Software is ubiquitous in the modern world. Its in our cars. Its in our phones. For some reason, its even in our fridges. Software is a powerful tool and we are finding new uses for it everyday. Given how crucial it is, one would hope that software engineers would design and build systems under the same scrutiny as other, traditional, engineering disciplines, such as how a structural engineer would a bridge. The harsh reality is that this is not the case. Computer science has evolved at a breakneck pace in the last few decades, and, as a result, it has been difficult for the field of software engineering to keep up. Researchers continues to produce new program analysis techniques to improve the quality of software. However, as software systems continue to grow in size and complexity, these techniques become more computationally expensive to run. As a result, powerful program analysis tools go unused used and bugs result in the loss of money and even life [9].

Learning to Improve Program Analysis

My research is focused on improving the effectiveness of program analysis tools so they will be more widely adopted. To do this, I analyze problems in the program analysis space and identify *decision points*, a forking point in the program analysis tool where there are several ways to proceed. Depending on the way the tool proceeds, it may be able to solve the given problem faster, e.g. finding a bug sooner, increasing code coverage, etc. Traditionally, program analysis tools use a manually crafted heuristic to handle a given decision point. However, human designed heuristics are often suboptimal. In my work, I replace these heuristics with machine learning models, allowing the data to decide how best to proceed.

The key to my work lies in how a program, the input to the model, is represented. The compilers, programming languages, and program analysis communities have introduced many graph representations of programs that encode information about the behavior of the program [1, 4, 8]. In my work, programs are represented using several of these graphs, allowing the model to learn about the semantics of the program and how that relates to a given program analysis.

Traditional machine learning models cannot by default accept graphs as input. Graph neural networks (GNNs) are a class of machine learning models that use both the edges and nodes of a graph to form a prediction. In the last 5 years, GNNs have become more widely adopted as new GNN layers have been introduced which can achieve state-of-the-art performance. I leverage the power of bleeding edge GNNs to learn models which decide how program analysis techniques act at a given decision point.

In my thesis, I introduce a general framework for learning program analysis based problems, GRASP (**G**raph **R**epresentations for **A**daptive **S**emantic **P**rogram Analysis). GRASP allows users to select a set of graph representations of programs which they deem are most applicable to the program analysis problem they are targeting. Given a program, the framework automatically generates a graph representation which is a combination of the selected graphs, merging the appropriate nodes and maintaining the edges between them. Using a dataset of programs, users can train models to decide how to proceed at a given decision point. I have instantiated GRASP for several programming languages and applied it to different program analysis problems. In the following sections, I briefly describe these problems and how my work has improved them.

Formal Verification

Formal verification is the process of proving a program meets some definition of correctness. Unlike program testing, which can only show the presence of a flaw in the program, formal verification can prove the absence of violations. There are many verification techniques, and modern tools often make use of several of them. In practice, it has been shown that there is no single best verifier [2]. In [5], we introduce GRAVES, a technique for predicting how best to use a set of verifiers to efficiently verify a given program.

GRAVES uses an instantiation of GRASP for the C programming language. We identified several graphs related to the different verification techniques and trained a GNN model to predict the performance of several C program verifiers, improving the state-of-the-art. Further, we performed a study targeted at interpreting GRAVES' GNN. We found a clear pattern that showed GRAVES' GNN was most influenced by edges related to the how an error state can be reached, indicating it was learning about the general problem of verification and not overfitting to the data.

We entered GRAVES-CPA, an instantiation of GRAVES using the CPAChecker verifier framework [3], in the 2022 Competition on Software Verification (SV-Comp) and received a bronze medal in the Software Systems cate-

gory, which evaluates tools on large, realistic software projects [6].

SMT Selection

Satisfiability modulo theories (SMT) is a generalization of the boolean satisfiability (SAT) problem. Frequently, program analysis techniques will reduce a given problem to a logical formula and then use an external SMT solver to decide if the formula is satisfiable or not. Like program verification, there is a diverse set of SMT solvers and many are able to achieve state-of-the-art performance on different types of formula. Typically, program analysis tools will select a single SMT solver and use it to solve every formula it generates. However, it has been shown that a significant amount of time can be saved by dynamically choosing the solver for a given formula [7].

In [7], we introduce Sibyl, a GNN based approach to SMT selection. Unlike prior approaches to SMT selection, Sibyl is able to adapt to a given domain during training. Sibyl's GNN learns which portions of the graph are most important to solving the formula, allowing Sibyl to adapt to the types of SMT formulas different tools produces. The program analysis tools we observed spent on average 80% of their time waiting on the SMT solving. Sibyl showed a 450% performance increase across the three software engineering domains we evaluated it on, significantly reducing the cost of running these analysis tools. For this work, we received the "ACM-SIGSOFT Distiguished Paper Award" at the International Conference on Software Engineering.

Predictive Mutation Analysis

Test suites are commonly used to evaluate software systems to ensure they behave as expected. Mutation analysis is a technique for determining how strong a test suite is at detecting faults. The source code is "mutated" by making small changes to it, e.g. changing a minus to a plus, replacing a conditional statement with the value "True", etc. The test suite is run on each of these mutated systems and if a test fails, then the suite has identified the mutation, similar to a test identifying a bug. On large systems, this is a very expensive process as many mutations can be generated and running the entire test suite on each is nontrivial. This has lead to the development of "Predictive Mutation Analysis" [10], a technique where machine learning models are used to predict if a mutant survives for a given test suite.

Prior PMT approaches trained models using a set of statistics meant to summarize the behavior of the program. The statistics fail to capture the semantics of the program, making it hard for the model to identify when the test suite can catch a given mutant. We introduced Darwin, a technique which creates graph representations of both the program and the test suite, capturing the semantics of both the mutated program and the tests that are supposed to identify them. Using a GNN, Darwin can mimic the flow of information from a test to the portion of the codebase which has been mutated. This allows the model to detect what tests will exercise the mutant and how they evaluate it. Darwin is able to identify which tests will survive and the mutation score, a score denoting how strong the test suite is, more accurately than any existing PMT approach. This allows developers to identify how strong their test suite is and which sections of the codebase need to be tested more thoroughly without wasting resources running tests that will pass.

Supporting Undergraduate Research

Providing undergraduate students the opportunity to perform interesting and impactful research is one of my main goals as a future faculty member. I went to Drake University, a small liberal arts college. During my junior and senior year, I was able to work with three professors on two different research projects. This experience was invaluable and helped me decide to pursue academia. When I entered graduate school, my ultimate goal was to become a professor at a school with small class sizes that also support professors doing impactful research. This way, I can connect with my students and give them the opportunity I was afforded.

During my time at the University of Virginia (UVA), several undergraduate students came to UVA for the research experience for undergraduates (REU) program. My advisor made a concerted effort to accept students from underrepresented groups in computer science by using the Summer Research Early Identification Program from the Leadership Alliance. Computer science should be a discipline for everyone and its important to have role models you can identify with. I believe the way I can best help to improve diversity in computer science is in a personal, one-on-one method by providing opportunities to individuals from underrepresented groups.

Last summer, three students from Howard University came to our lab as REU students. My labmate, Felipe, and I served as co-advisors for one student named Alexis. Her project involved using a large language model to convert natural language specifications to formal logic. To start the project, we set clear goals and expectations so Alexis could feel she was making progress towards a final deliverable. Since she had just finished her first year of undergraduate, our advisors suggested we start with the basics and build up from there. To help get her up to speed, we set aside time to discuss topics, such as machine learning and regular expressions, and provide exercises she could do to reinforce them. We were impressed with how quickly she picked up on complex ideas and her ability to ask questions that went beyond the material we covered. At the end of the project, she produced interesting results that led us to believe, with more time and effort, could be published at a top tier conference. Its important to give students the resources they need to succeed, which means you must be flexible and touch base with them. This experience taught me how to introduce a student to complex research topics but also how to adjust to the student. Had we not followed Alexis' pace, she may have never reached these results.

My thesis provides opportunities for students to get involved from an engineering standpoint and a theoretical standpoint. Once a student has started building up their programming skill in their second or third year, they can start contributing to the engineering portion of a project, e.g. they can generate a new dataset by applying a program analysis tool to a dataset of programs. At this point the student doesn't need to understand the inner workings of the tool, instead we can discuss it as a black box that accomplishes some task. After taking classes like programming languages, compilers, algorithms, or theory of computation, students will be more prepared to get involved with the theory behind my work. We will be able to discuss program analysis techniques and program graphs in greater detail, allowing them to identify different decision points and the graphs that relate to them.

I envision working with undergraduates as a mentor/mentee relationship, not a employer/employee relationship. It is important for there to be progress towards a research goal, but this should be a bi-product of their growth. There will be times a student will not know exactly how to progress in their work. I see it as my job to help them find the way forward through conversation and gentle hints so they can have the "eureka" moments themselves and can be proud of how far they've come. While this is not always the most efficient solution, it will help them grow and become better researchers and computer scientists.

Future Work

Thanks to tools such as Github Copilot and ChatGPT, I see the role of the developer changing. These tools can help generate code, but developers must ensure that the code serves its intended purpose. To do so, they can turn to program analysis tools which can test and verify programs meet a given specification. Codebases have been growing constantly, meaning program analysis will only get harder and harder. My work will continue to target these problems.

So far, my work has used relatively standard graph representations of programs. There are graphs that maintain richer information but are more expensive to generate. I would like to look into techniques which deal with multi-objective problems that try to optimize for accuracy and performance. This can allow users to build models to better fit their needs. Further, I am interested in program analysis in general, specifically formal verification. I would like to continue working on building verification tools, like the one I submitted to SV-Comp. I can see this as a possible avenue for student projects.

A side affect of my work is that I have developed a strong interest in and understanding of GNNs. Graphs are convenient ways to represent certain types of data, such as social networks, chemical compounds, or even images. I would be very interested in applying GNNs in cross-disciplinary work. I can see collaborations in chemistry, economics, applied mathematics, or computer vision.

References

[1] ALLEN, F. E. Control flow analysis. *ACM Sigplan Notices 5*, 7 (1970), 1–19.

[2] BEYER, D. Progress on software verification: Sv-comp 2022. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems* (2022), Springer, pp. 375–402.

[3] BEYER, D., AND KEREMOGLU, M. E. Cpachecker: A tool for configurable software verification. In *Computer Aided Verification: 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings 23* (2011), Springer, pp. 184–190.

[4] FERRANTE, J., OTTENSTEIN, K. J., AND WARREN, J. D. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems (TOPLAS) 9*, 3 (1987), 319–349.

[5] LEESON, W., AND DWYER, M. B. Algorithm selection for software verification using graph attention networks. *arXiv preprint arXiv:2201.11711* (2022).

[6] LEESON, W., AND DWYER, M. B. Graves-cpa: A graph-attention verifier selector (competition contribution). In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems* (2022), Springer, pp. 440–445.

[7] LEESON, W., DWYER, M. B., AND FILIERI, A. Sibyl: Improving software engineering tools with smt selection. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)* (2023), IEEE, pp. 2185–2197.

[8] RYDER, B. G. Constructing the call graph of a program. *IEEE Transactions on Software Engineering*, 3 (1979), 216–226.

[9] SADOWSKI, C., AFTANDILIAN, E., EAGLE, A., MILLER-CUSHON, L., AND JASPAN, C. Lessons from building static analysis tools at google. *Communications of the ACM 61*, 4 (2018), 58–66.

[10] ZHANG, J., WANG, Z., ZHANG, L., HAO, D., ZANG, L., CHENG, S., AND ZHANG, L. Predictive mutation testing. In *Proceedings of the 25th International Symposium on Software Testing and Analysis* (2016), pp. 342–353.