RESEARCH STATEMENT

Hong Jin Kang (hjkang@cs.ucla.edu)

My research field is in software engineering. Since software, including AI, is now ubiquitous, it is of vital importance that we improve how we build software systems. My research goal is **enhance developer productivity** through **automated approaches designed to leverage human knowledge**.

With this broad goal, I have designed **human-centric, data-driven** techniques addressing important software engineering challenges. My research investigates how to use human feedback as well as the artifacts left behind by engineers following software development processes. In my Ph.D. and postdoc research, I developed approaches to automate tasks that are error-prone for humans, as well as prevent and detect software defects.

My work has been published in highly reputable venues, including ICSE, FSE, ASE, and TSE. Papers from my research has been nominated for ACM SIGSOFT Distinguished Paper Awards. My work has real-world, practical impact. It has uncovered vulnerabilities leading to assignment of over 20 CVEs. The solutions developed as part of the industrial collaboration were deployed by our industrial partner, Veracode, to assist its security researchers. This instills confidence in the potential of my research agenda to have significant impact in the future.

Current Research

My research helps software developers in detecting bugs, preventing them, as well as improving program analyses/models. I have made innovative contributions for learning code patterns for code search and transformation, postprocessing the outputs of static analyzers, and for managing the software supply chain. These innovations can be conceptualized using three qualities:

- Active Learning with Human Feedback
- Task-Relevant Abstractions
- Software Development Process-Aware Techniques

Active Learning – soliciting and learning from human inputs

Designing solutions with a *human in the loop* establishes human-automation trust and increases the steerability of automated approaches.

Code Patterns. I developed ALP [10] and SURF [14], which infer and refine code patterns for searching for source code containing bugs related to API (Application Programming Interfaces) misuses. ALP expresses the constraints for human labelling as a logic program, and solves it to inform its queries to the human user, leading to close to 10% improvements in its effectiveness over existing techniques. To reduce the human effort required, SURF guides human users with what-if analyses to provide feature-level feedback for refining the inferred pattern. We evaluated SURF through a user study and found improvements of 20% in correctness and 30% in time saved.

Static Analysis. In studies for filtering false alarms from static analyzers, I found methodological errors that led to overoptimistic results reported by another team of researchers [8]. Subsequently, we worked together to put together a new solution that reestablishes state-of-the-art results using active learning [16]. The work demonstrates that even in low-resource settings, rather than obtaining more data, it is more important to "reflect more on that data".

Effort Reduction in Human Inspection. To provide feedback, human users face significant cognitive demands. Human cognition and effort is therefore a bottleneck in Active Learning. In my ongoing research, I developed Inspector, which allows users to identify and filter high quality data when generating a large synthetic dataset. To inspect warnings from static analyzers and suppress false alarms, I developed MARSH, which infers a code pattern representing the root cause of a false alarm (i.e., recurring code that are challenging for a precise analysis). MARSH empowers human users to simultaneously suppress false alarms, enabling a 20% speed up in the inspection of the static analysis warnings.

Designing task-relevant abstractions

Effective abstractions allow us to *simplify and manage complex problems.*

Code Patterns. I developed Coccinelle4J [13], a program matching and transformation tool. The tool uses a code representation that captures a range of relationships between program elements on the control-flow graph, allowing us to precisely describe a code transformation. This tool is the foundation that several tools for matching and transforming programs, described later, is built on.

Static Analysis. For improving a call graph analysis, often used as an upstream component to support other analyses, I guided the development of AutoPruner [3], an approach combining traditional static analysis with large language models of code, to prune errors in a call graph. This leads to improvements of 13% in identifying false positives.

Dynamic Analysis. I developed SkipFuzz [11] for fuzzing deep learning libraries. Through the course of the fuzzing campaign, SkipFuzz refines its model, expressed as a disjunction of conjuctions of logical predicates, of inputs accepted by each library function. This model informs the fuzzer's selection of inputs. The model was carefully designed to be sufficiently expressive to support its ability to make predictions about different inputs – following the model, inputs identified as similar should produce the same test outcomes when used in fuzzing. Avoiding the selection of inputs with the same outcomes reduces redundancy, and uncovers more vulnerabilities.

Software development process-aware techniques

Since software is *engineered* through rigorous and systematic processes, data generated through its development are informative and can enable powerful techniques.

Code Patterns. To mine API migration patterns for the Android API, I guided the development of AndroEvolve [1], which exploits the development practices within the Android SDK to support backwards compatibility. This allows us to identify a single code update pattern from a large space of candidate patterns.

Managing Software Dependencies. To automate the workflow of security researchers maintaining vulnerability databases, I guided the development of Chronos [4], Hermes [6], and Midas [5], collaborating with our industrial partners, including Veracode and Huawei. These techniques are designed following careful analysis of the limitations of state-of-the-art techniques, which are addressed by incorporating domain knowledge of software development processes. Next, to help developers assess the importance of a library vulnerability, I led the development of the Test Mimicry [12] technique. This performs evolutionary test case generation for a client program that depends on the library. Compared to the results of a static call graph analysis, inspecting a test case that reproduces the same program state reached by the library test case presents developers with more useful information about the exploitability of a library vulnerability.

Large Code Models. I contributed to CC2Vec [2], a distributed representation learned from code changes and their commit logs. This led to state-of-the-art performance on multiple downstream task, following my assessment of an older embedding model that could not generalize to different tasks [9]. I contributed to Compressor [7], which enables large code models (400+ MB) to be compressed into small models (3 MB), allowing their deployment on regular developer's laptops, with a negligible trade-off in model accuracy.

Future Research Plans

My research vision is to develop better human-in-the-loop approaches. Human-in-the-loop techniques offer many advantages and my research has shown the promise of Active Learning for building more powerful, human-centered solutions.

In the short-term, I plan to develop better active learning techniques by addressing shortfalls I observed during my research. I plan to explore improvements in a) what types of knowledge and feedback are solicited, as well as b) how the tools obtain the information or help human users in providing these feedback. Looking forward to the long term, I hope to lead a rethinking of automated assistants that work with software engineers.

Better Active Learning for improving developer productivity

My research has shown that automating developer activities benefit from the incorporation of human feedback and knowledge. Still, there is room to improve by expanding on the types of information utilized by active learning approaches. I will investigate multiple modalities of feedback (e.g., natural language, counterexamples), which, in appropriate contexts, can be more easily provided by a human user. Advances in deep learning have made it possible for models to combine information from multiple forms of modalities. I plan to to take advantage of these advances and investigate how to synergize feedback of different modalities interactively with a human user.

To reducing the significant amount of effort demanded from human users, I am eager to explore solutions involving the combination of symbolic techniques, such as logic programs, and neural methods. Symbolic techniques allow us to instill a strong inductive bias, allowing certain classes of problems to be solved with a much lower requirement for data. On the other hand, neural methods allow us to exploit situations where data is abundant and can produce powerful models. Carefully designing a neurosymbolic approach would allow active learning techniques that are effective without too heavy a labelling burden on a human user.

On particular domain I would like to investigate is vulnerability detection. In practice, software engineers in industry employ handwritten static analysis rules, e.g. CodeQL, for detecting vulnerabilities. These tools produce large amounts of false alarms and are challenging to scale up to more complex analyses. In research, researchers have proposed methods of using deep learning for vulnerability detection. These tools are opaque and would be difficult to interpret by human practitioners. I believe that a middle-ground can be achieved through Active Learning – interpretable static analysis rules can be inferred using deep learning models by using simple yet expressive abstractions designed to take advantage of lightweight human feedback.

Better Active Learning from improved program comprehension

Soliciting feedback effectively can be challenging because it relies on human users to accurately provide them. While active learning already strives to minimize the number of labels needed from a human annotator, my research has uncovered that this still imposes heavy cognitive demands on the human user. This points at the need to design better automated techniques to support human comprehension.

For reducing the cognitive demands required for active learning techniques, my plan involves the design and development of tools for enhancing program comprehension. Many powerful tools in the research community, such as specification miners, fuzz testers, and model checkers, can already help human users in discovering both useful and unexpected information about their programs. However, on their own, human users find it difficult to interpret these analyses as these tools are rarely made to be accessible. I aim to make these tools accessible to more people by developing new program analysis tools targeted for non-experts. For example, I aim to develop tools that generate natural language explanations for combining and interpret these analyses. With this information at their disposal, users can provide more informed and valuable feedback for Active Learning.

My work has also found many ways to utilize artifacts and metadata generated through the software development process. These artifacts and metadata would also be useful to a human user. I plan to investigate ways of using rich metadata about each program to help human users debug code.

Long-term: The Software Engineer's Apprentice

The research paper, The Programmer's Apprentice [15], written in 1982 (over 40 years ago!), predicted programming assistants today. The Programmer's Apprentice was envisioned to be an aspirational tool that communicates with programmers, assisting them in automatic programming. Today, programmers have embraced tools, such as ChatGPT, communicating with them to write and edit code.

A software engineer does more than write code. Software engineers debug, review, maintain, and deploy code. They work on challenging tasks including investigate bug reports, disclose and assess the impact of vulnerabilities, ensure that legal and privacy requirements are satisfied. These error-prone tasks are where developers need help in. An automated assistant should *collaborate* with engineers in these *challenging* aspects of software development. I seek to investigate the most effective methods through which developers can communicate with an automated assistant, how an assistant can optimally assist developers in obtaining and interpreting information, and how an assistant can most effectively utilize information provided by developers.

References

[1] Stefanus Haryono, Ferdian Thung, David Lo, Lingxiao Jiang, Julia Lawall, **Hong Jin Kang**, Lucas Serrano, and Gilles Muller. AndroEvolve: automated Android API update with data flow analysis and variable denormalization. *Empirical Software Engineering*, 2022.

[2] Thong Hoang, **Hong Jin Kang**, David Lo, and Julia Lawall. CC2Vec: distributed representations of code changes. In *42nd International Conference on Software Engineering, ICSE 2020.*

[3] Thanh Le-Cong, **Hong Jin Kang**, Truong Giang Nguyen, Stefanus Agus Haryono, David Lo, Xuan-Bach Dinh Le, and Huynh Quyet Thang. AutoPruner: Transformer-based call graph pruning. In *30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2022.*

[4] Yunbo Lyu, Thanh Le-Cong, **Hong Jin Kang**, Ratnadira Widyasari, Zhipeng Zhao, Xuan-Bach Dinh Le, Ming Li, and David Lo. CHRONOS: time-aware zero-shot identification of libraries from vulnerability reports. In *45th IEEE/ACM International Conference on Software Engineering, ICSE 2023.*

[5] Truong Giang Nguyen, Thanh Le-Cong, **Hong Jin Kang**, Ratnadira Widyasari, Chengran Yang, Zhipeng Zhao, Bowen Xu, Jiayuan Zhou, Xin Xia, Ahmed E. Hassan, Xuan-Bach Dinh Le, and David Lo. Multi-granularity detector for vulnerability fixes. *IEEE Transactions on Software Engineering*, 2023.

[6] Truong Giang Nguyen, **Hong Jin Kang**, David Lo, Abhishek Sharma, Andrew E. Santosa, Asankhaya Sharma, and Ming Yi Ang. HERMES: using commit-issue linking to detect vulnerability-fixing commits. In *IEEE International Conference on Software Analysis, Evolution and Reengineering, SANER 2022.*

[7] Jieke Shi, Zhou Yang, Bowen Xu, **Hong Jin Kang**, and David Lo. Compressing pre-trained models of code into 3 MB. In *37th IEEE/ACM International Conference on Automated Software Engineering, ASE 2022.*

[8] **Hong Jin Kang**, Khai Loong Aw, and David Lo. Detecting false alarms from automatic static analysis tools: How far are we? In *44th IEEE/ACM 44th International Conference on Software Engineering, ICSE 2022.*

[9] **Hong Jin Kang**, Tegawendé F. Bissyandé, and David Lo. Assessing the generalizability of code2vec token embeddings. In *34th IEEE/ACM International Conference on Automated Software Engineering, ASE 2019.*

[10] **Hong Jin Kang** and David Lo. Active learning of discriminative subgraph patterns for API misuse detection. *IEEE Transactions on Software Engineering*, 2022.

[11] **Hong Jin Kang**, Yunbo Lyu, Pattarakrit Rattanukul, Stefanus Agus Haryono, Truong Giang Nguyen, Chaiyong Ragkhitwetsagul, Corina S. Pasareanu, and David Lo. SkipFuzz: Active learning-based input selection for fuzzing deep learning libraries. *CoRR*, under submission, 2022.

[12] **Hong Jin Kang**, Truong Giang Nguyen, Xuan-Bach Dinh Le, Corina S. Pasareanu, and David Lo. Test mimicry to assess the exploitability of library vulnerabilities. In *31st ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2022.*

[13] **Hong Jin Kang**, Ferdian Thung, Julia Lawall, Gilles Muller, Lingxiao Jiang, and David Lo. Semantic patches for Java program transformation (experience report). In *33rd European Conference on Object-Oriented Programming, ECOOP 2019.* Schloss Dagstuhl - Leibniz-Zentrum für Informatik.

[14] **Hong Jin Kang**, Kevin Wang, and Miryung Kim. Scaling active learning of code patterns with what-if analysis (conditionally accepted). In *Conditionally Accepted at IEEE/ACM International Conference on Software Engineering, ICSE 2024.*

[15] Richard C. Waters. The programmer's apprentice: Knowledge based program editing. *IEEE Transactions on Software Engineering*, (1):1–12, 1982.

[16] Rahul Yedida, **Hong Jin Kang**, Huy Tu, Xueqi Yang, David Lo, and Tim Menzies. How to find actionable static analysis warnings: A case study with FindBugs. *IEEE Transactions on Software Engineering*, 2023.