Analyzing Code Corpora to Improve the Correctness and Reliability of Programs

Jibesh Patra
jibesh.patra@gmail.com
University of Stuttgart

Introduction Bugs in software are expensive, unavoidable, and present a key challenge in software development. To ensure the reliability of their products, businesses such as Google, Facebook spend millions of dollars in bug uncovering programs, i.e., bug bounty. Apart from the business perspective, bugs in software can be harmful, demonstrated by incidents such as radiation treatment overdose, or patriot missile defense system failure that have cost human lives [7].

Program analysis is a widely used approach to find bugs in software that either check for commonly made mistakes based on pre-defined rules or perform sophisticated analyses of programs. These approaches can be broadly classified into two groups. 1) *Static analysis-based approaches* that reason about programs without actually executing them. 2) *Dynamic analysis-based approaches* that analyze programs during execution and reason about program behavior based on runtime values.

The goal of the dissertation summarized here is to use program analysis and novel learning-based techniques to alleviate some of the challenges faced by developers while ensuring the correctness and reliability of programs. We focus on dynamically typed languages such as JavaScript and Python for their popularity and present six approaches that leverages analysis of code corpora in aiding to solve software engineering problems. This article is divided into three parts. In the first two parts, we present software defect detection involving static and dynamic analyses respectively over code corpora. In the third part, we present a technique of input reduction leveraging large code corpora. Figure 1 provides an outline of this article.

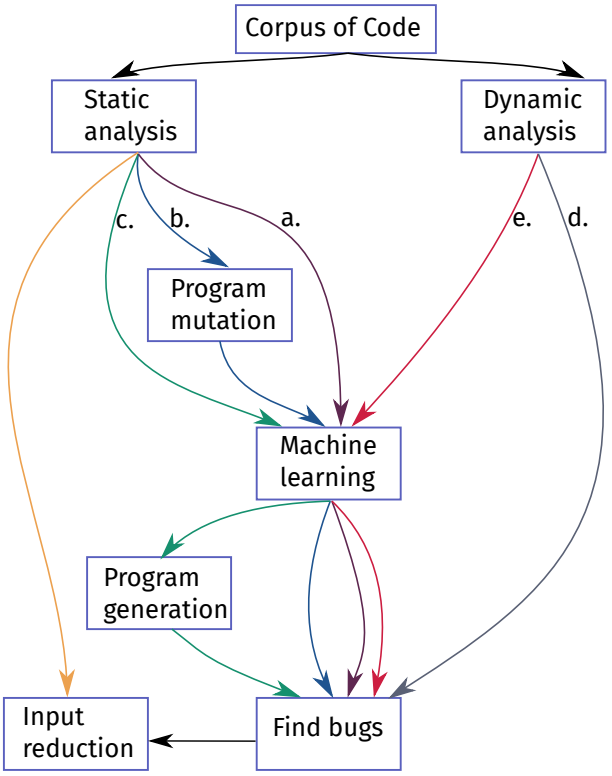

Figure 1: Overview of the contributions and the connections between them. Labels a.,b.,c. correspond to the static analysis approaches whereas d. and e. correspond to the dynamic analysis approaches presented in this article.

1 Software Defect Detection Using Static Analysis

Statically analyzing source code can uncover many interesting features about programs such as frequent code idioms, function signatures. We use static analysis to generate new programs, to seed bugs in programs, and to obtain data for training neural models.

a. Inferring Type Annotations Using Natural Language Information. Dynamically typed languages such as Python and JavaScript allow for fast prototyping. Unfortunately, this comes at a cost of many type-related issues at runtime. To mitigate some of these problems during development, we have trained neural models that aid developers by suggesting types in not-yet annotated JavaScript code and also by identifying inconsistencies in existing type annotations. We obtain the training data for the models by statically analyzing a corpus of JavaScript programs, and by extracting information such as comments, function names and parameter names.

b. Improving Bug Detectors by Semantic Bug Seeding. Machine learning on source code has myriad applications ranging from code completion [4], defect detection [5], [2], [3], to program synthesis [6]. To be effective, many such approaches need large training datasets that contain pairs of correct and buggy code. In our work, we generate a large dataset of buggy code by imitating accidental mistakes made by developers. Such mistakes are found by statically analyzing publicly accessible repositories and are used to create buggy code examples by mutating existing code. As a concrete application, we find the generated bug dataset to improve the effectiveness of an existing learning based bug detector called DeepBugs [5].

c. Fuzz Testing JavaScript Engines. In our recent survey [1], we find that like any other software, compilers are also not immune to bugs and significant research efforts have been devoted towards finding compiler bugs. JavaScript being one of the most popular languages, bugs in engines that process JavaScript programs can be disastrous. In our work, we statically analyze a large corpus of example programs to learn probabilistic, generative models that can generate new programs having properties similar to the corpus. Among others, we apply our approach to a corpus of JavaScript programs and generate large number of new programs that are further used to test JavaScript engines. During our testing, we find many inconsistencies among browsers, unimplemented language features and bugs in the JavaScript engines of Chrome[1] and Firefox[2].

2 Software Defect Detection Using Dynamic Analysis

Static analysis can only make approximations about a program. Dynamic analysis on the other hand keeps track of the runtime behavior of programs and can provide more precise information. The following summarizes our work that involve dynamic analysis to find defects in software.

d. Finding Inconsistencies Among JavaScript Libraries. Due to the lack of namespaces, when multiple JavaScript libraries are included together, they all share the same global namespace. This can cause libraries to modify and even delete each others APIs causing conflicts. In our work, we dynamically analyze a corpus of JavaScript libraries and find that even popular libraries when included together in a webpage can result in conflicts. Our work is useful for developers and users of JavaScript libraries by providing information about possible conflicting scenarios with other libraries.

e. Detecting Name-Value Inconsistencies. Identifiers play a crucial role in code understandably and maintainability. Developers strive to choose meaningful names that express the value and behavior a name is bound to. In our work, we dynamically analyze a corpus of Python programs to track assignment values to names and use them to train a neural model that predicts if a name and value fit together. The trained model is useful in finding name-value inconsistencies and bugs in real world code. Additionally, we are the first to use runtime values to train machine learning models that can find bugs in code.

[1] `https://bugs.chromium.org/p/v8/issues/detail?id=4669`

[2] `https://bugzilla.mozilla.org/show_bug.cgi?id=1231139`

3 Corpus-based Input Reduction

In addition to finding bugs, large corpora of code may be leveraged for other tasks such as reducing test inputs. We present an effective technique called Generalized Tree Reduction algorithm (GTR), to reduce arbitrary test inputs that can be represented as a tree, such as program code, PDF files, and XML documents. The efficiency of input reduction is increased by learning transformations from a corpus of example data.

References

[1] Chen, J., Patra, J., Pradel, M., Xiong, Y., Zhang, H., Hao, D., and Zhang, L. A survey of compiler testing. *ACM Comput. Surv. 53*, 1 (Feb. 2020).

[2] Li, Y., Wang, S., Nguyen, T. N., and Van Nguyen, S. Improving bug detection via context-based code representation learning and attention-based neural networks. *Proceedings of the ACM on Programming Languages 3*, OOPSLA (2019), 1–30.

[3] Li, Z., Zou, D., Xu, S., Ou, X., Jin, H., Wang, S., Deng, Z., and Zhong, Y. Vuldeepecker: A deep learning-based system for vulnerability detection. *arXiv preprint arXiv:1801.01681* (2018).

[4] Liu, F., Li, G., Zhao, Y., and Jin, Z. Multi-task learning based pre-trained language model for code completion. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering* (2020), pp. 473–485.

[5] Pradel, M., and Sen, K. Deepbugs: A learning approach to name-based bug detection. *Proceedings of the ACM on Programming Languages 2*, OOPSLA (2018), 1–25.

[6] Yahav, E. From programs to interpretable deep models and back. In *Computer Aided Verification - 30th International Conference, CAV 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings, Part I* (2018), H. Chockler and G. Weissenbacher, Eds., vol. 10981 of *Lecture Notes in Computer Science*, Springer, pp. 27–37.

[7] Zhivich, M., and Cunningham, R. K. The real cost of software errors. *IEEE Security Privacy 7*, 2 (2009), 87–90.