Research Statement

Kihong Heo

1 Goal

My research aims to develop a next-generation program analysis system for secure and reliable software. In the last decades, program analysis techniques have been adopted widely and successfully by the software industry at established companies including Facebook, Google, and Microsoft for program verification and bug-finding tasks. However, its inherent limitation—the undecidability of any non-trivial program property—still causes a large number of undesired results such as false positives (spurious warnings) and false negatives (missed bugs) in practice. From my perspective, these problems mainly occur because traditional systems are *i)* **inflexible**: the system hardly changes its strategies after deployment in spite of interacting with different inputs and environments, *ii)* **unidirectional**: the system blindly follows the designer's choices and delivers the results without interacting with the user, and *iii)* **narrow-sighted**: the system does not fully exploit a variety of available outside resources (e.g., analysis results on old versions of target programs) rather than the input program text. The ultimate goal of my research is to design a holistic AI-based program analysis system that overcomes these problems. In particular, I have been working on several sub-projects with the following criteria:

- **Adaptive reasoning system:** The system should adapt to a given task by automatically balancing between precision and cost within a given resource limit. To achieve this goal, I have developed adaptive program analysis techniques based on data-driven approaches. [Section 2]
- **Interactive reasoning system:** The system should incorporate user feedback and further refine the results. To achieve this goal, I have developed user-guided program analysis techniques that iteratively improve the quality of program analysis results based on user feedback. [Section 3]
- **Continuous reasoning system:** The system should capture semantic differences between versions of continuously evolving programs and highlight newly introduced behaviors after the change. To achieve this goal, I have developed techniques to reason about continuously evolving programs [Section 4]

The rest of this research statement will describe what I have been working on as well as what I plan to do in the future.

2 Adaptive Reasoning on Programs

In practice, designing a precise and scalable program analyzer requires a lot of heuristics for controlling their dynamic behaviors. These heuristic choices depend on various aspects such as target properties, programming idioms, or resource budgets. However, most program analyzers rely on fixed manually-designed heuristics that are usually suboptimal and brittle. Moreover, they impose a substantial amount of laborious

engineering effort on analysis designers and performance degradation for general-purpose domain-unaware program analyzers.

My research has shown that data-driven approaches can automate the development of such heuristics. The central idea is to collect data from multiple runs of the analyzer on a given codebase and learn probabilistic models that guide adaptive strategies for cost-effective heuristics. One application of such learned models is to guide the analyzer to apply computationally expensive reasoning techniques only when they are likely to improve the final precision [3, 4, 6]. I have further extended the adaptive program analysis system to control the balance between precision (i.e., producing less false alarms) and soundness (i.e., detecting more bugs) [7] where fully obtaining both of them in a single analyzer is unachievable in general.

In addition to solving the conventional problems in program analysis, I have designed relevant techniques in two different directions. The first one is to design an automatic feature generation method for data-driven program analysis [1]. All of my previous work largely relied on manually-designed features to learn probabilistic models, which requires a nontrivial amount of knowledge and efforts of domain experts. Instead, the proposed approach automatically generates small feature programs that minimally describe the characteristics of program analysis (e.g., when it is worth increasing the precision of the analysis). Intuitively these feature programs capture programming patterns whose analysis results benefit greatly from specific program reasoning techniques. The second direction targets to new applications in program analysis which are hardly achievable with purely logical techniques such as resource-aware analysis [5]. Such an analysis is aware of constraints on available physical resources, such as memory size, and adjusts its behaviors during computation in order to meet the constraint as well as achieve high precision.

All of my research on adaptive program analysis have been implemented on top of SPARROW[1], an open-source industrial-strength static analyzer for C programs, that I have been contributing to as a main developer for last 10 years [11, 12, 10, 13].

3 Interactive Reasoning on Programs

Even though a variety of techniques in Section 2 have significantly enhanced the performance of program analyzers, program analysis systems are always limited in their accuracy due to the fundamental reason, *undecidability*. This limitation forces the systems to necessarily make approximations that often lead them to report true alarms (i.e., real bugs) interspersed with many false alarms (i.e., spurious warnings). This situation imposes a big burden to users that inspect a number of false alarms.

My research has tackled this fundamental limitation by incorporating user feedback into program analysis system [14]. Alarms produced by program analyzers are often correlated as multiple true alarms share root causes, and multiple false alarms are caused by a common source of imprecision from the system. Hence, a small amount of user feedback can be leveraged to suppress a large portion of false alarms and increase the fraction of true alarms presented to the user. To achieve such an interactive system, we use the Bayesian inference technique on program analysis results by attaching a probability for each elementary logical reasoning rule. The probability represents our belief that the rule yields spurious conclusions despite having valid hypotheses because of its fundamental incompleteness. The system has a confidence score for each alarm—derived by applying such rules—to be a real bug, where the scores are able to be updated in response to new information obtained by user feedback. The generalized user feedback via the Bayesian inference significantly reduces the alarm inspection burden by the user. To further this research, I am developing an open-source library that supports the ranking system for general program analyses.

[1] https://github.com/ropas/sparrow

The effectiveness of this technique was demonstrated in open source program analysis engines in Sparrow and Petablox[2], on which I have been working on as a main developer, that target different languages (C, Java, and Android) and different properties (buffer overflow, datarace, information flow, etc.). Currently, I am extending this system to support a variety of static analyzers such as Clang static analyzer by the LLVM community.

4 Continuous Reasoning on Programs

Software often evolves continuously by integrating changes from multiple users. In this context of continuously developing software, the new versions of programs largely remain the same as the old versions but only a few parts change. Therefore, developers are primarily concerned with new bugs introduced by the current commit and are less worried about bugs in existing code which has been practically validated. However, most state-of-the-art program analysis systems do not support reasoning about continuously evolving programs (thereby, the system produces the same alarms repeatedly from unchanged parts of code), or rely on naive syntactic heuristics to suppress repetitive alarms (thereby, the system is at the risk of missing newly introduced bugs).

Recently we have developed a system for reasoning on two contiguous versions of a program [8]. Unlike the existing syntactic approach, our system captures *semantic* differences between the analysis results on the program before and after the change. The proposed differencing technique computes a confidence score for each alarm based on the likelihood of relevance to the program change, thereby enable to rank alarms with respect to the change. Our system outperforms conventional syntactic differencing in terms of manual alarm inspection burden by emphasizing newly introduced bugs and understating redundant alarms. Moreover, our study has demonstrated that the performance can be further improved by integrating interactive user feedback as described in Section 3.

In ongoing work, I am working on deploying this system on a large scale in collaboration with GitHub. Our team is planning to launch a Github CI (Continuous Integration) app that supports interactive and continuous reasoning on the Github platform.

5 Plan: AI-based Systems for Safe, Simple and Smart Programming

The ultimate agenda of my research is to realize a safe, simple, and smart programming system using learning-based approaches combined with logical techniques. In the future, I will focus on the following three sub-projects to achieve the goals:

Pattern-based security vulnerability detection I plan to develop a method to achieve *pattern-based* bug detection that can be automatically specialized to each particular deep semantic vulnerability pattern (e.g., CVE patterns). Currently, most of the semantic-based static analyzers are *property-based*—the analyzers compute the semantics of input programs and detect erroneous properties (e.g., buffer overflow) according to their logical definitions (e.g., buffer size $\leq$ buffer index). Such analyzers focus on "what bug means"; they detect a target class of bugs with a set of general logical rules that determine erroneous behaviors regardless of how the bugs look. Instead, pattern-based analyzers capture "how bug happens"; they target to only specific subclasses of bugs by capturing common buggy patterns. Therefore, they can provide more useful information to users than property-based analyzers such as the detailed explanation for each alarm

[2]https://petablox-project.github.io

or suggestions to fix them. However, most of the existing pattern-based analyzers are based on shallow syntactic patterns which are insufficient to precisely detect deep semantic bugs and also require a lot of human effort to define such patterns.

My future plan is to design a framework that automatically learns *vulnerable semantic patterns* from a large corpus. It has been hardly achievable so far because of the lack of large data (i.e., buggy source code) with precise labels (i.e., fine-grained bug locations). Fortunately, such large data sets now become publicly available via open platforms such as GHArchive[3] and Google's oss-fuzz[4]. Moreover, the emergence of new machine learning techniques such as graph neural network can enable the system to make great advances.

Safe and Reliable Program Debloating My recent research aims at developing a system for secure software in a fundamentally different way—*program debloating*. The size and complexity of modern software have been rapidly increasing, thereby causing security vulnerabilities as well as performance degradation. The goal is to build an automatic program debloating system to reverse this trend. Unlike traditional approaches such as code optimization, our system aggressively removes undesired functionalities and customizes the original program with respect to a given high-level specification. Recently we have released the system, called CHISEL[5], where we introduced a learning-based method for efficient program debloating [2]. So far, we have focused on the efficiency of the system, but CHISEL has raised many new research questions on which I am now working, such as *1)* how do we guarantee the robustness of debloated programs? *2)* how do we concisely provide the high-level specification? To address the problems, I am working on developing user-guided verification of reduced programs and designing a domain-specific language that can guide robust and efficient program debloating.

Scalable Program Synthesis The last long-term goal is to develop a scalable program synthesize system. My interest in program synthesis is mainly for two different yet coherent goals: *1)* automatically synthesizing high-performance program analyzers, and *2)* automatically fixing buggy programs. Program synthesis problems basically involve huge search spaces in the real world. Therefore, the key to success is to effectively guide the search toward solutions using data-driven and numerical optimization techniques that have been successfully developed by the machine learning community. I have witnessed the initial success through my recent work using learned probabilistic models dictating the likelihood of each program [9], and numerical relaxation techniques [15]. I envision that more advanced learning techniques can further improve state-of-the-art synthesis systems. This will open new possibilities for synthesizing highly customized program analyzers given codebases and effective patches fixing deep semantic bugs.

References

[1] Kwonsoo Chae, Hakjoo Oh, Kihong Heo, and Hongseok Yang. Automatically generating features for learning program analysis heuristics for C-like languages. *PACMPL*, 1(OOPSLA):101:1–101:25, 2017.

[2] Kihong Heo, Woosuk Lee, Pardis Pashakhanloo, and Mayur Naik. Effective Program Debloating via Reinforcement Learning. In *ACM Conference on Computer and Communications Security (CCS)*, 2018.

[3] http://www.gharchive.org
[4] https://github.com/google/oss-fuzz
[5] https://chisel.cis.upenn.edu

[3] Kihong Heo, Hakjoo Oh, and Hongseok Yang. Learning a Variable-Clustering Strategy for Octagon from Labeled Data Generated by a Static Analysis. In *International Static Analysis Symposium (SAS)*, 2016.

[4] Kihong Heo, Hakjoo Oh, and Hongseok Yang. Learning Analysis Strategies for Octagon and Context Sensitivity from Labeled Data Generated by Static Analyses. *Formal Methods in System Design*, 53(2):189–220, 2018.

[5] Kihong Heo, Hakjoo Oh, and Hongseok Yang. Resource-aware Program Analysis via Online Abstraction Coarsening. In *ACM/IEEE International Conference on Software Engineering (ICSE)*, 2019.

[6] Kihong Heo, Hakjoo Oh, Hongseok Yang, and Kwangkeun Yi. Adaptive Static Analysis via Learning with Bayesian Optimization. *ACM Transactions on Programming Languages and Systems*, 40(4), 2018.

[7] Kihong Heo, Hakjoo Oh, and Kwangkeun Yi. Machine-Learning-Guided Selectively Unsound Static Analysis. In *International Conference on Software Engineering (ICSE)*, 2017.

[8] Kihong Heo, Mukund Ragothaman, Xujie Si, and Mayur Naik. Continuously Reasoning about Programs via Differential Bayesian Inference. In *ACM Conference on Programming Language Design and Implementation (PLDI)*, 2019.

[9] Woosuk Lee, Kihong Heo, Rajeev Alur, and Mayur Naik. Accelerating Search-Based Program Synthesis Using Learned Probabilistic Models. In *ACM Conference on Programming Language Design and Implementation (PLDI)*, 2018.

[10] Hakjoo Oh, Kihong Heo, Wonchan Lee, Woosuk Lee, Daejun Park, Jeehoon Kang, and Kwangkeun Yi. Global Sparse Analysis Framework. *ACM Transactions on Programming Languages and Systems*, 36(3):8:1–8:44, 2014.

[11] Hakjoo Oh, Kihong Heo, Wonchan Lee, Woosuk Lee, and Kwangkeun Yi. Design and Implementation of Sparse Global Analyses for C-like Languages. In *ACM Conference on Programming Language Design and Implementation (PLDI)*, 2012.

[12] Hakjoo Oh, Wonchan Lee, Kihong Heo, Hongseok Yang, and Kwangkeun Yi. Selective Context-sensitivity Guided by Impact Pre-analysis. In *ACM Conference on Programming Language Design and Implementation (PLDI)*, 2014.

[13] Hakjoo Oh, Wonchan Lee, Kihong Heo, Hongseok Yang, and Kwangkeun Yi. Selective X-Sensitive Analysis Guided by Impact Pre-Analysis. *ACM Transactions on Programming Languages and Systems*, 38(2):6:1–6:45, 2016.

[14] Mukund Ragothaman, Sulekha Kulkarni, Kihong Heo, and Mayur Naik. User-Guided Program Reasoning Using Bayesian Inference. In *ACM Conference on Programming Language Design and Implementation (PLDI)*, 2018.

[15] Mukund Ragothaman, Xujie Si, Kihong Heo, and Mayur Naik. Synthesizing Datalog Programs using Numerical Relaxation. In *In submission*, 2019.