1 Introduction

Program invariants describe properties that always hold at a program location. Examples of invariants include program pre- and post-conditions, loop invariants, and assertions. Invariants are frequently in formal verification, e.g., Hoare logic, and progarm synthesis, but have also found uses in many other programming tasks, such as documentation, testing, debugging, code generation, and synthesis [6, 25, 30].

Invariants can be discovered using static or dynamic analysis. Static technique analyze program code to infer invariants, while dynamic approaches compute invariants from program execution traces. Static analysis provides soundness guarantee but also more expensive and less scalable than dynamic analysis, which can efficiently compute invariants by sacrificing soundness (can produce spurious results that hold over observed traces but not in general).

Daikon [5,6] is a wide-known dynamic approach that infers invariants under templates. However, despite having over 200 templates, Daikon's invariants are limited to simple invariants, e.g., predicates over two variables such as $x \leq y$, and also can be spurious. Our work on **DIG** [29, 30] improves upon Daikon and targets rich numerical invariants using dynamic inference and symbolic checking. Dynamic inference allows DIG to efficiently discover many useful and rich classes of invariants from program traces, while symbolic and static checking allows DIG to check and remove spurious inferred invariants.

DIG was first published in 2012 [25] (and received the Distinguished Paper Award at ICSE'12) and since then has evolved with numerous improvements [22, 23, 25–27, 29, 30]. Multiple projects were also inspired by or built on top of DIG to support a wide-range of application domains (e.g., program termination [13], heap analysis [14], program rewriting and transformation [2], complexity analysis [9, 24], configuration analysis [20, 28], algebraic specifications [16], and even explainable AI [19, 31].

Despite the fruitful and abundance research on invariant discovery, existing techniques and tools are not widely used in practice, e.g., in industry, or even in classrooms. One of the main reason is that research prototypes were created mainly to demonstrate the feasibility of individual research ideas, and therefore are not optimized for real-world usage. For example, while DIG is effective at inferring numerical invariants, it can be slow and not scalable enough for large programs. Perhaps more importantly, invariant research and prototype tools are in generally not easily accessible to software developers and engineers, who might not be familiar with research papers and tools or have the time to learn and use them.

Intellectual Merit Our motivation to make DIG practical is in part due its adoption by Grammatech in their ambitious Mnemosyne project [18], which integrates various popular formal method tools into a single program analysis platform for practitioners[1]. While Grammatech recently appears to lost its interest in marketing the Mnemosyne (though its Gitlab repo still shows frequent updates and activities), this shows that invariant research, with better usability and performance, can be promising and used in an industry settings. This proposal aims to develop **DIG-I**ndustry to do just that. DIG-I will be more efficient and scalable, and have applications beyond just invariant discovery. It will also be modern and leverages recent AI to efficiently learn and reason about

[1] A clip [17] from Grammatech demonstrating Mnemosyne using DIG to generate assertions to aid a debugging session within the Emacs editor.

invariants. Finally, it will be intergrated to popular IDEs to improve its usability.

Specifically, in Research Component (RC) #1 we will improve the efficiency of DIG-I by transforming expensive matrix and linear constraint solving operations in DIG-I to CUDA kernels to be run efficiently on GPU, and support more useful classes of invariants by integrating existing work relying DIG-I directly into DIG's base code. In RC#2 we will extend DIG-I to integrate large language models (LLMs) to learn invariants. Finally, RC#3 improves the usability and adoption of DIG-I by devloping an LSP (Language Server Protocal) that allows DIG-I to integrate with popular IDEs and editors such as Visual Studio (VS) Code and VIM.

PI's Qualifications and Preparation PI Nguyen and Kapur were the original authors [25–27] of the DIG-I project, which is Nguyen's PhD Dissertation topic under Kapur. Since his graduation in 2016, Nguyen has extended DIG [22,23,29,30] and built various dynamic analyses and tools using DIG (e.g., [2, 9, 10, 13, 14, 16, 24]). Kapur also has developed multiple formal method techniques and algorithms for invariant inference and reasoning (e.g., [7, 11, 12]).

DIG-I will build upon DIG with inspiration from Memosys. RC#1 on GPU processing will leverage our experience in optimizing neural network verification. RC#2 on using LLM will be based on our initial experience with using LLM to generate invariants. RC#3 on IDE integration build upon our work on creating VS Code extensions [15].

The proposal includes supporting letters from Keren Zhou, a collaborator at GMU (previously at OpenAI) who will help with GPU optimizations, and Quoc-Sang Phan, a Meta/Facebook researcher who will help with industrial adoption and evaluation.

2 Broader Impacts

This proposal will bring invariant research to practice. Our collaborator at Meta will evaluate the tool, which is integrated in their VSCode IDE, and provide valuable feedback for improvement. If the project is successfully adopted by at Meta, we anticipate that it will be adopted at other organizations and projects.

The findings from this project will be used in FM and SE courses taught by the PIs and for mentoring and outreach activities. UNM is an EPSCoR state and is also one of only two universities in the U.S. that is both a Carnegie RU/VH "Very High Research Activity Institution" and a designated "Minority Serving Institution". PI Kapur is working with a Hispanic PhD student on invariant analysis, and multiple female undergraduates on various topics in FM. Nguyen has successfully involved undergraduate students in his research (e.g., [9, 10, 20, 21, 24, 29, 30]). In particular, KimHao Nguyen, who helped develop the DIG prototype [29,30] and program complexity inference [9, 10], was an undergraduate at UNL and received the Oustanding Undergraduate Researcher award at UNL for his effort on dynamic invariant inference.

3 Background: The DIG-I prototype and Preliminary Results

Fig. 1 gives an overview of DIG-I, which takes as inputs a program written in C, Java, or Java bytecode (`.class`) marked with target locations, and returns invariants found at those locations. First, DIG-I **instruments** the program and **records** its *symbolic states* using a symbolic execution tool and **concrete states** (execution traces) during program execution.

Next, DIG-I **finds invariants** by iterating between (i) *dynamic analysis*, which infers candidate equalities from concrete states obtained by running the program from sample inputs and (ii) *symbolic checking*, which checks candidates against the program using the obtained symbolic states. If

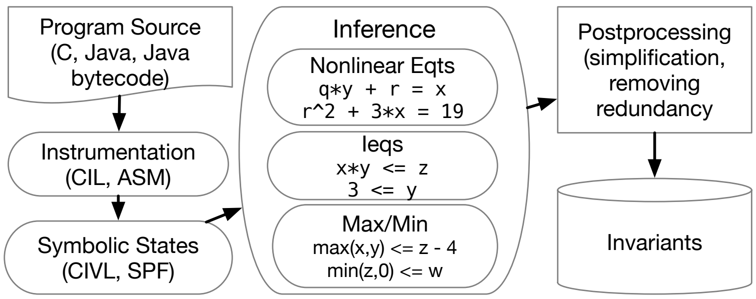

Fig. 1: DIG-I overview

```
int cohendiv(int x, int y){
  assert(x >= 0 && y >= 1);
  int q=0; int r=x;
  while(r >= y){
    int a=1; int b=y;
    while[L1](r >= 2*b){
      a=2*a; b=2*b;
    }
    r=r-b; q=q+a;
  }
  [L2]
  return q;
}
```

Fig. 2: Cohen example

Concrete States

x	y	a	b	q	r
15	2	1	2	0	15
15	2	2	4	0	15
15	2	1	2	4	7
			$\vdots$		
4	1	1	1	0	4
4	1	2	2	0	4
			$\vdots$		

Symbolic States

Path Conditions (Π_{L1})	**Variable Mappings** (σ_{L1})
$0 < y \wedge y \leq x$	$q \mapsto 0; r \mapsto x; a \mapsto 1; b \mapsto y$
$0 < y \wedge 2y \leq x$	$q \mapsto 0; r \mapsto x; a \mapsto 2; b \mapsto 2y$
$0 < y \wedge 2y + y \leq x < 4y$	$q \mapsto 2; r \mapsto x - 2y; a \mapsto 1; b \mapsto y$
$\vdots$	$\vdots$

Fig. 3: Concrete and symbolic states of the program in Fig. 2 observed at location L1 of `cohendiv`.

a candidate invariant is spurious, the checker also provides counterexamples, which are concrete states and recycled to improve dynamic inferencence and repeat the process.

Reporting too many invariants, even if they are all valid, would be a burden to the user. Thus, DIG-I uses a **post-processing step** to reduce the number of reported invariants (e.g., using SMT checking to eliminate weaker or implied invariants).

Example To demonstrate DIG-I we use the C `cohendiv` integer division algorithm in Fig. 2, marked with L1 and L2 as the locations of interest. Fig. 3 shows the concrete states observed at L1 when running the program on inputs (15,2) and (4,1) and the symbolic states at L1, which compactly encode concrete states *and* those obtained when running the program on different inputs. For this example, DIG-I returns at L1 (loop) invariants such as

$$x = qy + r; \quad ay = b; \quad b \leq x; \quad y \leq r; \quad 0 \leq q; \quad 1 \leq b; \quad 1 \leq y$$

and at L2 the (post-condition) invariants such as

$$x = qy + r; \quad r \leq y - 1; \quad 0 \leq r; \quad r \leq x$$

These relations are sufficiently strong to understand the semantics and determine the correctness of `cohendiv`. The key invariant is the nonlinear equality $x = qy + r$, which captures the precise behavior of integer division: the dividend x equals the divisor y times the quotient q plus the remainder r. The other inequalities also provide useful information. For example, the invariants at the program exit reveal several required properties of the remainder r, e.g., non-negative ($0 \leq r$), at most the dividend ($r \leq x$), but strictly less than the divisor ($r \leq y - 1$).

Results To the best of our knowledge, DIG-I is one of the state of the art techniques in numerical invariant analysis. Our experiments [30] show that DIG-I is able to infer the ground truth polynomial invariants for 106 of 108 programs obtained from SV-COMP; the next best tool can infer only 89. In many cases, DIG-I found undocumented but interesting invariants revealing useful facts about program semantics. The ability to exploit and reuse symbolic states allows DIG-I to strike a balance between expressive power and computational cost, while guaranteeing correctness, to establish state-of-the-art performance in numerical invariant inference.

4 Proposed Work

4.1 RC#1: Applications and Performance Improvement

In this RC#1, we will focus on the applications and performance improvement for DIG. While the proposed work reuses existing techniques and might appear "engineering", it is crucial to the practicality of DIG-I and will be the foundation for other RCs.

Preparation and Prior Work For applications we will start with our own work, e.g., termination and complexity analyses [9,13,24], that rely on DIG's invariants. For performance impprovement, our GPU expert and collaborator have help us profile DIG-I and determine its computation can be greatly improved.

4.1.1 Applications

DIG is often used as blackbox by other research work to infer needed invariants for various applications such as termination checking [13] and code transformation [2]. We will extend DIG-I to have such applications, so that practioner can use them directly from DIG-I. This allow these work, which were built as separate research prototypes, to leverage DIG-I's framework capabilities such as checking and refining inferred candidates.

We will integrating our own work, which was built on top of DIG, directly into DIG-I. For example, DIG-I will support additional invariants in separation logic to reason about memory usage and dynamic object creation [14] and "algebraic specifications" to capture relationships among classes and objects. DIG-I will have the ability to verify and reason about program termination/non-termination [13] (by capturing ranking functions and recurrent sets). DIG-I will also support runtime complexity analysis (e.g., this program has a $O(N \lg N)$ complexity) [9, 10, 24]. These application examples help demonstrate the modularity and extensibility of DIG-I that practitioners can adopt for their own needs.

4.1.2 Performance Improvement

Our recent work on neural network verification [3, 4] shows **GPU processing** can significantly speed up many heavy computation, in particular the constraint solving and matrix operations, which are abundant in DIG-I codebase. We will use Triton [34], an opensource project from OpenAI that allows for creating easy GPU acceleration from Python code. The main idea is (i) profiling current DIG code to see which parts can be sped up by GPU, (ii) using Triton to rewrite expensive parts using custom computation kernels capable of running at maximal throughput on modern GPU hardware, and (iii) integrating these kernels into DIG-I's codebase.

While Triton is developed for deep learning in Python, we believe that it can be used to speed up DIG-I's codebase, which is also written in Python and has similar computation such as matrix operations. In fact, we have started exploring this idea through our collaboration with Keren Zhou, the core developer of Triton. Keren has helped us profile DIG-I's codebase using his NVIDIA GPUs performance advisor tool [38] that suggests potential code optimization opportunities individual lines, loops, and functions. We are both embarrassed and excited to find that GPA determines that our current is terrible (unsurprisingly, we never attempted GPU processing for DIG) and can be speed up by 100x using Triton.

We believe that GPU (and multicore processing, which is already used extensively in DIG) will be crucial to the performance of DIG-I and makes it scale better to large programs and number of invariants, especially with the addition of new invariants and applications as described in §4.1.1.

4.2 RC#2: Integrating with LLM and other static analyzers

As shown in Fig. 1, DIG is a general framework that integrates with various techniques and tools for invariant generation and checking. Here, we will explore LLMs to synthesize candidate invariants and other checker, e.g., the Ultimate static analyzer or a fuzzing tool, to check candidate invariants.

Preparation and Prior Work AI have brought many exciting research directions for program analysis and verification. Large language models (LLMs) have been developed to aid programmers in writing proofs and lemmas, e.g., Isabelle/HOL [8], Lean [37], Dafny [39].

To evaluate this idea of LLM-based invariant generation, we have tried ChatGPT on various invariant tasks. For the `cohendiv` in Fig. 2, ChatGPT found at L1:

$$r + b = x; \quad b = 2^y; \quad r > 0.$$

Compared to the invariants produced by DIG-I, ChatGPT missed several (e.g., the crucial nonlinear $x = qy + r$, and $ay = b$), but found the exponential $b = 2^y$, which is interesting and beyond the capability of DIG (and other invariant tools). ChatGPT found $x = qy + r$ at L2, but that is the only invariant it found there. However, by tweaking the prompt query to ChatGPT, e.g., explicitly tell it to find inequalities at L2, ChatGPT was able to find the same inequalities as DIG-I.

4.2.1 LLM

We will explore invariant inference using LLM models. As mentioned above, we were able to use ChatGPT to generate interesting candidate invariants. This RC will further look into this direction.

First, we will explore modern LLMs such as those from LLama [35] and GPT3.x or 4.x from OpenAI; these LLMs have APIs that DIG-I can interact with. Given that LLMs are trained on different models, we might achieve complementary and better invariant results from using multiple LLMs together. Next, we will explore and refine prompts sent to the LLMs for better results (i.e., prompt engineering). For example, DIG-I will ask for inductive loop invariants under specific templates, e.g., polynomial relations up to degree 2 over the variable in scope, or ask to find invariants for specific purpose such as ranking functions over counting variables for termination analysis. In a sense, these are what we already use in our invariant work, but here DIG-I encodes them into the prompts to the LLM. Finally, LLM is used as our inference and its results will be checked by DIG-I's symbolic and SMT solver to ensure correctness and postprocessed to reduce redunancy. Feedback from the checker, e.g., counterexamples, or fail messages gives the opportunities for DIG-I to refine the LLM's results; for example telling it about the counterexamples or failed checks and asking it to find invariants that avoid these.

Note that while we aim to reduce the work from the user as much as possible, we believe that interaction with the LLMs will be crucial to obtain the best results. For example, after seeing the LLM results the user can refine or telling it that a certain invariant is missing or incorrect, and the LLMs can use this feedback to refine its future results.

4.2.2 Other static analyzers

DIG currently uses SMT-based symbolic execution tools (Symbolic Pathfinder for Java and CIVL for C). We will also explore using fuzzing tools, e.g., AFL++ [1] for C programs or oss-fuzz [32] from Google for C and Java, as a complementary approach to symbolic execution. Moreover, while fuzzing and symbolic execution are good at refuting spurious results, they are not as powerful as other static analyzers such as Ultimate [36] in *proving* invariants. We will integrate DIG with these tools to check candidate invariants. Note that we have already used Ultimate in our prior work [2, 13] and that the extensible and modular design of DIG allows us and end-users to easily integrate with other static analyzers.

4.3 RC#3: IDE Integration to Aid Developers

To help with usability and adoption of DIG-I, we will integrate it with modern code editors and IDEs. Specifically, we will create an an interactive, menu-based extension for popular IDEs through the LSP (Language Server Protocol) approach adopted by major code editors including Visual Studio (VS) Code, Atom, and even Emacs and VIM. We will focus on using VS Code first due to its popularity. We will leverage our experience in developing a LSP extension for VS Code that supports the development and analysis of the COOL language [15].

VS Code already has very strong C/C++ and JAVA LSP extension (with almost millions of downloads), both of which are supported by DIG-I. These provide, among others, syntax highlighting support for editing programs written in these languages. We will extend these LSP extensions by having DIG-I runs as an backend service to capture invariants and use a dropdown menu that allows the user to interact with DIG-I. For example, the user can indicate a desired location, the LSP will invoke DIG-I in the background to infer and display invariant results there. The user can also highlight a piece of code (e.g., a method or block of code) and ask if it terminates and for its runtime complexity.

4.4 Targeted Users and Evaluation

Currently, our target users are industrial developers, e.g., our collaborators at Meta who are interested in discovery progarm specifications and invariants for documentations and debugging (or even GrammaTech engineers who adopted DIG-I in Mnemosyne). To meet their needs, we will focus on developing and evaluating DIG-I for existing C and Java projects used at Meta, such as VOIP in WhatsApp and the Folly library. Our development process will be user-driven, taking into consideration the features and applications required by Meta engineers.

In addition, we also want to use DIG-I to introduce invariant generation and checking to students in our program analysis and verification courses. For example, Nguyen is teaching SWE619, a required course for the MS program at GMU where most students are professional developers and engineers. Our goal is to introduce these professionals to the power of formal methods through tools such as DIG-I and to inspire them to use and contribute to DIG-I.

5 Timeline

We estimate that DIG-I will take 24 months to complete. The first 8-month is dedicated to improve DIG-I's performance and integrating existing invariant work (RC#1). The second 8-month is for adding LLM and AI capabilities to the tool to improve expressiveness and efficiency (RC#2). The last 8-month focuses on improving usability by integrating the tool into an IDE (RC#3). Throughout the 24-month period, we will continuously evaluate and improve the tool based on our industry collaborator and other users' feedback. The project will support one GRA and we will seek REU supplement for undergraduate researchers.

6 Results from Prior NSF Support

Nguyen's most related NSF funding is CCF-1948536: *CRII: Analyzing the Linux's Kbuild Makefile*, $175,000, 4/1/2020–3/31/2023. **Intellectual Merit**: This project uses symbolic analysis to understand the build process of the Linux kernel. The DIG-I prototype adopted several optimizations from this project, and this FMiTF-T2 proposal aims to further improve the prototype with a focus on scalability, automated reasoning, and industrial adoption. **Broader Impacts**: This work produces efficient techniques for analyzing Linux Kbuild Makefiles. The PI has produced several publications [24, 29, 33] with an undergraduate student, KimHao Nguyen, on this project.

Kapur has a current NSF award *AF 1908804: Comprehensive Gröebner , Parametric GCD Computations and Real Geometric Reasoning*, $300,000. 10/1/2019–9/30/2024 (extended twice to support undergraduate student research). **Intellectual Merit**: This work focuses on using Gröebner basis to develop algorithms for parametric polynomial systems that can be helpful. While this project has no direct relation with the proposed topic, Gröebner basis has been found useful in static invariant reasoning. **Broader Impacts**: This project has produced 10 papers [**?**, **?**, **?**, **?**, **?**, **?**, **?**, **?**, **?**, **?**] and support a Hispanic PhD student and two undergraduate students.

References

[1] American Fuzzy Lop plus plus (AFL++). `https://github.com/AFLplusplus/AFLplusplus`, accessed on February 13, 2024.

[2] Y. Cyrus Liu, T.-C. Le, T. Antonopoulos, E. Koskinen, and T. Nguyen. Drnla: Extending verification to non-linear programs through dual re-writing. *arXiv e-prints*, pages arXiv–2306, 2023.

[3] H. Duong, T. Nguyen, and M. Dwyer. A DPLL(T) Framework for Verifying Deep Neural Networks. *arXiv preprint arXiv:2307.10266*, 2024.

[4] H. Duong, D. Xu, T. Nguyen, and M. Dwyer. Harnessing Neuron Stability to Improve DNN Verification. *Proceedings of the ACM on Software Engineering (PACMSE)*, FSE:to appear, 2024.

[5] M. D. Ernst. *Dynamically detecting likely program invariants*. PhD thesis, University of Washington, 2000.

[6] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao. The daikon system for dynamic detection of likely invariants. *Science of computer programming*, 69(1-3):35–45, 2007.

[7] S. Falke and D. Kapur. When is a formula a loop invariant? In *Logic, Rewriting and Concurrency: Essays dedicated to Jose Meseguer on the Occasion of His* 65^{th} *Birthday*, LNCS 9200, pages 264–286. Springer-Verlag, 2015.

[8] E. First, M. Rabe, T. Ringer, and Y. Brun. Baldur: Whole-proof generation and repair with large language models. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1229–1241, 2023.

[9] D. Ishimwe, K. Nguyen, and T. Nguyen. Dynaplex: analyzing program complexity using dynamically inferred recurrence relations. *Proceedings of the ACM on Programming Languages*, 5(OOPSLA):1–23, 2021.

[10] D. Ishimwe, T. Nguyen, and K. Nguyen. Dynaplex: Inferring asymptotic runtime complexity of recursive programs. In *2022 IEEE/ACM 44th International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, pages 61–64. IEEE, 2022.

[11] D. Kapur. Nonlinear polynomials, interpolants and invariant generation for system analysis. In *Proc.* SC^2, August 2017.

[12] D. Kapur. Invariant generation as semantic unification: A new perspective. In *UNIF 2023: The* 37^{th} *International Workshop on Unification*, July 2023.

[13] T. Le, T. Antonopoulos, P. Fathololumi, E. Koskinen, and T. Nguyen. Dynamite: dynamic termination and non-termination proofs. *PACMPL*, 4(OOPSLA):1–30, 2020.

[14] T. C. Le, G. Zheng, and T. Nguyen. Sling: using dynamic analysis to infer program invariants in separation logic. In *Programming Language Design and Implementation*, pages 788–801, 2019.

[15] L. Li and T. Nguyen. Coolio: A language support extension for the classroom object oriented language, 2023.

[16] B. Mariano, J. Reese, S. Xu, T. Nguyen, X. Qiu, J. S. Foster, and A. Solar-Lezama. Program synthesis with algebraic library specifications. *Proceedings of the ACM on Programming Languages*, 3(OOPSLA):1–25, 2019.

[17] DIG demonstration in Mnemosyne. `https://grammatech.gitlab.io/Mnemosyne/docs/`, accessed on February 13, 2024.

[18] Mnemosyne from GrammaTech. `https://grammatech.gitlab.io/Mnemosyne/docs/`, accessed on February 13, 2024.

[19] D. Nguyen, H. M. Vu, C.-T. Le, B. Le, D. Lo, and C. Pasareanu. Inferring properties of graph neural networks, 2024.

[20] K. Nguyen and T. Nguyen. Gentree: Using decision trees to learn interactions for configurable software. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pages 1598–1609. IEEE, 2021.

[21] K. Nguyen, T. Nguyen, and Q.-S. Phan. Analyzing the CMake Build System. In *2022 IEEE/ACM 44th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, pages 27–28. IEEE, 2022.

[22] T. Nguyen, T. Antonopoulos, A. Ruef, and M. Hicks. Counterexample-guided approach to finding numerical invariants. In *Foundations of Software Engineering*, pages 605–615, 2017.

[23] T. Nguyen, M. Dwyer, and W. Visser. Syminfer: Inferring program invariants using symbolic states. In *Automated Software Engineering*, pages 804–814. IEEE, 2017.

[24] T. Nguyen, D. Ishimwe, A. Malyshev, T. Antonopoulos, and Q.-S. Phan. Using dynamically inferred invariants to analyze program runtime complexity. In *Proceedings of the 3rd ACM SIGSOFT International Workshop on Software Security from Design to Deployment*, pages 11–14, 2020.

[25] T. Nguyen, D. Kapur, W. Weimer, and S. Forrest. Using dynamic analysis to discover polynomial and array invariants. In *International Conference on Software Engineering*, pages 683–693. IEEE, 2012.

[26] T. Nguyen, D. Kapur, W. Weimer, and S. Forrest. DIG: A Dynamic Invariant Generator for Polynomial and Array Invariants. *Transactions on Software Engineering Methodology*, 23(4):30:1–30:30, 2014.

[27] T. Nguyen, D. Kapur, W. Weimer, and S. Forrest. Using dynamic analysis to generate disjunctive invariants. In *International Conference on Software Engineering*, pages 608–619. IEEE, 2014.

[28] T. Nguyen, U. Koc, J. Cheng, J. S. Foster, and A. A. Porter. iGen: Dynamic interaction inference for configurable software. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 655–665, 2016.

[29] T. Nguyen, K. Nguyen, and H. Duong. Syminfer: Inferring numerical invariants using symbolic states. In *2022 IEEE/ACM 44th International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, pages 197–201. IEEE, 2022.

[30] T. Nguyen, K. Nguyen, and M. Dwyer. Using symbolic states to infer numerical invariants. *Transactions on Software Engineering (TSE)*, 2021.

[31] T.-D. Nguyen, T. Le-Cong, T. H. Nguyen, X.-B. D. Le, and Q.-T. Huynh. Toward the analysis of graph neural networks. In *2022 IEEE/ACM 44rd International Conference on Software Engineering-New Ideas and Emerging Results (ICSE-NIER)*, 2022.

[32] OSS-Fuzz: Continuous Fuzzing for Open Source Software. `https://github.com/google/oss-fuzz`, accessed on February 13, 2024.

[33] Q.-S. Phan, K. Nguyen, and T. Nguyen. The Challenges of Shift Left Static Analysis. In *International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, page to appear. IEEE, 2023.

[34] P. Tillet, H.-T. Kung, and D. Cox. Triton: an intermediate language and compiler for tiled neural network computations. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*, pages 10–19, 2019.

[35] H. Touvron, L. Martin, K. Stone, P. Albert, A. Almahairi, Y. Babaei, N. Bashlykov, S. Batra, P. Bhargava, S. Bhosale, et al. Llama 2: Open foundation and fine-tuned chat models. *arXiv preprint arXiv:2307.09288*, 2023.

[36] Ultimate. Ultimate automizer, 2020. `https://monteverdi.informatik.uni-freiburg.de/tomcat/Website/?ui=tool&tool=ltl_automizer`.

[37] S. Welleck and R. Saha. Llmstep: Llm proofstep suggestions in lean, 2023.

[38] K. Zhou, X. Meng, R. Sai, and J. Mellor-Crummey. Gpa: A gpu performance advisor based on instruction sampling. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 115–125, Feb 2021.

[39] Álvaro Silva, A. Mendes, and J. F. Ferreira. Leveraging large language models to boost dafny's developers productivity, 2024.