Learning to Analyze Program Behaviors

Aashish Yadavally
University of Texas at Dallas
Texas, USA
aashish.yadavally@utdallas.edu

ABSTRACT

Analyzing programs typically entails: (1) examining all program behaviors that might arise at run time on the basis of program semantics (*i.e.*, *static*), (2) establishing precise run-time behaviors on the basis of the actual execution of test suite (*i.e.*, *dynamic*). By design, static analyses overestimate and are imprecise. On the other hand, dynamic analyses are more precise, but can not generalize. In practice, developers often have to choose between one or the other.

To bridge this gap, we propose a new paradigm of *predictive program analysis*, which aims to learn to analyze program behaviors from the semantic and execution-level analyses of programs in open-source software repositories. This helps widen the scope of traditional program analyses with missing dependency information and prior execution knowledge. As a result, it can extend program analysis to incomplete code, circumvent the need to setup run-time environments, reduce search space, and enhance generalizability. [1]

CCS CONCEPTS

• **Computing methodologies** → **Neural networks**; • **Theory of computation** → **Program analysis**.

KEYWORDS

AI4SE, Neural Networks, Program Analysis, LLM

1 INTRODUCTION

Program analysis is the process of examining a program to reason about its properties, structures, and behaviors. There are two main themes of program analysis: *static* and *dynamic*. The former uses program semantics to examine all possible behaviors that might arise at run time, while the latter uses actual execution for specific inputs to establish precise run-time behaviors.

Static analyses, while comprehensive, scale super-linearly and introduce complex interactions. Moreover, since they approximate certain aspects of program behavior (for *e.g.*, user input or network state), they also need to deal with the resulting unknowns. Thus, they are less precise and tend to *overestimate*, thereby limiting their overall effectiveness. On the other hand, dynamic analyses provide precise insights over executions of the test suite. However, it is an *under-approximation* and can not generalize, as the test suite may not cover all possible execution paths the program can take.

As Ernst [3] notes, it is desirable to bridge the gap between static and dynamic analyses, compromising a bit on the soundness of the former and the preciseness of the latter. To this end, my research proposes an alternative **predictive program analysis**. Following from the hypothesis of the *naturalness of software* [6], *i.e.*, the program properties are predictable, predictive analysis:

> *aims to learn to analyze program behaviors from similar structural, semantic and execution-level analyses of entire programs obtained from ultra-large-scale, open-source software repositories.*

We envision multiple benefits to such a predictive paradigm. *First*, static analysis tools falter when missing information, as in the case of dependency injection, or multilingual analysis. However, learning-based approaches can help acquire the knowledge of such fine-grained dependencies, while also being programming language-agnostic. With the generalization capabilities of advanced machine learning models, such *intrinsic* knowledge can help reduce the false positives and over-approximation in static analysis.

Second, dynamic analysis typically requires access to the entire source code for a system, as well as the collection and storage of runtime data. These challenges are exacerbated in security-sensitive and resource-constrained environments. In contrast, a predictive paradigm *side-steps the need for actual program execution* by utilizing complete code corpora and associated prior execution knowledge to enable approximate run-time analyses at a large-scale.

Third, predictive analyses can utilize pre-trained code language models. These models typically input sequences of code tokens, thereby freeing them from the dependence on the completeness of programs. As a result, we can extend such analyses to incomplete code fragments from developer forums such as StackOverflow, which further helps assess and enhance their usability.

Overall, predictive analysis is useful to widen the scope of static analysis by helping fill in the necessary missing information either implicitly or explicitly; and for dynamic analysis by helping it generalize to inputs beyond the test suite. Thus, it can be more precise than static analysis, and more complete than dynamic analysis.

2 CONTRIBUTIONS SO FAR

As a proof-of-concept for such predictive analysis, we studied a fundamental class of program analysis, namely, *dependence analysis*, which focuses on reasoning about the dependencies of one component of a program on the other components. This choice is rooted in exploiting the operation of program dependence analysis in both static and dynamic settings. In this section, we present our works on exploring the effectiveness of learning-based methodologies in capturing program dependencies at different levels of granularity.

[1] Research Supervisor: Dr. Tien N. Nguyen, University of Texas at Dallas, Texas, USA.

FSE'24, July 2024, Porto de Galinhas, Brazil

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM
https://doi.org/10.1145/nnnnnnn.nnnnnnn

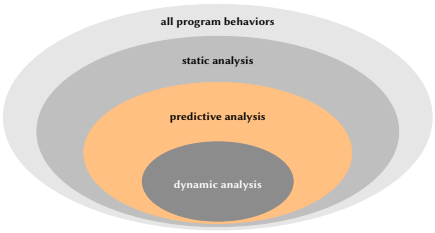

Figure 1: Hierarchy of program analysis techniques. Note that the effectiveness of predictive analysis can typically scale with an increase in training data and model size.

2.1 Learning-Guided Static Analysis

2.1.1 Inter-Statement Dependencies. ***Status:*** *Completed.*

In an empirical study on the repetitiveness, containment, and composability of program dependence graphs (PDGs) in open-source projects, Nguyen *et al.* [8] reported that among 17.5M PDGs with 1.6B PDG subgraphs, 14.3% of the PDGs have all of their subgraphs repeating 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. Based on this observation, we hypothesized that it is possible to *learn to analyze the inter-statement dependencies* captured in a PDG from complete code in existing code corpora.

Accordingly, we introduced NeuralPDA [11], a neural network-based PDG-building tool, which *implicitly* learns the semantic relations between program statements by learning the dependencies between their dense, contextualized latent representations obtained from mapping them into an embedding space. In other words, we reformulate PDG construction in NeuralPDA as structured link prediction, learning to predict whether a program dependence relation exists between pairs of statements in a program.

While trained on PDGs extracted from complete code corpora, NeuralPDA can generate highly accurate PDGs for (in)complete programs as well – all without an explicit dependence analysis. For complete Java and C/C++ programs, the generated CFG+PDGs are **94.29%** and **92.46%** accurate, respectively, and for incomplete programs, they are 94.29–97.17% and 92.46–96.01% accurate. Moreover, we observed that it is **380**× faster than a state-of-the-art, traditional PDG-building tool. We also demonstrated the usefulness of such predicted PDGs in *detecting vulnerabilities in partial code snippets* from StackOverflow, which is particularly useful for preventing the propagation of defects while reusing the code. Overall, we note that the PDGs predicted by NeuralPDA are useful for other software engineering applications as well, those of which can tolerate a low level of errors and imprecision in inter-statement dependencies.

2.1.2 Variable-Statement Dependencies. ***Status:*** *Completed.*

Static program slicing is useful in manual/automated debugging, allowing developers to focus on the minimal subprogram containing faults. This improves code understanding, and facilitates effective bug fixing. Traditional slicing strategies typically rely on the construction of a system dependence graph (SDG), which is computationally intensive and is only feasible for complete code. In the case of incomplete code, missing declared data types, variable declarations, and the absence of external libraries limits their applicability. Here, NeuralPDA fails as well, as it operates solely at the statement-level and does not provide the fine-grained dependencies among the variables required to extract precise static slices.

To address these limitations, we introduced NS-Slicer [10], a neural network-based approach to static program slicing, which works for both complete and incomplete code. At its core, it utilizes code language models [4, 5] that benefit from pre-training knowledge encompassing code structure [7] and data-flow information [5]. This enables NS-Slicer to capture intricate *variable-statement dependencies* effectively, thus helping predict all statements that can influence, or be influenced by the slicing criterion. For a dataset containing Java programs, NS-Slicer predicts the backward and forward slices with an F1-score of **97.41%** and **95.82%**, respectively, while achieving an overall F1-score of **96.77%**. Notably, in **85.20%** of the cases, the static program slices predicted by NS-Slicer*exactly match entire slices from the oracle*. For partial programs, it achieved an F1-score of 96.77%–97.49% for backward slicing, 92.14%–95.40% for forward slicing, and an overall F1-score of 94.66%–96.62%.

By design, static program slicing serves as a means for probing the semantic knowledge in the code language models in NS-Slicer. Our experiments demonstrated a high performance in predicting such slices, *showcasing the effectiveness of learning-based approaches in analyzing and comprehending variable-statement dependencies*. Thus, NS-Slicer empowers program analysis approaches, which are traditionally reliant on static program slicing and require access to complete code, to also operate effectively on partial code.

2.2 Learning-Guided Dynamic Analysis

Our success in effectively capturing dependencies across different granularity levels using learning-based approaches prompted us to ask the question: *will this scale to dependencies that are tied to the execution of a program for specific inputs*?

In essence, learning-guided dynamic analysis would be akin to learning to select the specific execution path corresponding to the inputs from among all possible program paths. Subsequently, dynamic dependence analysis would entail pinpointing the program elements along the predicted execution path that are dependent on a particular point of interest. This is far more challenging. For instance, consider a scenario in which the point of interest lies within the scope of a `while`-loop. In this case, the loop could be executed multiple times, causing the variables within the loop to be affected by different sets of program elements in different executing iterations. Therefore, learning to discern between the dependencies across executing iterations would become critical.

2.2.1 Dynamic Dependencies. ***Status:*** *Completed.*

With a focus on enhancing the run-time analysis of programs, we looked at the notion of dynamic dependence learning in the context of *predictive slicing*, which is designed to *predict* the dynamic slices for a program. ND-Slicer [9] presents a new step forward in this direction, yielding approximate slices at scale. We tackled the problem of scalability by *side-stepping the need to execute the program altogether*, to compute the slice. To enable such a process, we leveraged execution-aware pre-training to learn the dynamic data and control dependencies between variables in the slicing criterion and the remaining program statements.

We observed that ND-Slicer predicted the dynamic backward slices with a high accuracy for both executable and non-executable

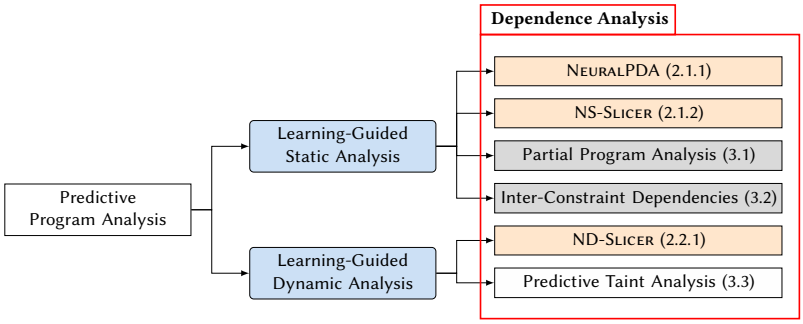

Figure 2: A taxonomy of different predictive program analysis tasks. Here ▭: "Completed", ▭: "In Progress", and ▭: "Planned".

(*i.e.*, those lacking referenced methods/classes) programs, determining the exact match slice in **81.3%** and and **57.5**% of the instances, respectively, with ROUGE-LCS F1-scores of **0.95** and **0.82**. Subsequently, it was effective in the task of crash detection as well, helping localize the crash faults. Finally, we exhibited that the learning objective in ND-Slicer helped it gain proficiency in comprehending complex program structures, including `if-else` blocks, loops, and control flow in inter-procedural calls – indicating the promise in the idea that dynamic dependence is "learnable" with neural networks.

3 GOING AHEAD

3.1 Comprehensive Partial Program Analysis

Status: *In Progress.*

Traditional program analysis methods typically necessitate access to entire source code to effectively analyze the program in question. However, in many real-world scenarios, obtaining the complete source code may not be feasible, or desirable even, due to several constraints. In this case, the ambiguity stemming from missing information leads static analysis tools to be even more conservative, and a lot of program properties could simply be skipped. Moreover, since the incomplete code would not be executable, dynamic analysis tools would be rendered ineffective altogether.

To address these limitations, we propose to use a learning-based approach in tandem with traditional program analysis approaches, wherein, the former could leverage its knowledge to fill in the missing information required to improve the effectiveness of the latter. In the context of building PDGs, this departs from NeuralPDA, where complete code was used as the training data to build a model that implicitly learns the dependence between program statements. Such an *interplay between learning-based and program analysis agents could help enhance both the precision and recall of program properties* – even more significantly for incomplete code snippets.

3.2 Reduction in Search Space

Status: *In Progress.*

In program analysis tasks like symbolic execution, each program path is represented as a logical constraint, which then leverage SMT solvers like Z3 [2], CVC5 [1], *etc.* as the back-end reasoning engine to aid a systematic exploration of all possible execution paths. However, an exponential search complexity is inherent in these solvers, thus limiting their real-world applicability.

In this research, we propose a direct minimization of the size of the constraint system while ensuring the satisfiability of the application-specific heuristics. We can enable this process by exploiting the *inter-constraint dependencies* and leverage a large language model (LLM) to macro-reason about the constraints and identify the non-contributing ones. We posit that the exposure of LLMs to massive amounts of source code and symbolic reasoning datasets during pre-training equips them to generalize across SMT theories (*e.g.*, integer and real arithmetic, strings, and bit vectors). Minimizing a constraint system with n constraints to m in this manner reduces the search space from $O(2^n)$ to $O(2^m)$.

In the context of predictive analysis, this research helps extend learning-guided static analysis to other analyses as well, including constraint-based program slicing, finding unreachable code, *etc.*

3.3 Predictive Taint Analysis

Status: *Planned.*

Motivated by the ability of code language models to "learn" dynamic dependencies, we aim to explore more dynamic analysis techniques that utilize such knowledge. For example, consider traditional dynamic taint analysis techniques. It runs a program and observes which computations are affected by predefined taint sources such as user input, external data sources, URLs, *etc.* In a learning-based paradigm, this would be similar in spirit to *predictive slicing*, except in the nature and scope of the point sources.

4 CONCLUSION

In this research, we advocate for a new paradigm for program analysis, in combination with learning-based methodologies. We evaluated the applicability of such a predictive analysis towards building program dependence graphs, as well as for enabling static and dynamic slicing of both complete and partial programs. As a future work, we motivate the use of predictive analysis in reducing the search space by analyzing the inter-constraint dependencies, improving the precision and recall of program properties while analyzing incomplete code snippets, and the exploration of more applications that leverage the learning of dynamic dependencies such as dynamic taint analysis. We expect that the proposed predictive program analysis will help bridge the gap between, and widen the scope of traditional static and dynamic analyses.

REFERENCES

[1] Haniel Barbosa, Clark W. Barrett, Martin Brain, Gereon Kremer, Hanna Lachnitt, Makai Mann, Abdalrhman Mohamed, Mudathir Mohamed, Aina Niemetz, Andres Nötzli, Alex Ozdemir, Mathias Preiner, Andrew Reynolds, Ying Sheng, Cesare Tinelli, and Yoni Zohar. 2022. cvc5: A Versatile and Industrial-Strength SMT Solver. In *TACAS (1) (Lecture Notes in Computer Science, Vol. 13243)*. Springer, 415–442.

[2] Leonardo Mendonça de Moura and Nikolaj S. Bjørner. 2008. Z3: An Efficient SMT Solver. In *TACAS (Lecture Notes in Computer Science, Vol. 4963)*. Springer, 337–340.

[3] Michael D. Ernst. 2004. Invited Talk Static and dynamic analysis: synergy and duality. In *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*, Cormac Flanagan and Andreas Zeller (Eds.). ACM, 35. https://doi.org/10.1145/996821.996823

[4] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. 2020. CodeBERT: A Pre-Trained Model for Programming and Natural Languages, Vol. EMNLP 2020. Association for Computational Linguistics, 1536–1547.

[5] Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, Michele Tufano, Shao Kun Deng, Colin B. Clement, Dawn Drain, Neel Sundaresan, Jian Yin, Daxin Jiang, and Ming Zhou. 2021. GraphCodeBERT: Pre-training Code Representations with Data Flow. In *ICLR*. OpenReview.net.

[6] Abram Hindle, Earl T. Barr, Mark Gabel, Zhendong Su, and Premkumar T. Devanbu. 2016. On the naturalness of software. *Commun. ACM* 59, 5 (2016), 122–131.

[7] José Antonio Hernández López, Martin Weyssow, Jesús Sánchez Cuadrado, and Houari A. Sahraoui. 2022. AST-Probe: Recovering abstract syntax trees from hidden representations of pre-trained language models. *CoRR* abs/2206.11719 (2022).

[8] Anh Tuan Nguyen, Hoan Anh Nguyen, and Tien N. Nguyen. 2016. A large-scale study on repetitiveness, containment, and composability of routines in open-source projects. In *MSR*. ACM, 362–373.

[9] Aashish Yadavally, Yi Li, and Tien N. Nguyen. 2024. Predictive Program Slicing via Execution Knowledge-Guide Dynamic Dependence Learning. *Proc. ACM Program. Lang.* 1, FSE (2024), To Appear.

[10] Aashish Yadavally, Yi Li, Shaohua Wang, and Tien N. Nguyen. 2024. A Learning-Based Approach to Static Program Slicing. *Proc. ACM Program. Lang.* 8, OOPSLA (2024), To Appear.

[11] Aashish Yadavally, Tien N. Nguyen, Wenbo Wang, and Shaohua Wang. 2023. (Partial) Program Dependence Learning. In *ICSE*. IEEE, 2501–2513.