN be eliminated.
For steps 3 and 4, try to find as many such groups as possible.
Step 5: Try to identify pairs of constraints that directly conflict with each other. These need to definitely be part of the output subset. Note that such conflicts or inconsistencies WILL arise in the given example.
Think step by step.

Output Format:
After identifying all such constraints that can safely be removed, output a comma-separated list of all remaining clauses. Each clause should be inserted between <c> and </c> tags, and the output between <output> and </output> tags.

Disclaimer
Try to minimize input constraints as best as possible, while also ensuring output subset's unsatisfiability.

Figure 6.4: Prompt to Explorer LLM in SafeMin for safely minimizing infeasible constraints.

reasoning trace and decompose the input constraint system into constraint sub-groups, macro-reasoning about them; and the prescribed *Output Format* for the candidate SUSes. In our preliminary experiments, we observed that directing the LLM to output the entire list of constraints that belong to the candidate SUSes nudges it to recall the inferred knowledge from the macro-reasoning steps, as opposed to using special markers for identifying the constraints. In this way, the LLM plays the role of exploring the constraint space towards summarizing the conflicts in a constraint system in the form of their SUSes.

[2] Exploring Diverse Reasoning Paths. As noted earlier, it is possible for a constraint system to have multiple MUSes, *i.e.*, multiple sources of inconsistencies. In this context, another dimension of LLM operability is the *parallelized, partial enumeration of such non-unique MUSes*. To enable this process, we propose a *"sample-and-enumerate"* decoding strategy, where we first *sample* from the LLM's decoder to generate a diverse set of reasoning paths. While the basis of these paths is rooted in *macro-reasoning on the input formula*, each may be focused on a different source of inconsistency, thus leading the LLM to *explore* non-unique candidate SUSes. Subsequently, we can leverage these candidates to extract non-unique MUSes. Another advantage of this design is that it naturally facilitates parallelization, as candidate SUSes can be independently analyzed towards computing the MUSes. In theory, sufficiently increasing the sample size during self-exploration can result in a complete enumeration of all MUSes.

Correctness Guarantees. A key limitation of LLMs is their hallucination, *i.e.*, generating plausible, but non-sensical information. In our design, Explorer LLM can generate candidate SUSes that do not that do not belong to the input constraint system, or they themselves can possibly not be unsatisfiable (as is required by the definition of an SUS). To mitigate such hallucinations, we incorporate two forms of correctness verification:

[1] Constraint Validity. The first class of hallucinations is *orthographic*, as it is possible that a constraint in the candidate SUS could not belong to the input constraint set altogether. To mitigate this, we incorporate a *Parser* into our design, which maps the incorrect constraint to the closest constraint in the input formula using *edit distance* as a heuristic. We adopt this measure for such an "auto-correction" due to its lightweight nature. Note that this can be swapped with *embedding-based similarity metrics* as well, which is left for future work. Overall, this step ensures the validity of the candidate SUSes.

[2] Unsatisfiability Verification. The next class of hallucinations is *rationale*-specific, as it is possible for the LLM-generated candidate SUSes, which it deduces to be unsatisfiable via several macro-reasoning steps, is not actually so. In this case, we see two ways of ensuring correctness, by using an *SMT solver* to verify (a) the correctness of the macro-reasoning steps, (b) the unsatisfiability of the final generated candidate SUS. While the former ensures a robust generation where only valid macro-reasoning steps are considered to infer the SUS, it can require multiple calls to the solver and can be prohibitively expensive. Moreover, during our preliminary evaluation, we also observed that the LLM sometimes tends to self-correct through the course of generation. Thus, we chose to include an SMT verifier after the fact, *i.e.*, which checks whether the final generated candidate SUS is indeed unsatisfiable, thus ensuring the correctness of candidate SUSes.

Overall, the *Explorer LLM* (used for self-exploration), along with the *Parser* and *SMT Verifier* (for correctness verification) work together to ensure the safe minimization of unsatisfiable constraint systems, facilitating a parallelized enumeration of SUSes and MUSes.

6.2.4 Problem Formulation

For an unsatisfiable input constraint system $C = \{C_1, C_2, ..., C_n\}$, let $\mathscr{P}\{C\}$ be its power set, *i.e.*, $\mathscr{P}(C) = \{\phi, \{C_1\}, \{C_2\}, ..., \{C_n\}, \{C_1, C_2\}, ..., \{C_1, C_2, ..., C_n\}\}$. The Explorer LLM in *self-exploration* phase takes C as input, and returns multiple candidate SUSes $\cup_{i=1}^{k} S_i$, where $S_i \in \mathscr{P}(C)$ and k is its decoding sample size. Next, we check $\cup_{i=1}^{k} S_i$ for *correctness* to obtain $\cup_{i=1}^{l} S_i$ ($l \leq k$) such that, $S_i \in \mathscr{P}(C)$ (*i.e.*, from Parser) and S_i is indeed unsatisfiable (*i.e.*, from SMT-Verifier). Finally, MUSes can be extracted in parallel from $\cup_{i=1}^{l} S_i$.

6.3 Empirical Evaluation

6.3.1 Effectiveness in Safe Minimization of String Constraints

Data Collection. With the first objective being to evaluate the LLM's capabilities in safe minimization, we used the string benchmark set in SMT-COMP'23 (smt, 2024) that contains quantifier-free string formulae with constraints reasoning about string lengths (*i.e.*, QF_SLIA from QF_Strings division). In particular, we picked the *LeetCode* benchmark containing 2,666 string formulae obtained from several LeetCode interview questions. These cover a wide range of complex string equations, inequalities, regular expressions, and include string functions such as `str.indexof`, `str.substr`, and `str.at`.

For the ground truth, we used the state-of-the-art tool CVC5 (Barbosa et al., 2022) to solve these constraints. Of the 2,666, we identified a total of 774 unsatisfiable instances, disregarding the remaining 1,892 satisfiable ones. Next, we simplified the unsatisfiable instances algebraically via rewriting operations to extract corresponding lists of constraints. Finally, we stratified them based on the number of constraints in each formula, and split them equally into *validation* and *test* sets. We use the former for tuning the prompt in Explorer LLM (Figure 6.3), and report the final performance on the test split.

Procedure. In this experiment, we aim to assess how well the LLMs in SafeMin reduce the given unsatisfiable set of constraints to the corresponding SUS toward safe minimization. To this end, we adopted Chain-of-Thought (CoT) (Wei et al., 2022) prompting to assess the models' macro-reasoning capabilities. Then, we used Chain-of-Thought prompting with Self-Exploration (CoT-$\mathcal{SE}$), enabling them to explore diverse reasoning paths and capture multiple SUSes. We compare the performance of SafeMin when using both prompting strategies, with GPT-3.5, GPT-4, Gemini-1.5 Pro, and Claude 3.5 Sonnet.

Consider a constraint system C=$\{C_1, C_2, ..., C_n\}$ which has a power set lattice $\mathscr{P}(C) = \{\phi, \{C_1\}, \{C_2\}, ..., \{C_n\}, \{C_1, C_2\}, ..., \{C_1, C_2, ..., C_n\}\}$. As the baselines for our advanced

Table 6.1: Effectiveness evaluation on string constraints.

Approach (↓) **#-Constraints (→)**	**Evaluation Metrics**							
	0–10	10–20	20–30	30–40	40–50	*Total*	$m_{SUS,\,\mathcal{D}_u}$	$m_{SUS,\,\mathcal{D}}$
Random (*pass*@1)	15	39	18	42	14	128 (32.9%)	0.46	0.15
CoT *w/* GPT-3.5	14	48	27	34	7	130 (33.5%)	0.41	0.14
GPT-4	41	91	69	85	24	310 (79.9%)	0.61	0.48
Gemini-1.5 Pro	41	90	58	85	23	297 (76.5%)	0.72	0.58
Claude-3.5 Sonnet	40	84	73	99	28	**324 (83.5%)**	**0.81**	**0.78**
Random (*pass*@5)	39	87	52	97	27	302 (77.8%)	0.45	0.35
Depth-First (*max_visited*=5)	29	90	69	109	31	328 (84.5%)	0.05	0.05
Breadth-First (*max_visited*=5)	31	104	76	114	33	358 (92.3%)	0.06	0.05
CoT-$\mathcal{SE}$ *w/* GPT-3.5	38	90	56	101	22	307 (79.1%)	0.48	0.38
GPT-4	44	107	77	115	29	**372 (95.9%)**	0.70	0.67
Gemini-1.5 Pro	42	92	59	96	27	316 (81.4%)	0.76	0.64
Claude-3.5 Sonnet	45	97	75	117	32	366 (94.3%)	**0.83**	**0.82**

prompting in SafeMin, we considered state space exploration with systematic search-based, as well as random strategies. *First*, we compared the CoT-based approaches with a baseline that randomly selects an element from $\mathscr{P}(C)$ (*i.e.*, random pass@1). These are equivalent, since both check the unsatisfiability of the candidate SUS once. *Second*, to benchmark our CoT-$\mathcal{SE}$, we also established a baseline that randomly selects k elements from $\mathscr{P}(C)$ (*i.e.*, random pass@k), where k is the number of samples produced in CoT-$\mathcal{SE}$.

Finally, adapting the traditional, non-LLM-based MARCO algorithm (Liffiton et al., 2016), which employs systematic state space exploration to enumerate MUSes, we also established baselines that eliminate the constraints in the input constraint set C by exploring $\mathscr{P}(C)$ via breadth-first (BFS) and depth-first search (DFS), checking the unsatisfiability of the subsets in order to enumerate the SUSes. For a fair comparison with CoT-$\mathcal{SE}$, which checks for unsatisfiability of k candidate SUSes, we limit the traversal in BFS and DFS to a maximum of k nodes, involving a maximum of k checks of unsatisfiability. In Figure 6.2, we present an illustration of such search-based baselines. Also, given the computational cost of the unsatisfiability checks with an SMT solver, we set k as 5 in this chapter. In Table 6.1, we call these baselines Depth-First and Breadth-First (*max_visited*=5).

Evaluation Metrics. To intrinsically measure the models' performance in macro-reasoning for safe minimization, we define *Minimization Accuracy* (A_m), which measures the ratio of the number of instances in which the candidate SUSes are unsatisfiable to the total number of instances. If D represents the test set and D_u represents the set of instances in which the candidate SUSes are indeed unsatisfiable, mathematically, $A_m = \frac{|D_u|}{|D|}$. In the case of Chain-of-Thought prompting with Self-Exploration (*i.e.*, CoT-$\mathcal{SE}$) with multiple candidate SUSes, we consider prediction of even one unsatisfiable candidate SUS as a correct instance.

While A_m measures the proportion of instances in the dataset in which the input formula is safely minimized, it does not quantify the reduction in search space. Thus, we measure the quality of safe minimization of the constraint system C to the SUS S (*i.e.*, $C \rightarrow S$) with *Minimization Ratio*, which is defined as $m_{SUS} = \frac{|C|-|S|}{|C|}$. Finally, we define two aggregated variants of m: (a) that on the entire test set, *i.e.*, $m_{SUS,\,\mathcal{D}} = \frac{1}{|\mathcal{D}|}\sum_{\in\mathcal{D}} m_{SUS}$, and (b) that where candidate SUSes are unsatisfiable, $m_{SUS,\,\mathcal{D}_u} = \frac{1}{|\mathcal{D}_u|}\sum_{\in\mathcal{D}_u} m_{SUS}$.

Empirical Results. In Table 6.1, we report the performance of different variants of the models. Interestingly, when prompted *without self-exploration* from SafeMin, GPT-3.5 (row 2) performs slightly better than the naive approach (row 1), despite the latter only picking a random constraint subset from $\mathscr{P}(C)$ as the SUS. Upon further analyzing the failures, we discovered that there were parsing errors in 13.1% of the cases, *i.e.*, the model did not adhere to the prescribed output format due to which the SUS could not be retrieved. Moreover, in *52.6% of the cases, it determined satisfiable constraint subsets as unsatisfiable*, which we were able to reject with the SMT verifier. This shows the limitations of GPT-3.5 in following our instructions for computing the SUS.

In contrast, by prompting GPT-4 with the same instructions (*i.e.*, CoT, row 3), we observed an improvement in performance by 138.5%. This establishes a direct comparison between both GPT variants in their ability to follow our instructions and CoT steps. Such

improvement is 128.5% for Gemini-1.5 (row 4) and 149.2% for Claude-3.5 (row 5). In particular, there were *no* parsing errors with these LLMs, and the number of cases in which the model incorrectly determined satisfiable constraint subsets as unsatisfiable dropped to 18.6%, 19.3%, and 12.4% (rows 3–5).

SafeMin aims to address these limitations and enhance the LLM's ability to safely minimize constraint sets. Compared to the baseline with 5 random selections, we observed the improvements for SafeMin with GPT-4, Gemini-1.5 Pro and Claude-3.5 Sonnet from 4.6%–23.2% in A_m and 8.6%–91.4% in m_{SUS}. This improvement for SafeMin with GPT-3.5 over that random picker is smaller as GPT-3.5 is limited as seen.

We also compared SafeMin with the MARCO algorithm with systematic BFS and DFS searches. With the traversal limited to a maximum of 5 nodes, eliminating constraints to navigate the state space results in a limited exploration depth. Accordingly, we observed *low minimization ratios* for both baselines. However, the possibility of hitting a candidate SUS which is a superset of the MUS in the constraint state subspace is high. As a result, the search-based baselines achieve a high minimization accuracy. This trade-off shows the limitation of search-based strategies for safely minimizing constraint sets.

Next, we compare the performance of our *self-exploration* prompting with different LLMs (*i.e.*, rows 9–12). As seen, SafeMin improves performance over only-CoT in minimization accuracy by 136.1%, 20%, 6.4%, and 13% when using GPT-3.5, GPT-4, Gemini-1.5 Pro, and Claude-3.5 Sonnet, respectively. Moreover, all LLMs with CoT-$\mathcal{SE}$ outperform their non-$\mathcal{SE}$ variants across all intervals, when stratified on the number of constraints, demonstrating the effectiveness of CoT-$\mathcal{SE}$ in SafeMin. While SafeMin with GPT-4 and Claude-3.5 Sonnet achieve similar minimization accuracy, minimization ratio for the latter is 17.1% higher.

Performance on String Operations. Figure 6.5 shows the LLMs' performance with CoT-$\mathcal{SE}$ prompting on Top-5 most occurring SMT2 string operations. Among the instances containing `str.at` (string-character function), `str.concat` (string concatenation function), `str.len` (string

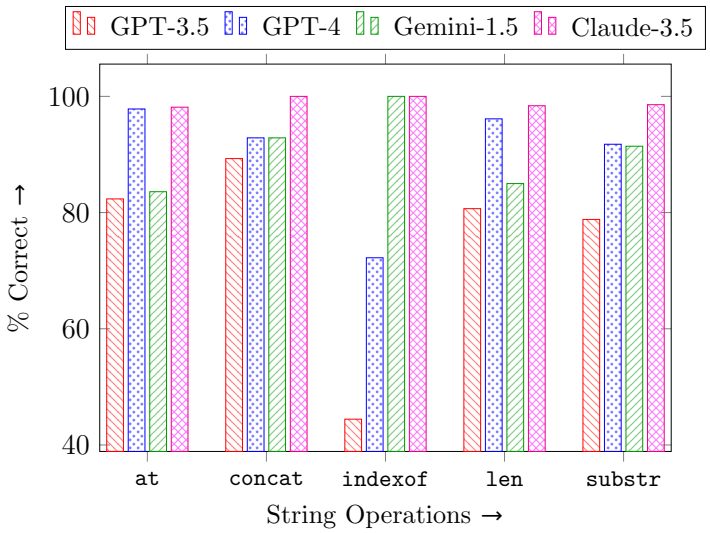


Figure 6.5: Performance comparison of CoT-$\mathcal{SE}$ in LLMs on the most-occurring string operations: `at`, `concat`, `indexof`, `len`, and `substr`.

length function), and `str.substr` (substring function), all LLMs make correct predictions in more than 80% of the cases (with GPT-4 and Claude-3.5 showing a better understanding of these operations). GPT-3.5 records a particularly low performance for `str.indexof` (returns the index of the first occurrence of the specified value), minimizing only 21% of these occurrences correctly. This can be attributed to the complexity of the `str.indexof` function, which often occurred in more complex constraints with other operations, which GPT-3.5 failed to resolve. In contrast, GPT-4 correctly predicts 72.2% of these instances, while Gemini-1.5 and Claude-3.5's correctly predict all of them. This result exhibits the improved macro-reasoning capabilities of the other LLMs over GPT-3.5.

Overall, with SafeMin's CoT-$\mathcal{SE}$ prompting strategy, we record the best performance with GPT-4 (row 10) and Claude-3.5 Sonnet (row 12), *safely minimizing the input formulae to SUSes in* ~ 95% *of the cases*. It outperforms the other baselines in minimization accuracy by 3.9%–190.6% and minimization ratio by 17.1%–1540%. In brief, we can see that SafeMin with GPT-4 or Claude-3.5 Sonnet, when prompted with CoT-$\mathcal{SE}$, was able to navigate the intricacies of the constraints and inter-constraint relations/inconsistencies.

RQ. How well do LLMs in SafeMin safely minimize string constraint systems?

RA. (1) GPT-4 and Claude-3.5 Sonnet demonstrate superior macro-reasoning capabilities in exploring inter-constraint relationships to achieve safe minimization.

(2) SafeMin with CoT-$\mathcal{SE}$ enhances the LLMs' macro-reasoning about sources of inconsistencies, enabling safe minimization of **~95%** of the string constraints.

Time Efficiency for Safe Minimization. We report the time for safe minimization taken by all LLMs for safely minimizing the input formulae when prompted with SafeMin's CoT-$\mathcal{SE}$ strategy. For GPT-3.5, the minimum observed processing time was 0.84 seconds, while the maximum recorded time was 142.3 seconds. The mean processing time for GPT-3.5 was 26.3 seconds (with a standard deviation of 31.8 seconds). Conversely, GPT-4 recorded minimum and maximum processing times of 1.13 and 111.3 seconds, respectively. The mean processing time for GPT-4 was 20.97 seconds (with a standard deviation of 25.87 seconds). For Claude-3.5 Sonnet, the times ranged from 0.64 seconds to 103.2 seconds, with a mean of 16.9 seconds and a standard deviation of 14.1 seconds. Similarly, Gemini-1.5 Pro recorded a minimum processing time of 0.72 seconds and a maximum of 122.6 seconds, with a mean of 22.1 seconds and a standard deviation of 21.2 seconds. While being subject to network latency, these metrics provide insights into scaling to real-world systems with LLMs.

6.3.2 Qualitative Analysis of Macro-Reasoning

In this experiment, we aim to assess the quality of macro-reasoning of LLMs used within SafeMin. Due to the effort involved in manually parsing through the LLM's textual responses, we randomly picked 5 examples from each of the input formulae sub-groups in Section 6.3.1 (*i.e.*, 0–10, 10–20, ..., 40–50), collecting a total of 25 instances and their corresponding LLM responses, when prompted with CoT-$\mathcal{SE}$. From our analysis, we could overarchingly identify the following *strengths* in all LLMs' macro-reasoning:

Table 6.2: Qualitative study on macro-reasoning of LLMs in SafeMin.

Weaknesses (→) LLM (↓)	W_1	W_2	W_3	W_4	W_5	W_6
GPT-3.5	5	4	5	7	4	2
GPT-4	4	1	1	–	–	–
Claude-3.5 Sonnet	2	2	1	–	3	–

(S_1) Able to pick contradicting logic pairs.

(S_2) Able to resolve intricate combinations of operations.

Other LLMs (not GPT-3.5):

(S_3) Able to combine constraints via transitive inference.

(S_4) Understands `str.substr` and `str.at` operations well.

(S_5) Able to combine and resolve combinations of operations, *e.g.*, the length of a substring (`str.len` and `str.substr`).

We observed the following categories of *inaccuracies* in all LLMs' macro-reasoning:

(W_1) Makes some logical errors when constraints involve `not` operation. For *e.g.*, GPT-3.5 resolved `not(a≤b)` as `a≤b`.

(W_2) Errors in combining constraints. For *e.g.*, GPT-3.5 resolved `(a==1) and not(a≤1)` as `not(a==1)`.

(W_3) Errors in understanding complex constraints. For *e.g.*, `str.len(str.substr(s, 0, 1))==1`

Only GPT-3.5:

(W_4) Tends to forget some of the previously inferred steps.

(W_5) Combines constraints based on incorrect reasoning.

(W_6) Fails to localize sources of inconsistencies.

```
Not(beginWord == endWord)
Not(Length(beginWord) <= 0)
endWord == Concat(str.substr(beginWord, 0, 2), Concat("t", str.substr(
 beginWord, 3, -3 + Length(beginWord))))
endWord == Concat(str.substr(beginWord, 0, 1), Concat("o", str.substr(
 beginWord, 2, -2 + Length(beginWord))))
endWord == Concat("d", str.substr(beginWord, 1, -1 + Length(beginWord)
 ))
beginWord == "dot"
Length(beginWord) >= 3
Length(beginWord) >= 2
Length(beginWord) >= 1
```

Figure 6.6: String formula safely minimized by GPT-4 with CoT-$\mathcal{SE}$ prompting.

Table 6.2 displays the counts of LLMs for all inaccuracies listed above (except Gemini-1.5, which only produced the output constraint subsets without any analyses, due to which we could not identify the weaknesses). Note that these counts are not mutually exclusive, *i.e.*, one instance can be counted multiple times across categories. Furthermore, in some cases, we noticed that GPT-4 and Claude-3.5 tend to ignore the prescribed thought steps, even more so than GPT-3.5. However, its superior performance is indicative of its adaptability during constraint resolution. Refer to (Yadavally, 2024) for more details.

As an illustration, consider the string formula in Figure 6.6. GPT-4, when prompted with CoT-$\mathcal{SE}$, was able to not only safely minimize it, but also accurately predict the corresponding MUS, *i.e.*, $\{C_1, C_4, C_6\}$. Notably, it was able to: (a) resolve constraints involving multiple variables (C_1 and C_6); (b) understand and analyze constraints involving intricate `Concat` and `str.substr` operations (C_3, C_4 and C_5).

We also tested the motivating example from Figure 6.1 with GPT-4 using CoT-$\mathcal{SE}$. Interestingly, we observed that GPT-4 was able to: (a) group all constraints about `Length(s)` to combine them into a single constraint `Length(s) == 8`; (b) reason about the transitive relationships between C_1, C_{26}, and C_{29}, to recognize that C_1 and C_{26} together would contradict C_{29}, since `At(s, 1) == At(s, 5)` and `At(s, 1) == At(s, 2)` would resolve to `At(s, 2) == At(s,`

5), which contradicts with its negation (C_{29}). These reasoning steps align with our hypotheses in Section 6.2.3, thus confirming the LLMs' capabilities in reasoning-based safe minimization.

RQ. How accurate is LLM's macro-reasoning on input string constraint systems?

RA. LLMs demonstrate superior macro-reasoning capabilities to reason about the inter-constraint inconsistencies and safely minimize infeasible constraint systems with a high accuracy, thus exhibiting a potential to scale.

6.3.3 Usefulness of SafeMin in Computing Minimal Unsatisfiable Subsets

In this study, we aim to explore the usefulness of SUSes predicted by SafeMin in the application of computing MUSes with the reduction of search space via safe minimization.

Data Collection. First, we leverage CVC5 to compute the MUSes for all constraint sets in the test set in Section 6.3.1. At its core, CVC5 uses either the constructed proof, or an assumption-based approach for MUS extraction. Due to its lightweight nature, we opt for the latter. Accordingly, for each instance in the test set, we establish $\langle C, \mathcal{S}, \mathcal{M} \rangle$ tuples, where, each corresponds to the input constraint set, candidate SUS from SafeMin, and the corresponding MUS computed by CVC5, respectively. Note that we can extract multiple $\mathcal{M}$'s for a C. In this experiment, we ensure that both C and $\mathcal{S}$ correspond to the same $\mathcal{M}$.

Procedure. We compare all LLMs with the best-performing CoT-$\mathcal{SE}$ prompting for SafeMin (in Section 6.3.1), as used for computing MUSes from the corresponding SUSes. While Section 6.3.1 measures the reduction in search space of the input formula, it does not measure this relative to the size of the MUS. To this effect, we define *minimization ratio* as $m_{MUS} = \frac{|C|-|\mathcal{S}|}{|C|-|\mathcal{M}|}$. By definition, m_{MUS} ranges from 0 to 1. If $m_{MUS} \to 0$, it means that Explorer LLM in SafeMin removed few or no constraints from C, and $\mathcal{S} \approx C$. On the contrary, if $m_{MUS} \to 1$, it indicates that Explorer LLM removed most of the non conflict-causing constraints from C, *i.e.*, $\mathcal{S} \approx \mathcal{M}$. We also define aggregated variants: $m_{MUS,\,\mathcal{D}}$ and $m_{MUS,\,\mathcal{D}_u}$.

Table 6.3: Usefulness of SUSes produced by SafeMin towards Computing MUSes.

#-Constraints (→) / Approach (↓)	Evaluation Metrics (*i.e.*, $m_{MUS,\,\mathcal{D}_i}$)											
	0 − 10		10 − 20		20 − 30		30 − 40		40 − 50		***Total***	
	$\mathcal{D}_u$	$\mathcal{D}$	$\mathcal{D}_u$	$\mathcal{D}$	$\mathcal{D}_u$	$\mathcal{D}$	$\mathcal{D}_u$	$\mathcal{D}$	$\mathcal{D}_u$	$\mathcal{D}$	$\mathcal{D}_u$	$\mathcal{D}$
CoT-$\mathcal{SE}$ *w/* GPT-3.5	0.65	0.53	0.56	0.46	0.63	0.44	0.54	0.46	0.39	0.26	0.57	0.45
GPT-4	0.84	0.79	0.82	0.79	0.86	0.83	0.81	0.79	0.73	0.65	0.82	0.79
Gemini-1.5 Pro	0.88	0.81	0.88	0.81	0.92	0.72	0.90	0.73	0.89	0.73	0.90	0.76
Claude-3.5 Sonnet	0.98	0.96	0.96	0.93	0.98	0.97	0.98	0.98	0.99	0.96	**0.98**	**0.96**

Empirical Results. In Table 6.3, we report the minimization ratios for all LLMs with SafeMin's CoT-$\mathcal{SE}$ prompting strategy. This illustrates the usefulness of the SUSes generated in Section 6.3.1, in the context of their corresponding MUSes. Overall, we can see that among the 366 safely minimized SUSes (as noted in Table 6.1), *Claude-3.5 Sonnet records an average minimization ratio with respect to their MUSes by 98%*, followed by Gemini-1.5 Pro at 90%, GPT-4 at 82%, and GPT-3.5 at 57%. That is, the SUSes generated by Claude-3.5 Sonnet are on average only 2% larger than the corresponding MUSes. When also including the 22 instances for which Claude-3.5 Sonnet could not safely minimize to the SUS (here, m=0), the average is 96%. For Gemini-1.5 Pro and GPT-4, these aggregated minimization ratio variants are 76% and 79%, respectively.

The breakdown in Table 6.3 on the basis of the number of constraints further sheds light on the quality of the SUSes. For instance, consider the string formulae containing 40–50 constraints. GPT-3.5 safely minimizes 66.7% of them (as noted in Table 6.3). The average minimization in these formulae is 39.4%, *i.e.*, the corresponding MUS computation involved starting from SUSes containing only 24–31 constraints. In contrast, GPT-4 safely minimized 87.9% of the formulae for an average minimization of 73.4%; Gemini-1.5 Pro, 81.8% for an average minimization of 89%; and Claude-3.5 Sonnet, 87.9% for an average minimization of 98%. Thus, with Claude-3.5 Sonnet, the MUS computation for these formulae involved SUSes containing only 7–9 constraints, while the average size of these MUSes is 3.2. In terms of the search space for MUSes, *this represents a reduction from* $O(2^{50}) \rightarrow O(2^{31})$ *with GPT-3.5, and* $O(2^{50}) \rightarrow O(2^{9})$ *with Claude-3.5 Sonnet.*

Furthermore, for 244 string formulae, we noticed that Claude-3.5 Sonnet with CoT-$\mathcal{SE}$ minimizes *exactly* to their MUSes (*i.e.*, $m = 1$). In comparison, GPT-4, Gemini-1.5 Pro, and GPT-3.5 do so only 73, 57, and 9 times. Upon further inspection, we observed that *for 80.9% of the instances in the* 0–10 *constraint range, Claude-3.5 Sonnet safely minimizes exactly to the MUSes.* Among the 10–20, 20–30, 30–40, and 40–50 constraint ranges, CoT-$\mathcal{SE}$ minimizes to the MUS via macro-reasoning for 69, 47, 72, and 18 instances, respectively.

RQ. How close are the SUSes predicted by SafeMin to the actual MUSes?

RA. SafeMin with Claude-3.5 Sonnet helps minimize unsatisfiable string constraints to SUSes that are only ~2% larger than the corresponding MUSes, reducing their search space from $O(2^{50}) \rightarrow O(2^{9})$ and localizing exactly to the MUSes in 62.9% of the cases.

6.3.4 Usefulness of SafeMin in the Parallelized, Partial Enumeration of Minimal Unsatisfiable Subsets

By design, SafeMin is able to *explore diverse reasoning paths to generate multiple SUS candidates via parallelized partial enumeration.* Thus, for a decoding sample size of k in the Explorer LLM, it can generate $O(k)$ SUSes. In this experiment, we use Claude-3.5 Sonnet as the LLM within SafeMin and assess its ability to identify multiple SUSes in parallel for a given constraint set, which highlights its usefulness in producing non-unique MUSes.

Procedure. Let us use $\mathcal{D}$ to denote our dataset of string formulae. The MUS computation from an input formula by using an SMT solver is deterministic, and exhibits a one-to-one correspondence. Accordingly, we compare the partial enumeration of MUSes by SafeMin along two dimensions, and establish loose lower and upper bounds for the total number of unique MUSes produced by the SMT solver as follows:

Our first experiment setting helps establish the benefits of using *self-exploration* in SafeMin toward partial enumeration. Specifically, for the lower bound, we extract MUSes

for each of the input string formulae ($C \rightarrow \mathcal{M}$) from the SMT solver, as well as the candidate SUSes (k = 5) generated by SafeMin for each formula ($\forall_{i \leq k} \mathcal{S}_i \rightarrow \mathcal{M}$). Thus, it represents the comparison between the sets $\cup_j(C \rightarrow \mathcal{M}_j)$ and $\cup_j(\cup_i^k(\mathcal{S}_i \rightarrow \mathcal{M}_{ij}))$, where j = 1..$|\mathcal{D}|$.

The second setting acts as a benchmark for SafeMin. Specifically, we mimic the partial enumeration of the MUSes from the original constraints C by using k unique random seeds within SMT solver, and collect the MUSes ($\forall_{i \leq k} C_i \rightarrow \mathcal{M}_i$). The basis for selecting those k seeds is to make the comparison consistent with the k candidates SUSes, and establish a loose upper bound for all possible unique MUSes (search space of $O(2^{|C|})$). Thus, it represents the comparison between $\cup_j(\cup_i^k(C_i \rightarrow \mathcal{M}_{ij}))$ and $\cup_j(\cup_i^k(\mathcal{S}_i \rightarrow \mathcal{M}_{ij}))$ where j=1..$|\mathcal{D}|$.

Empirical Results. First, we conducted an overlapping analysis of MUS enumeration, using the SMT solver directly on the input formula (*i.e.*, $C \rightarrow \mathcal{M}$), and by combining GPT-4 in CoT-$\mathcal{SE}$ with the solver (*i.e.*, $\mathcal{S} \rightarrow \mathcal{M}$). Of the MUSes, we can see that 246 are common to both approaches. Furthermore, SafeMin helps localize 277 additional MUSes that were not captured by the solver directly, and misses 120 that were computed deterministically.

Upon further analysis, we observed that SafeMin computes non-unique MUSes for 32.8% of the instances, resulting in it localizing 42.9% more MUSes than when the input formulae were solved directly. That is, in 32.8% of the instances, it explored different reasoning paths to successfully identify non-unique sources of inconsistencies. In the second setting, with multiple runs of the SMT solver on the input formula, we observed that it computes a total of 887 distinct MUSes. Of these, 385 were also computed by SafeMin, and 138 were computed by SafeMin but not the solver. Overall, SafeMin computes 58.9% of the total MUSes in the benchmark. Thus, our findings corroborate our design that *self-exploration* helps the LLMs exploit diverse reasoning paths, facilitating a parallelized enumeration of MUSes.

```
1 public static byte opaques(byte x, byte y) {
2     byte z;
3     if ((byte) (151 * (39 * ((x ^ y) + 2 * (x & y))
4         + 23) + 111) >(byte) ((x ^ y) + 2 * (x & y)))
5     {
6         ...//BLOCK1
7     }
8     else if
9     ((byte) (x - y + 2 * (~ x & y) - (x ^ y)) == 0x17)
10    {
11        ...//BLOCK2
12    }
13    else if
14    ((byte) (195 + 97 * x + 159 * y + 194 *
15       ~ (x |~ y) + 159 * (x ^ y) + (163 + x + 255 * y +
16       2 * ~ (x |~ y) + 255 * (x ^ y)) * (232 + 248 * x +
17       8 * y + 240 * ~ (x |~ y) + 8 * (x ^ y)) - 57) < 100)
18    { ...//BLOCK3
19    } else
20    { ...//BLOCK4
21    }
22    return z;
23 }
```

Figure 6.7: A case study on detecting infeasible paths in source code. Lines highlighted in *green* and *red* indicate feasible and infeasible paths, respectively.

RQ. How useful is SafeMin in using diverse reasoning paths in enumerating different MUSes for a constraint system?

RA. The SUSes from SafeMin helps SMT solver capture non-unique MUSes in 32.8% of the instances, thus computing 42.9% more MUSes than the original solver without SafeMin, and 58.9% of the total ones.

6.3.5 Usefulness of SafeMin in Detection of Infeasible Program Paths

In this section, we present a case study illustrating the application of SafeMin for *detecting infeasible paths in source code*. We selected a C code snippet from a collection of programs with known infeasible paths (i Montolio, 2023), and translated it into Java

(Figure 6.7). The feasible paths are highlighted in green and infeasible paths in red. After converting the paths into the SMT-Lib format, we used CVC5 to test the path constraints.

The constraint system derived from the program comprises three constraints, two of which, as in lines 3–4 and line 9, contain conflicts and are unsatisfiable, resulting in two distinct MUSes. We extracted all path constraints and input them to SafeMin, which successfully minimized them and directly returned the MUSes, thereby *identifying the infeasible paths*. This case study demonstrates how SafeMin can be used to aid developers in diagnosing unreachable code. By safely minimizing the constraint system to an MUS, SafeMin not only confirms unsatisfiability but also provides a concise and interpretable explanation, helping developers understand the root cause in terms of the minimal conflict-causing constraints.

6.4 Concluding Remarks

While the earlier chapters (3–5) addressed the challenges posed by partial programs in traditional program analyses, this chapter focused on a different bottleneck: state space explosion problem inherent in SMT solvers, which limits their scalability and applicability in large-scale software systems. To address this, we introduced SafeMin, a framework that leverages large language models (LLMs) to enable minimization of unsatisfiable constraint systems such that they remain unsatisfiable. We showed that SafeMin reduces the constraint search space in 94.3% of instances by an average of 98%, as well as its applicability in practical downstream tasks such as infeasible path detection. Overall, our findings illustrate the potential of LLMs to act as complementary macro-reasoners in constraint-based program analyses, aiding the scaling of formal methods in software engineering.

CHAPTER 7

CONCLUSION AND FUTURE WORK

Neural language models (NLMs) are increasingly being adopted in the field of software engineering as AI-in-the-loop processes, where software developers collaborate with NLMs to optimize different phases of the software development lifecycle (SDLC), or in human-in-the-loop settings, where the goal is to automate these phases altogether (Natarajan et al., 2025). Despite this, NLMs remain unreliable: they can excel at complex tasks such as bug detection, yet fail at simpler ones like generating nested loops. These inconsistencies often arise due to biases in their training data or sensitivity to model inputs. To better understand and quantify this uncertainty, which further limits the scalability and trustworthiness of AI-based coding tools, this dissertation investigates their ability to reason about intrinsic properties of source code. In particular, we used the formalisms of traditional program analysis tasks as a proxy for evaluating how well NLMs can understand different aspects of source code.

In Chapters 3 and 4, we considered settings in which the reasoning capabilities of NLMs are projected into the latent space, enabling them to implicitly model dependencies at different levels of granularity: inter-statement and variable-statement dependencies. In the case of the former, we analyzed control and data dependencies that form the basis of program dependence graphs. For the latter, we focused on identifying the parts of a program that may affect (*i.e.*, backward slicing) or are affected by (*i.e.* forward slicing) a variable at a specific point of interest, in both static and dynamic settings. In Chapter 5, we extended this reasoning to explicitly analyze dependencies in partial programs. In particular, we showed the abilities of NLMs to reason about type dependencies, expressed as verbalized reasoning traces, to infer the necessary missing information for disambiguating unknown symbols. Our proposed framework *precisely* predicted *more* dependencies in partial programs than traditional tools, improving both precision and recall. In Chapter 6, we broadened the scope of reasoning from local reasoning within a method (Chapters 3 and 4) and code fragment (Chapter 5) to

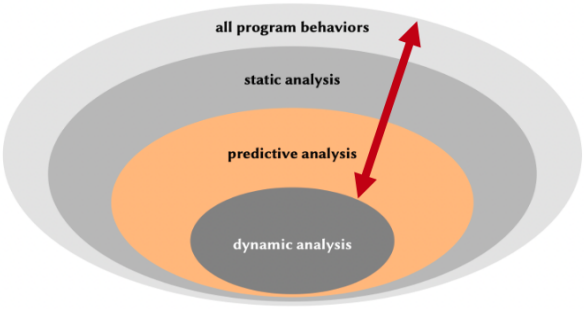


Figure 7.1: Hierarchy of program analysis techniques: *predictive analysis* encapsulates more program behaviors from history and bridges the gap between static and dynamic analyses. Note that its effectiveness typically scales with increased training data and model capacity.

global reasoning over complete programs. Here, we investigated whether NLMs can reason about logical dependencies between constraints that govern execution paths. In doing so, we addressed the state space explosion problem inherent in SMT solver-based program analyses, thus aiding the applicability and scalability of formal methods in software engineering.

As a consequence, this dissertation frames the reasoning capabilities of NLMs as a new paradigm of program analysis, *predictive program analysis*, in which NLMs are used to approximate and reason about program behaviors directly. Traditional static analysis often over-approximates behaviors and breaks down for programs with missing or incomplete information. Meanwhile, dynamic analysis, while generally more precise, requires access to full source code and runtime environments, making it impractical in many real-world settings. In contrast, the proposed predictive paradigm is a first step towards using the generalizability of NLMs to infer program behaviors without requiring complete codebases or relying on actual execution, thereby offering a scalable and execution-free alternative.

To conclude, this dissertation serves as a foundation for several future research directions:

[1] *Partial Program Analysis.* While this dissertation shows the ability of NLMs to approximate partial program behaviors, future research can further extend partial program analysis in several ways, including to handle contexts where only fragments of multiple files are available, thus enhancing its utility in several practical scenarios.

Building on the predictive slices generated in Chapter 4, future work could also explore generating natural language explanations and actionable feedback.

[2] *Static Estimation of Runtime Behaviors.* Chapter 4 introduces the notion of statically estimating dynamic behaviors, such as approximating dynamic slices, without relying on instrumentation or test execution. A promising direction for future work is to further unify static and dynamic reasoning in NLMs by incorporating analyses over execution traces, program inputs and outputs, updates to program states, thereby enabling models to internalize broader dimensions of program behaviors. These can be useful to statically detect runtime errors, identify unreachable code, or aid debugging.

[3] *Scalability and Efficiency.* Chapter 6 addresses the scalability challenge inherent in symbolic program analyses by mitigating the combinatorial explosion of constraint state spaces through macro-reasoning. Future research can build on this by exploring modular reasoning approaches, where NLMs can operate over constraints or program components at varying abstraction levels (*e.g.*, function- or module-level). Such decomposition may enable more scalable and interpretable analyses. In addition, these can facilitate the generalization of theory-specific solvers, improve runtime efficiency, and extend the applicability of NLM-aided symbolic analysis to more complex software systems.

[4] *Programming Language-Agnostic and Multilingual Analyses.* NLMs are inherently language-independent. This opens up new opportunities for analyzing modern software that comprise interactions between heterogeneous language components. In these settings, traditional tools fail due to their reliance on language-specific grammars and lack of cross-language understanding. Future research can use these capabilities to aid multilingual software development, from helping to choose the appropriate programming languages, generating multilingual codebases to capturing cross-language bugs and handling errors. In doing so, NLMs could thus serve as a unifying reasoning engine for analyzing and understanding programs that span multiple languages.

REFERENCES

(2022). A static analysis library for computing graph representations of python programs.

(2024). Leet code.

(2024). SMT-LIB Benchmarks.

Agrawal, H. and J. R. Horgan (1990). Dynamic program slicing. In *Proceedings of the ACM SIGPLAN 1990 Conference on Programming Language Design and Implementation*, PLDI '90, New York, NY, USA, pp. 246–256. Association for Computing Machinery.

Allamanis, M., E. T. Barr, S. Ducousso, and Z. Gao (2020). Typilus: Neural type hints. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2020, New York, NY, USA, pp. 91–105. Association for Computing Machinery.

Allamanis, M., M. Brockschmidt, and M. Khademi (2018). Learning to represent programs with graphs. In *ICLR*. OpenReview.net.

Allamanis, M. and C. Sutton (2013). Mining Source Code Repositories at Massive Scale using Language Modeling. In *The 10th Working Conference on Mining Software Repositories*, pp. 207–216. IEEE.

Andreasen, E., L. Gong, A. Møller, M. Pradel, M. Selakovic, K. Sen, and C.-A. Staicu (2017, sep). A survey of dynamic analysis and test generation for javascript. *ACM Comput. Surv. 50*(5).

Anthropic (2023). Claude model card. `https://www.anthropic.com/index/claude-model-card`.

Baah, G. K., A. Podgurski, and M. J. Harrold (2008). The probabilistic program dependence graph and its application to fault diagnosis. In *Proceedings of the 2008 International Symposium on Software Testing and Analysis*, ISSTA '08, New York, NY, USA, pp. 189–200. Association for Computing Machinery.

Backes, J., U. Berrueco, T. Bray, D. Brim, B. Cook, A. Gacek, R. Jhala, K. S. Luckow, S. McLaughlin, M. Menon, D. Peebles, U. Pugalia, N. Rungta, C. Schlesinger, A. Schodde, A. Tanuku, C. Varming, and D. Viswanathan (2020). Stratified abstraction of access control policies. In S. K. Lahiri and C. Wang (Eds.), *Computer Aided Verification - 32nd International Conference, CAV 2020, Los Angeles, CA, USA, July 21-24, 2020, Proceedings, Part I*, Volume 12224 of *Lecture Notes in Computer Science*, pp. 165–176. Springer.

Bahdanau, D., K. Cho, and Y. Bengio (2015). Neural machine translation by jointly learning to align and translate. In *ICLR*.

Barbosa, H., C. W. Barrett, M. Brain, G. Kremer, H. Lachnitt, M. Mann, A. Mohamed, M. Mohamed, A. Niemetz, A. Nötzli, A. Ozdemir, M. Preiner, A. Reynolds, Y. Sheng, C. Tinelli, and Y. Zohar (2022). cvc5: A versatile and industrial-strength SMT solver. In *TACAS (1)*, Volume 13243 of *Lecture Notes in Computer Science*, pp. 415–442. Springer.

Barnett, M., B. E. Chang, R. DeLine, B. Jacobs, and K. R. M. Leino (2005). Boogie: A modular reusable verifier for object-oriented programs. In F. S. de Boer, M. M. Bonsangue, S. Graf, and W. P. de Roever (Eds.), *Formal Methods for Components and Objects, 4th International Symposium, FMCO 2005, Amsterdam, The Netherlands, November 1-4, 2005, Revised Lectures*, Volume 4111 of *Lecture Notes in Computer Science*, pp. 364–387. Springer.

Belinkov, Y. (2022). Probing classifiers: Promises, shortcomings, and advances. *Comput. Linguistics 48*(1), 207–219.

Belov, A., M. Heule, and J. Marques-Silva (2014). MUS extraction using clausal proofs. In C. Sinz and U. Egly (Eds.), *Theory and Applications of Satisfiability Testing - SAT 2014 - 17th International Conference, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 14-17, 2014. Proceedings*, Volume 8561 of *Lecture Notes in Computer Science*, pp. 48–57. Springer.

Binkley, D. and K. Gallagher (1996). Program slicing. *Journal of Advanced Computing 43*, 1–50.

Binkley, D., N. Gold, M. Harman, S. Islam, J. Krinke, and S. Yoo (2014). ORBS: Language-independent program slicing. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pp. 109–120. ACM.

Bruening, D., T. Garnett, and S. Amarasinghe (2003). An infrastructure for adaptive dynamic optimization. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization*, CGO '03, USA, pp. 265–275. IEEE Computer Society.

Bubeck, S., V. Chandrasekaran, R. Eldan, J. Gehrke, E. Horvitz, E. Kamar, P. Lee, Y. T. Lee, Y. Li, S. M. Lundberg, H. Nori, H. Palangi, M. T. Ribeiro, and Y. Zhang (2023). Sparks of artificial general intelligence: Early experiments with GPT-4. *CoRR abs/2303.12712*.

Cadar, C., D. Dunbar, and D. R. Engler (2008). KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. In R. Draves and R. van Renesse (Eds.), *8th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2008, December 8-10, 2008, San Diego, California, USA, Proceedings*, pp. 209–224. USENIX Association.

Cai, Y., A. Yadavally, A. Mishra, G. Montejo, and T. Nguyen (2024). Programming assistant for exception handling with codebert. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, ICSE '24, New York, NY, USA. Association for Computing Machinery.

Campbell, J. C., A. Hindle, and J. N. Amaral (2014). Syntax errors just aren't natural: Improving error reporting with language models. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, MSR 2014, New York, NY, USA, pp. 252–261. Association for Computing Machinery.

Canfora, G., A. Cimitile, and A. D. Lucia (1998). Conditioned program slicing. *Inf. Soft. Technology 40*(11-12), 595–608.

Chakraborty, S., Y. Ding, M. Allamanis, and B. Ray (2020). Codit: Code editing with tree-based neural models. *IEEE Transactions on Software Engineering*.

Chen, D. and C. Manning (2014, October). A fast and accurate dependency parser using neural networks. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, Doha, Qatar, pp. 740–750. Association for Computational Linguistics.

Chen, J., C. Chen, J. Hu, J. Grundy, Y. Wang, T. Chen, and Z. Zheng (2024). Identifying smart contract security issues in code snippets from stack overflow. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2024, New York, NY, USA, pp. 1198–1210. Association for Computing Machinery.

Chen, X., M. Lin, N. Schärli, and D. Zhou (2024). Teaching large language models to self-debug. In *ICLR*. OpenReview.net.

Clark, K., U. Khandelwal, O. Levy, and C. D. Manning (2019, August). What does BERT look at? an analysis of BERT's attention. In *Proceedings of the 2019 ACL Workshop BlackboxNLP: Analyzing and Interpreting Neural Networks for NLP*, Florence, Italy, pp. 276–286. Association for Computational Linguistics.

Clark, K., M. Luong, Q. V. Le, and C. D. Manning (2020). ELECTRA: pre-training text encoders as discriminators rather than generators. In *ICLR*. OpenReview.net.

Conneau, A., G. Kruszewski, G. Lample, L. Barrault, and M. Baroni (2018). What you can cram into a single \$&!#* vector: Probing sentence embeddings for linguistic properties. In *ACL (1)*, pp. 2126–2136. Association for Computational Linguistics.

Dagenais, B. and L. Hendren (2008a). Enabling static analysis for partial java programs. In *Proceedings of the 23rd ACM SIGPLAN Conference on Object-Oriented Programming Systems Languages and Applications*, OOPSLA '08, New York, NY, USA, pp. 313–328. Association for Computing Machinery.

Dagenais, B. and L. Hendren (2008b). Enabling static analysis for partial java programs. In *ACM Sigplan Notices*, Volume 43, pp. 313–328. ACM.

de Lucia, A., A. R. Fasolino, and M. Munro (1996). Understanding function behaviors through program slicing. In *Proceedings of the 4th International Workshop on Program Comprehension*, pp. 9–18. IEEE Computer Society.

de Moura, L. M. and N. S. Bjørner (2008). Z3: an efficient SMT solver. In C. R. Ramakrishnan and J. Rehof (Eds.), *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*, Volume 4963 of *Lecture Notes in Computer Science*, pp. 337–340. Springer.

DeepMind, G. (2024). Gemini 1.5 technical report. `https://deepmind.google/technologies/gemini/`.

Devlin, J., M. Chang, K. Lee, and K. Toutanova (2019). BERT: pre-training of deep bidirectional transformers for language understanding. In *NAACL-HLT (1)*, pp. 4171–4186. Association for Computational Linguistics.

Ding, Y., B. Steenhoek, K. Pei, G. E. Kaiser, W. Le, and B. Ray (2024). TRACED: execution-aware pre-training for source code. In *Proceedings of the International Conference on Software Engineering (ICSE'24)*. ACM Press.

D'Silva, V. V., D. Kroening, and G. Weissenbacher (2008). A survey of automated techniques for formal software verification. *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst. 27*(7), 1165–1178.

Eclipse Foundation (2024). Eclipse integrated development environment (ide). Accessed: March 26, 2024.

Ernst, M. D. (2004). Invited talk static and dynamic analysis: synergy and duality. In C. Flanagan and A. Zeller (Eds.), *Proceedings of the 2004 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis For Software Tools and Engineering, PASTE'04, Washington, DC, USA, June 7-8, 2004*, pp. 35. ACM.

Fan, J., Y. Li, S. Wang, and T. N. Nguyen (2020a). A C/C++ code vulnerability dataset with code changes and CVE summaries. In *MSR*, pp. 508–512. ACM.

Fan, J., Y. Li, S. Wang, and T. N. Nguyen (2020b). A c/c++ code vulnerability dataset with code changes and cve summaries. In *Proceedings of the 17th International Conference on Mining Software Repositories*, MSR '20, New York, NY, USA, pp. 508–512. Association for Computing Machinery.

Feng, M. and R. Gupta (2010). Learning universal probabilistic models for fault localization. In *Proceedings of the 9th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, PASTE '10, New York, NY, USA, pp. 81–88. Association for Computing Machinery.

Feng, Z., D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang, and M. Zhou (2020a). Codebert: A pre-trained model for programming and natural languages. In *EMNLP (Findings)*, Volume EMNLP 2020 of *Findings of ACL*, pp. 1536–1547. Association for Computational Linguistics.

Feng, Z., D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang, and M. Zhou (2020b). Codebert: A pre-trained model for programming and natural languages.

Ferrante, J., K. J. Ottenstein, and J. D. Warren (1987, July). The program dependence graph and its use in optimization. *ACM Trans. Program. Lang. Syst. 9*(3), 319–349.

Field, J., G. Ramalingam, and F. Tip (1995). Parametric program slicing. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pp. 379–392. ACM.

First, E., M. N. Rabe, T. Ringer, and Y. Brun (2023). Baldur: Whole-proof generation and repair with large language models. In S. Chandra, K. Blincoe, and P. Tonella (Eds.), *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2023, San Francisco, CA, USA, December 3-9, 2023*, pp. 1229–1241. ACM.

Fisherman, A. (2015, April). Re: How to read command output line by line in c++ using 'popen'. Stack Overflow answer to "how to read command output line by line in gcc in windows just as with the standard input?". Accessed:June 13, 2025.

Francel, M. A. and S. Rugaber (2001). The value of slicing while debugging. *Sci. Comput. Program. 40*(2-3), 151–169.

Gabel, M. and Z. Su (2010). A study of the uniqueness of source code. In G. Roman and A. van der Hoek (Eds.), *Proceedings of the 18th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2010, Santa Fe, NM, USA, November 7-11, 2010*, pp. 147–156. ACM.

Galindo, C., S. Perez, and J. Silva (2022). A program slicer for java (tool paper). In *Software Engineering and Formal Methods: 20th International Conference, SEFM 2022, Berlin, Germany, September 26–30, 2022, Proceedings*, Berlin, Heidelberg, pp. 146–151. Springer-Verlag.

Gallagher, K., D. Binkley, and M. Harman (2006). Stop-list slicing. In *Proceedings of the Sixth IEEE International Workshop on Source Code Analysis and Manipulation*, pp. 11–20. IEEE Computer Society.

Godefroid, P., M. Y. Levin, and D. A. Molnar (2012). SAGE: whitebox fuzzing for security testing. *Commun. ACM 55*(3), 40–44.

Griggio, A. (2009). *An Effective SMT Engine for Formal Verification*. Ph. D. thesis, University of Trento, Italy.

Guo, D., S. Lu, N. Duan, Y. Wang, M. Zhou, and J. Yin (2022). Unixcoder: Unified cross-modal pre-training for code representation. In *ACL (1)*, pp. 7212–7225. Association for Computational Linguistics.

Guo, D., S. Ren, S. Lu, Z. Feng, D. Tang, S. LIU, L. Zhou, N. Duan, A. Svyatkovskiy, S. Fu, M. Tufano, S. K. Deng, C. Clement, D. Drain, N. Sundaresan, J. Yin, D. Jiang, and M. Zhou (2021). Graphcode{bert}: Pre-training code representations with data flow. In *International Conference on Learning Representations*.

Gupta, P., N. Mehrotra, and R. Purandare (2020). Jcoffee: Using compiler feedback to make partial code snippets compilable. In *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pp. 810–813.

Harman, M. and S. Danicic (1997). Amorphous program slicing. In *Proceedings of the 5th International Workshop on Program Comprehension*, pp. 70–79. IEEE Computer Society.

Harman, M., S. Danicic, Y. Sivagurunathan, and D. Simpson (1996). The next 700 slicing criteria. In *Proceedings of the 2nd U.K. Workshop on Program Comprehension*.

Harman, M., R. Hierons, C. Fox, S. Danicic, and J. Howroyd (2001). Pre/post conditioned slicing. In *Proceedings of the IEEE International Conference on Software Maintenance*, pp. 138–147. IEEE Computer Society.

Hatcliff, J., M. B. Dwyer, and H. Zheng (2000). Slicing software for model construction. *Higher Order Symbolic Computation 13*(4), 315–353.

Hernández López, J. A., M. Weyssow, J. S. Cuadrado, and H. Sahraoui (2023). Ast-probe: Recovering abstract syntax trees from hidden representations of pre-trained language models. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, ASE '22, New York, NY, USA. Association for Computing Machinery.

Hindle, A., E. T. Barr, M. Gabel, Z. Su, and P. T. Devanbu (2016). On the naturalness of software. *Commun. ACM 59*(5), 122–131.

Huang, Q., Z. Yuan, Z. Xing, X. Xu, L. Zhu, and Q. Lu (2022). Prompt-tuned code language model as a neural knowledge base for type inference in statically-typed partial code.

Husain, H., H. Wu, T. Gazit, M. Allamanis, and M. Brockschmidt (2019). Codesearchnet challenge: Evaluating the state of semantic code search. *CoRR abs/1909.09436*.

i Montolio, A. G. (2023). A gentle introduction to SMT-based program analysis. Accessed: 2024-07-25.

Ivanov., V., V. Romanov., and G. Succi. (2021). Predicting type annotations for python using embeddings from graph neural networks. In *Proceedings of the 23rd International Conference on Enterprise Information Systems - Volume 1: ICEIS,*, pp. 548–556. INSTICC: SciTePress.

JetBrains (2024). Intellij idea. Accessed: March 26, 2024.

Jhala, R. and R. Majumdar (2005). Path slicing. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 38–47. ACM.

Joern (2023). Open-source code analysis platform for C/C++ based on code property graphs.

Karaivanov, S., V. Raychev, and M. T. Vechev (2014). Phrase-based statistical translation of programming languages. In A. P. Black, S. Krishnamurthi, B. Bruegge, and J. N. Ruskiewicz (Eds.), *Onward! 2014, Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software, part of SPLASH '14, Portland, OR, USA, October 20-24, 2014*, pp. 173–184. ACM.

Korel, B. and J. Laski (1988, October). Dynamic program slicing. *Inf. Process. Lett. 29*(3), 155–163.

Kuck, D. J., R. H. Kuhn, D. A. Padua, B. Leasure, and M. Wolfe (1981). Dependence graphs and compiler optimizations. In *POPL*, pp. 207–218. ACM Press.

Lagniez, J. and A. Biere (2013). Factoring out assumptions to speed up MUS extraction. In M. Järvisalo and A. V. Gelder (Eds.), *Theory and Applications of Satisfiability Testing - SAT 2013 - 16th International Conference, Helsinki, Finland, July 8-12, 2013. Proceedings*, Volume 7962 of *Lecture Notes in Computer Science*, pp. 276–292. Springer.

Lee, S., D. Binkley, R. Feldt, N. Gold, and S. Yoo (2019). Moad: Modeling observation-based approximate dependency. In *2019 19th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pp. 12–22.

Lee, S., D. Binkley, N. Gold, S. Islam, J. Krinke, and S. Yoo (2020, feb). Evaluating lexical approximation of program dependence. *J. Syst. Softw. 160*(C).

Lewis, M., Y. Liu, N. Goyal, M. Ghazvininejad, A. Mohamed, O. Levy, V. Stoyanov, and L. Zettlemoyer (2020). BART: denoising sequence-to-sequence pre-training for natural language generation, translation, and comprehension. In *ACL*, pp. 7871–7880. Association for Computational Linguistics.

Lewkowycz, A., A. Andreassen, D. Dohan, E. Dyer, H. Michalewski, V. V. Ramasesh, A. Slone, C. Anil, I. Schlag, T. Gutman-Solo, Y. Wu, B. Neyshabur, G. Gur-Ari, and V. Misra (2022). Solving quantitative reasoning problems with language models. In S. Koyejo, S. Mohamed, A. Agarwal, D. Belgrave, K. Cho, and A. Oh (Eds.), *Advances in Neural Information Processing Systems 35: Annual Conference on Neural Information Processing Systems 2022, NeurIPS 2022, New Orleans, LA, USA, November 28 - December 9, 2022*.

Li, M., Z. Shi, Q. Lai, S. Khan, S. Cai, and Q. Xu (2023). DeepSAT: An EDA-Driven Learning Framework for SAT.

Li, R., L. B. Allal, Y. Zi, N. Muennighoff, D. Kocetkov, C. Mou, M. Marone, C. Akiki, J. Li, J. Chim, Q. Liu, E. Zheltonozhskii, T. Y. Zhuo, T. Wang, O. Dehaene, M. Davaadorj, J. Lamy-Poirier, J. Monteiro, O. Shliazhko, N. Gontier, N. Meade, A. Zebaze, M.-H. Yee, L. K. Umapathi, J. Zhu, B. Lipkin, M. Oblokulov, Z. Wang, R. Murthy, J. Stillerman, S. S. Patel, D. Abulkhanov, M. Zocca, M. Dey, Z. Zhang, N. Fahmy, U. Bhattacharyya, W. Yu, S. Singh, S. Luccioni, P. Villegas, M. Kunakov, F. Zhdanov, M. Romero, T. Lee, N. Timor, J. Ding, C. Schlesinger, H. Schoelkopf, J. Ebert, T. Dao, M. Mishra, A. Gu, J. Robinson, C. J. Anderson, B. Dolan-Gavitt, D. Contractor, S. Reddy, D. Fried, D. Bahdanau, Y. Jernite, C. M. Ferrandis, S. Hughes, T. Wolf, A. Guha, L. von Werra, and H. de Vries (2023). Starcoder: may the source be with you! *CoRR abs/2305.06161*.

Li, Y., S. Wang, T. N. Nguyen, and S. Van Nguyen (2019, October). Improving bug detection via context-based code representation learning and attention-based neural networks. *Proc. ACM Program. Lang. 3*(OOPSLA).

Li, Y., A. Yadavally, J. Zhang, S. Wang, and T. N. Nguyen (2023). Deminify: Neural variable name recovery and type inference. In *ESEC/SIGSOFT FSE*, pp. 758–770. ACM.

Li, Z., D. Zou, S. Xu, X. Ou, H. Jin, S. Wang, Z. Deng, and Y. Zhong (2018). Vuldeepecker: A deep learning-based system for vulnerability detection. In *25th Annual Network and Distributed System Security Symposium, NDSS 2018, San Diego, California, USA, February 18-21, 2018*. The Internet Society.

Liang, T., A. Reynolds, N. Tsiskaridze, C. Tinelli, C. W. Barrett, and M. Deters (2016). An efficient SMT solver for string constraints. *Formal Methods Syst. Des. 48*(3), 206–234.

Liffiton, M. H. and A. Malik (2013). Enumerating Infeasibility: Finding Multiple MUSes Quickly. In C. P. Gomes and M. Sellmann (Eds.), *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems, 10th International Conference, CPAIOR 2013, Yorktown Heights, NY, USA, May 18-22, 2013.*, Volume 7874 of *Lecture Notes in Computer Science*, pp. 160–175. Springer.

Liffiton, M. H., A. Previti, A. Malik, and J. Marques-Silva (2016). Fast, flexible MUS enumeration. *Constraints An Int. J. 21*(2), 223–250.

Liu, C., S. Lu, W. Chen, D. Jiang, A. Svyatkovskiy, S. Fu, N. Sundaresan, and N. Duan (2023). Code execution with pre-trained language models. In *ACL (Findings)*, pp. 4984–4999. Association for Computational Linguistics.

Liu, Y., M. Ott, N. Goyal, J. Du, M. Joshi, D. Chen, O. Levy, M. Lewis, L. Zettlemoyer, and V. Stoyanov (2019). Roberta: A robustly optimized bert pretraining approach. *arXiv preprint arXiv:1907.11692*.

Liu, Z., Z. Tang, J. Zhang, X. Xia, and X. Yang (2024). Pre-training by predicting program dependencies for vulnerability analysis tasks. In *ICSE*, pp. 151:1–151:13. ACM.

Lu, S., N. Duan, H. Han, D. Guo, S. won Hwang, and A. Svyatkovskiy (2022). Reacc: A retrieval-augmented code completion framework.

Mai, Y., Z. Gao, X. Hu, L. Bao, Y. Liu, and J. Sun (2024, July). Are human rules necessary? generating reusable apis with cot reasoning and in-context learning. *Proc. ACM Softw. Eng. 1*(FSE).

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

Marques-Silva, J., I. Lynce, and S. Malik (2021). Conflict-driven clause learning SAT solvers. In A. Biere, M. Heule, H. van Maaren, and T. Walsh (Eds.), *Handbook of Satisfiability - Second Edition*, Volume 336 of *Frontiers in Artificial Intelligence and Applications*, pp. 133–182. IOS Press.

Mikolov, T., K. Chen, G. S. Corrado, and J. Dean (2013). Efficient estimation of word representations in vector space.

Mir, A. M., E. Latoškinas, S. Proksch, and G. Gousios (2022). Type4py: Practical deep similarity learning-based type inference for python. In *Proceedings of the 44th International Conference on Software Engineering*, ICSE '22, New York, NY, USA, pp. 2241–2252. Association for Computing Machinery.

Morishita, T., G. Morio, A. Yamaguchi, and Y. Sogawa (2023). Learning deductive reasoning from synthetic corpus based on formal logic. In A. Krause, E. Brunskill, K. Cho, B. Engelhardt, S. Sabato, and J. Scarlett (Eds.), *International Conference on Machine Learning, ICML 2023, 23-29 July 2023, Honolulu, Hawaii, USA*, Volume 202 of *Proceedings of Machine Learning Research*, pp. 25254–25274. PMLR.

Natarajan, S., S. Mathur, S. Sidheekh, W. Stammer, and K. Kersting (2025). Human-in-the-loop or ai-in-the-loop? automate or collaborate? In *AAAI*, pp. 28594–28600. AAAI Press.

Nguyen, A. T., H. A. Nguyen, and T. N. Nguyen (2016). A large-scale study on repetitiveness, containment, and composability of routines in open-source projects. In *MSR*, pp. 362–373. ACM.

Nguyen, A. T., T. T. Nguyen, and T. N. Nguyen (2013). Lexical statistical machine translation for language migration. In B. Meyer, L. Baresi, and M. Mezini (Eds.), *Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, ESEC/FSE'13, Saint Petersburg, Russian Federation, August 18-26, 2013*, pp. 651–654. ACM.

Nguyen, T. T., A. T. Nguyen, H. A. Nguyen, and T. N. Nguyen (2013). A statistical semantic language model for source code. In B. Meyer, L. Baresi, and M. Mezini (Eds.), *Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, ESEC/FSE'13, Saint Petersburg, Russian Federation, August 18-26, 2013*, pp. 532–542. ACM.

Nguyen, T. V., A. Yadavally, and T. N. Nguyen (2022). Phrase2set: Phrase-to-set machine translation and its software engineering applications. In *SANER*, pp. 502–513. IEEE.

Nikitopoulos, G., K. Dritsa, P. Louridas, and D. Mitropoulos (2021). Crossvul: a cross-language vulnerability dataset with commit data. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2021, New York, NY, USA, pp. 1565–1569. Association for Computing Machinery.

Nishimatsu, A., M. Jihira, S. Kusumoto, and K. Inoue (1999). Call-mark slicing: An efficient and economical way of reducing slice. In *Proceedings of the 21st International Conference on Software Engineering*, pp. 422–431. ACM.

OpenAI (2023a). GPT-4 technical report. *CoRR abs/2303.08774*.

OpenAI (2023b). Gpt-4 technical report. Accessed: 2025-06-10.

Orso, A., S. Sinha, and M. J. Harrold (2001). Incremental slicing based on data-dependence types. In *Proceedings of the IEEE International Conference on Software Maintenance*, pp. 158–167. IEEE Computer Society.

Ouyang, L., J. Wu, X. Jiang, et al. (2022). Training language models to follow instructions with human feedback. *NeurIPS*.

Peng, Y., C. Gao, Z. Li, B. Gao, D. Lo, Q. Zhang, and M. Lyu (2022). Static inference meets deep learning: A hybrid type inference approach for python. In *Proceedings of the 44th International Conference on Software Engineering*, ICSE '22, New York, NY, USA, pp. 2019–2030. Association for Computing Machinery.

Phan, H., H. A. Nguyen, N. M. Tran, L. H. Truong, A. T. Nguyen, and T. N. Nguyen (2018). Statistical Learning of API Fully Qualified Names in Code Snippets of Online Forums. In *Proceedings of the 40th International Conference on Software Engineering*, ICSE '18, New York, NY, USA, pp. 632–642. Association for Computing Machinery.

Podgurski, A. and L. A. Clarke (1990). A formal model of program dependences and its implications for software testing, debugging, and maintenance. *IEEE Trans. Software Eng. 16*(9), 965–979.

Prabhu, N., S. Min, H. Nam, G. Tewolde, and J. Kwon (2020). Integrated framework of autonomous vehicle with traffic sign recognition in simulation environment. In *2020 IEEE International Conference on Electro Information Technology (EIT)*, pp. 514–521.

Pradel, M., G. Gousios, J. Liu, and S. Chandra (2020). Typewriter: Neural type prediction with search-based validation. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2020, New York, NY, USA, pp. 209–220. Association for Computing Machinery.

Puri, R., D. S. Kung, G. Janssen, W. Zhang, G. Domeniconi, V. Zolotov, J. Dolby, J. Chen, M. R. Choudhury, L. Decker, V. Thost, L. Buratti, S. Pujar, and U. Finkler (2021). Project codenet: A large-scale AI for code dataset for learning a diversity of coding tasks. *CoRR abs/2105.12655*.

Radford, A., J. Wu, R. Child, D. Luan, D. Amodei, I. Sutskever, et al. (2019). Language models are unsupervised multitask learners. *OpenAI blog 1*(8), 9.

Raffel, C., N. Shazeer, A. Roberts, K. Lee, S. Narang, M. Matena, Y. Zhou, W. Li, and P. J. Liu (2020). Exploring the limits of transfer learning with a unified text-to-text transformer. *J. Mach. Learn. Res. 21*, 140:1–140:67.

Ragkhitwetsagul, C., J. Krinke, M. Paixao, G. Bianco, and R. Oliveto (2021). Toxic code snippets on stack overflow. *IEEE Transactions on Software Engineering 47*(3), 560–581.

Ricca, F. and P. Tonella (2001). Web application slicing. In *Proceedings of the IEEE International Conference on Software Maintenance*, pp. 148–157. IEEE Computer Society.

Ricca, F. and P. Tonella (2002). Construction of the system dependence graph for web application slicing. In *Proceedings of the Second IEEE International Workshop on Source Code Analysis and Manipulation*, pp. 123–132. IEEE Press.

Saifullah, C. M. K., M. Asaduzzaman, and C. K. Roy (2019). Learning from examples to find fully qualified names of API elements in code snippets. In *ASE*, pp. 243–254. IEEE.

Schkufza, E., R. Sharma, and A. Aiken (2016). Stochastic program optimization. *Commun. ACM 59*(2), 114–122.

Schmidhuber, J. (2004). Optimal ordered problem solver. *Mach. Learn. 54*(3), 211–254.

See, A., P. J. Liu, and C. D. Manning (2017). Get to the point: Summarization with pointer-generator networks. In *ACL (1)*, pp. 1073–1083. Association for Computational Linguistics.

Selsam, D. and N. S. Bjørner (2019). Guiding high-performance SAT solvers with UNSAT-Core predictions. In M. Janota and I. Lynce (Eds.), *Theory and Applications of Satisfiability Testing - SAT 2019 - 22nd International Conference, SAT 2019, Lisbon, Portugal, July 9-12, 2019, Proceedings*, Volume 11628 of *Lecture Notes in Computer Science*, pp. 336–353. Springer.

Selsam, D., M. Lamm, B. Bünz, P. Liang, L. de Moura, and D. L. Dill (2019). Learning a SAT solver from single-bit supervision. In *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*. OpenReview.net.

Shi, Z., M. Li, Y. Liu, S. Khan, J. Huang, H. Zhen, M. Yuan, and Q. Xu (2023). SATformer: Transformer-Based UNSAT Core Learning. In *IEEE/ACM International Conference on Computer Aided Design, ICCAD 2023, San Francisco, CA, USA, October 28 - Nov. 2, 2023*, pp. 1–4. IEEE.

Silva, J. (2012, June). A vocabulary of program slicing-based techniques. *ACM Comput. Surv. 44*(3), 12:1–12:41.

Souza, B. and M. 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*, ESEC/FSE 2023, New York, NY, USA, pp. 1522–1534. Association for Computing Machinery.

Svajlenko, J., J. F. Islam, I. Keivanloo, C. K. Roy, and M. M. Mia (2014). Towards a big data curated benchmark of inter-project code clones. In *ICSME*, pp. 476–480. IEEE Computer Society.

Tonella, P. and F. Ricca (2005). Web application slicing in presence of dynamic code generation. *Journal of Automated Software Engineering 12*(2), 259–288.

Touvron, H., L. Martin, K. Stone, P. Albert, A. Almahairi, Y. Babaei, N. Bashlykov, S. Batra, P. Bhargava, S. Bhosale, D. Bikel, L. Blecher, C. Canton-Ferrer, M. Chen, G. Cucurull, D. Esiobu, J. Fernandes, J. Fu, W. Fu, B. Fuller, C. Gao, V. Goswami, N. Goyal, A. Hartshorn, S. Hosseini, R. Hou, H. Inan, M. Kardas, V. Kerkez, M. Khabsa, I. Kloumann, A. Korenev, P. S. Koura, M. Lachaux, T. Lavril, J. Lee, D. Liskovich,

Y. Lu, Y. Mao, X. Martinet, T. Mihaylov, P. Mishra, I. Molybog, Y. Nie, A. Poulton, J. Reizenstein, R. Rungta, K. Saladi, A. Schelten, R. Silva, E. M. Smith, R. Subramanian, X. E. Tan, B. Tang, R. Taylor, A. Williams, J. X. Kuan, P. Xu, Z. Yan, I. Zarov, Y. Zhang, A. Fan, M. Kambadur, S. Narang, A. Rodriguez, R. Stojnic, S. Edunov, and T. Scialom (2023). Llama 2: Open foundation and fine-tuned chat models. *CoRR abs/2307.09288*.

Vaswani, A., N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin (2017). Attention is all you need. In *NIPS*, pp. 5998–6008.

Verdi, M., A. Sami, J. Akhondali, F. Khomh, G. Uddin, and A. K. Motlagh (2022). An empirical study of C++ vulnerabilities in crowd-sourced code examples. *IEEE Trans. Software Eng. 48*(5), 1497–1514.

Visser, W., K. Havelund, G. P. Brat, and S. Park (2000). Model checking programs. In *ASE*, pp. 3–12. IEEE Computer Society.

Wang, C., J. Zhang, Y. Lou, M. Liu, W. Sun, Y. Liu, and X. Peng (2025, May). TIGER: A Generating-Then-Ranking Framework for Practical Python Type Inference . In *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*, Los Alamitos, CA, USA, pp. 321–333. IEEE Computer Society.

Wang, S., D. Chollak, D. Movshovitz-Attias, and L. Tan (2016). Bugram: bug detection with n-gram language models. In D. Lo, S. Apel, and S. Khurshid (Eds.), *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, ASE 2016, Singapore, September 3-7, 2016*, pp. 708–719. ACM.

Wang, W., Y. Hu, M. Tiwari, S. Khurshid, K. L. McMillan, and R. Miikkulainen (2021). Neurocomb: Improving SAT solving with graph neural networks. *CoRR abs/2110.14053*.

Wang, W., T. N. Nguyen, S. Wang, Y. Li, J. Zhang, and A. Yadavally (2023). Deepvd: Toward class-separation features for neural network vulnerability detection. In *ICSE*, pp. 2249–2261. IEEE.

Wang, X., J. Wei, D. Schuurmans, Q. Le, E. Chi, S. Narang, A. Chowdhery, and D. Zhou (2022). Self-consistency improves chain of thought reasoning in language models. *arXiv preprint arXiv:2203.11171*.

Wang, Y., W. Wang, S. R. Joty, and S. C. H. Hoi (2021). Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. In *EMNLP (1)*, pp. 8696–8708. Association for Computational Linguistics.

Wei, J., X. Wang, D. Schuurmans, M. Bosma, B. Ichter, F. Xia, E. H. Chi, Q. V. Le, and D. Zhou (2022). Chain-of-thought prompting elicits reasoning in large language models. In S. Koyejo, S. Mohamed, A. Agarwal, D. Belgrave, K. Cho, and A. Oh (Eds.), *Advances*

in Neural Information Processing Systems 35: Annual Conference on Neural Information Processing Systems 2022, NeurIPS 2022, New Orleans, LA, USA, November 28 - December 9, 2022.

Weiser, M. D. (1984). Program slicing. *IEEE Trans. Software Eng. 10*(4), 352–357.

White, C. (2013, April). Hadoop filesplit reading. Stack Overflow answer. Online; accessed June 19, 2025.

Wu, Y., D. Zou, S. Dou, W. Yang, D. Xu, and H. Jin (2022). Vulcnn: An image-inspired scalable vulnerability detection system.

Xu, B., Z. Chen, and H. Yang (2002). Dynamic slicing object-oriented programs for debugging. In *Proceedings. Second IEEE International Workshop on Source Code Analysis and Manipulation*, pp. 115–122.

Xu, B., J. Qian, X. Zhang, Z. Wu, and L. Chen (2005, mar). A brief survey of program slicing. *SIGSOFT Softw. Eng. Notes 30*(2), 1–36.

Yadavally, A. (2024). Replication package for safemin.

Yadavally, A., Y. Li, and T. N. Nguyen (2024b). Predictive program slicing via execution knowledge-guided dynamic dependence learning. *Proc. ACM Softw. Eng. 1*(FSE), 271–292.

Yadavally, A., Y. Li, S. Wang, and T. N. Nguyen (2024a). A learning-based approach to static program slicing. *Proc. ACM Program. Lang. 8*(OOPSLA1), 83–109.

Yadavally, A., T. N. Nguyen, W. Wang, and S. Wang (2023). (partial) program dependence learning. In *Proceedings of the 45th International Conference on Software Engineering*, ICSE '23, pp. 2501–2513. IEEE Press.

Yadavally, A., X. Rong, P. Nguyen, and T. N. Nguyen (2025). Large language models for safe minimization. In *ICSE*, pp. 1114–1126. IEEE.

Yasunaga, M. and P. Liang (2020). Graph-based, self-supervised program repair from diagnostic feedback. In *ICML*, Volume 119 of *Proceedings of Machine Learning Research*, pp. 10799–10808. PMLR.

Yu, X., J. Liu, Z. Yang, and X. Liu (2017, dec). The bayesian network based program dependence graph and its application to fault localization. *J. Syst. Softw. 134*(C), 44–53.

Zhang, J., X. Wang, H. Zhang, H. Sun, K. Wang, and X. Liu (2019). A novel neural source code representation based on abstract syntax tree. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pp. 783–794.

Zhao, W. and M. H. Liffiton (2016). Parallelizing partial MUS enumeration. In *ICTAI*, pp. 464–471. IEEE Computer Society.

Zhong, H. and X. Wang (2017). Boosting complete-code tool for partial program. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 671–681.

Zhou, Y. and A. Sharma (2017). Automated identification of security issues from commit messages and bug reports. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2017, New York, NY, USA, pp. 914–919. Association for Computing Machinery.

Zhu, H., P. A. V. Hall, and J. H. R. May (1997). Software unit test coverage and adequacy. *ACM Comput. Surv. 29*(4), 366–427.

BIOGRAPHICAL SKETCH

Aashish Yadavally is a PhD candidate in the Erik Jonsson School of Engineering and Computer Science at The University of Texas at Dallas. He holds a master's degree in Artificial Intelligence from the University of Georgia. His research lies at the intersection of Artificial Intelligence and Software Engineering, with a focus on optimizing software development processes and enhancing software security. He has published in top-tier venues such as ICSE, FSE, ASE, and OOPSLA. His work has received multiple accolades, including the ACM SIGSOFT Distinguished Paper Award at FSE 2024 and the IEEE TCSE Distinguished Paper Award at SANER 2022. His long-term vision is to establish and advance scalable, AI-assisted development workflows for secure and reliable software engineering.

CURRICULUM VITAE

Aashish Yadavally

July 7, 2025

Contact Information:

Department of Computer Science
The University of Texas at Dallas
800 W. Campbell Rd.
Richardson, TX 75080-3021, U.S.A.

Email: aashish.yadavally@utdallas.edu

Educational History:

BTech, Computer Science, Indian Institute of Information Technology, Vadodara, 2018
MS, Artificial Intelligence, University of Georgia, 2020
PhD, Computer Science, University of Texas at Dallas, 2025

Neural Modeling of Reasoning about Program Behaviors
PhD Dissertation
Computer Science Department, University of Texas at Dallas
Advisor: Dr. Tien N. Nguyen

An Exploration of Machine Learning-Based Solar Irradiance Forecasting Methodologies
MS Thesis
Institute for Artificial Intelligence, University of Georgia
Advisor: Dr. Frederick Maier

Professional Recognitions and Honors:

ACM SIGSOFT Distinguished Paper Award, FSE 2024
Distinguished Junior PC Reviewer Award, MSR 2024
Nomination for ACM SIGSOFT Distinguished Paper Award, ICSE 2023
IEEE TCSE Distinguished Paper Award, SANER 2022
Research Scholarship w/ Full Tuition Remission, University of Georgia, 2018 – 2020

Publications:

[17] Aashish Yadavally, Xiaokai Rong, Phat Nguyen, and Tien N. Nguyen. 2025. "Large Language Models for Safe Minimization". In *Proceedings of the 47th IEEE/ACM International Conference on Software Engineering*, ICSE 2025, Ontario, Canada, April 27-May 3, 2025. IEEE, 1114-1126.

[16] Smit Soneshbhai Patel, Aashish Yadavally, Hridya Dhulipala, and Tien N. Nguyen. 2025. "Planning a Large Language Model for Static Detection of Runtime Errors in Code Snippets". In *Proceedings of the 47th IEEE/ACM International Conference on Software Engineering*, ICSE 2025, Ontario, Canada, April 27-May 3, 2025. IEEE, 872-884.

[15] Yi Li, Hridya Dhulipala, Aashish Yadavally, Xiaokai Rong, Shaohua Wang, and Tien N. Nguyen. 2025. "Blended Analysis for Predictive Execution". In *Proceedings of the ACM Software Engineering. Volume 2, FSE (2025)*, 2987-3008.

[14] Hridya Dhulipala, Aashish Yadavally, Smit Soneshbai Patel, and Tien N. Nguyen. 2025. "CRISPE: Semantic-Guided Execution Planning and Dynamic Reasoning for Enhancing Code Coverage Prediction". In *Proceedings of the ACM Software Engineering. Volume 2, FSE (2025)*, 2965-2986.

[13] Aashish Yadavally, Yi Li, and Tien N. Nguyen. 2024. "Predictive Program Slicing via Execution Knowledge-Guided Dynamic Dependence Learning". In *Proceedings of the ACM Software Engineering. Volume 1, FSE (2024)*, 271–292.

★ **ACM SIGSOFT Distinguished Paper Award**

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

[11] Hridya Dhulipala, Aashish Yadavally, and Tien N. Nguyen. 2024. "Planning to Guide LLM for Code Coverage Prediction". In *Proceedings of the 2024 IEEE/ACM First International Conference on AI Foundation Models and Software Engineering*, FORGE 2024, Lisbon, Portugal, 14 April 2024. ACM, 24–34.

[10] Yuchen Cai, Aashish Yadavally, Abhishek Mishra, Genesis Montejo, and Tien N. Nguyen. 2024. "Programming Assistant for Exception Handling with CodeBERT". In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*, ICSE 2024, Lisbon, Portugal, April 14-20, 2024. ACM, 94:1–94:13.

[9] Yi Li, Tien N. Nguyen, Yuchen Cai, Aashish Yadavally, Abhishek Mishra, and Genesis Montejo. 2024. "Neural Exception Handling Recommender". In *Proceedings of the 2024 IEEE/ACM 46th International Conference on Software Engineering: Companion Proceedings*, ICSE Companion 2024, Lisbon, Portugal, April 14-20, 2024. ACM, 316–317.

[8] Yi Li, Tien N. Nguyen, Shaohua Wang, and Aashish Yadavally. 2024. "Poirot: Deep Learning for API Misuse Detection". In *Proceedings of the 2024 IEEE/ACM 46th International Conference on Software Engineering: Companion Proceedings*, ICSE Companion 2024, Lisbon, Portugal, April 14-20, 2024. ACM, 302–303.

[7] Yi Li, Aashish Yadavally, Jiaxing Zhang, Shaohua Wang, and Tien N. Nguyen. 2023. "Commit-Level, Neural Vulnerability Detection and Assessment". In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2023, San Francisco, CA, USA, December 3-9, 2023. ACM, 1024–1036.

[6] Yi Li, Aashish Yadavally, Jiaxing Zhang, Shaohua Wang, and Tien N. Nguyen. 2023. "DeMinify: Neural Variable Name Recovery and Type Inference". In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2023, San Francisco, CA, USA, December 3-9, 2023. ACM, 758–770.

[5] Wenbo Wang, Tien N. Nguyen, Shaohua Wang, Yi Li, Jiyuan Zhang, and Aashish Yadavally. 2023. "DeepVD: Toward Class-Separation Features for Neural Network Vulnerability Detection". In *45th IEEE/ACM International Conference on Software Engineering*, ICSE 2023, Melbourne, Australia, May 14-20, 2023. IEEE, 2249–2261.

[4] Aashish Yadavally, Tien N. Nguyen, Wenbo Wang, and Shaohua Wang. 2023. "(Partial) Program Dependence Learning". In *45th IEEE/ACM International Conference on Software Engineering*, ICSE 2023, Melbourne, Australia, May 14-20, 2023. IEEE,2501–2513.

★ **Nomination for the ACM SIGSOFT Distinguished Paper Award**

[3] Hoan Anh Nguyen, Hung Dang Phan, Syeda Khairunnesa Samantha, Son Nguyen, Aashish Yadavally, Shaohua Wang, Hridesh Rajan, and Tien N. Nguyen. 2022. "A Hybrid Approach for Inference between Behavioral Exception API Documentation and Implementations, and Its Applications". In *37th IEEE/ACM International Conference on Automated Software Engineering*, ASE 2022, Rochester, MI, USA, October 10-14, 2022. ACM, 2:1–2:13.

[2] Anh Tuan Nguyen, Aashish Yadavally, and Tien N. Nguyen. 2022. "Next Syntactic-Unit Code Completion and Applications". In *37th IEEE/ACM International Conference on Automated Software Engineering*, ASE 2022, Rochester, MI, USA, October 10-14, 2022. ACM, 180:1–180:5.

[1] Thanh V. Nguyen, Aashish Yadavally, and Tien N. Nguyen. 2022. "Phrase2Set: Phrase-to-Set Machine Translation and Its Software Engineering Applications". In *IEEE International Conference on Software Analysis, Evolution and Reengineering*, SANER 2022, Honolulu, HI, USA, March 15-18, 2022. IEEE, 502–513.

★ **IEEE TCSE Distinguished Paper Award**

Professional Service:

Peer-Reviewed Conferences:

[6] Program Committee Member, International Conference on Evaluation and Assessment in Software Engineering, AI Models/Data Track (EASE'25).

[5] Program Committee Member, International Conference on Learning Representations, Research Track (ICLR'25).

[4] Shadow Program Committee Member, International Conference on Software Engineering, Research Track (ICSE'25).

[3] Program Committee Member, International Conference on Software Engineering, Artifact Evaluation (ICSE'24).

[2] Junior Program Committee Member, International Conference on Mining Software Repositories, Research Track (MSR'24).

★ **Distinguished Junior PC Reviewer Award**

[1] Junior Program Committee Member, International Conference on Mining Software Repositories, Research Track (MSR'23).

Peer-Reviewed Journals:

[2] Reviewer, IEEE Transactions on Software Engineering (TSE).

[1] Reviewer, Empirical Software Engineering (EMSE).

Professional Memberships:

Institute of Electrical and Electronics Engineers (IEEE), 2021–present
Association of Computing Machinery (ACM), 2021–present