Research Statement | Jian Wang

Software reliability remains a fundamental challenge in modern software engineering. The rapid rise of AI-assisted coding has dramatically improved development productivity but introduces a critical problem: AI-generated code cannot be inherently trusted. While AI coding tools accelerate development, they may produce code with subtle bugs, security vulnerabilities, and legal compliance issues, amplifying an already costly maintenance burden where developers spend the majority of their time fixing bugs. This unreliability of AI-generated code poses systemic risks to software quality, security, and compliance—particularly in safety-critical systems built on C/C++, which underpin infrastructure from operating systems to embedded devices.

My research goal is to **build trustworthy and semantically grounded automated program repair systems**. By leveraging the redundancy in code repositories and the reasoning capabilities of large language models, I aim to achieve this goal by developing novel **detection methods for AI-generated code and semantic-aware repair techniques**. Through the joint design of datasets, deep learning models, and semantic enhancement frameworks, I address the reliability crisis in AI-assisted software development, extending the capabilities of **automated program repair beyond pattern matching to genuine program understanding** in terms of semantic reasoning and runtime behavior.

My research is situated at the intersection of software engineering, machine learning, and program analysis. **From a foundational perspective, I employ deep learning and retrieval-augmented methods as the core driving force, construct high-quality benchmarks and datasets with executable test cases, and apply these systems to areas ranging from code security (such as AI-generated code detection and adversarial robustness) to software reliability (such as automated fault localization and patch generation)**. All three components—machine learning algorithms, benchmark construction, and practical applications—are systematically considered and jointly optimized. My key research approaches and topics are as follows:

- **Detection approaches to identify AI-generated code for quality assurance and compliance**. This is achieved by evaluating state-of-the-art detectors and developing fine-tuning-based methods from a data-driven learning perspective. Typical topics include:

  a) [Natural Language → Code] Comprehensive evaluation of 13 AIGC detectors (6 commercial, 7 open-source) on 2.23M samples across code summarization, code generation, and Q&A tasks [ASE 2024], and
  b) [Domain Adaptation] Fine-tuning strategies that substantially improve detector performance within the same domain while revealing generalization challenges across different software activities.

- **Benchmark construction approaches to enable reproducible evaluation and advance repair research**. This is achieved by curating large-scale datasets from real-world repositories with executable test cases. Typical topics include:

  a) [Java → C/C++] First large-scale executable benchmark Defects4C with 248 high-quality buggy functions and 102 vulnerable functions curated from 9M commits [ASE 2025],
  b) [Evaluation Infrastructure] Comprehensive evaluation of 24 state-of-the-art LLMs revealing significant performance gaps in C/C++ repair compared to Java benchmarks, and
  c) [Dataset Curation] In-the-wild dataset Ratchet-DS for evaluating deep learning-based repair approaches in realistic settings [ISSRE 2024].

- **Semantic enhancement approaches to improve LLM reasoning beyond pattern matching**. This is achieved by incorporating dynamic semantic information as a plug-and-play enhancement to training and inference procedures. Typical topics include:

  a) [Static → Dynamic] Generic framework for integrating execution traces into code task-relevant prompts [EMNLP 2025 Findings],
  b) [Training Enhancement] Investigation of trace-based semantic information for supervised fine-tuning of Code LLMs, and
  c) [Inference Enhancement] Exploration of semantic signals for test-time scaling, revealing surprising limitations of current approaches in semantic reasoning.

My research philosophy is driven by real-world impact and systematic evaluation. By identifying critical gaps in software reliability and breaking them down from first principles, I engage deeply with topics that are intellectually challenging and practically significant. I will highlight three research topics to demonstrate my philosophy. These three topics share a common spirit of applying deep learning and semantic reasoning approaches to enhance automated program repair. Over time, these topics

have posed increasingly challenging questions, involving AI-generated code detection without explicit training data, automated repair for under-resourced C/C++ programs, and semantic enhancement for genuine program understanding beyond surface patterns.

Automated Program Repair | Localizing and Fixing Bugs with Deep Learning

Software developers spend the majority of their time debugging and maintaining code rather than writing new features. There is a fundamental trade-off between development speed and code quality because of the growing complexity of modern software systems. In particular, AI-assisted coding tools can generate thousands of lines of code rapidly, but this code may contain subtle bugs that are difficult to detect. How to automatically localize faults and generate correct patches is a major challenge, i.e., identifying buggy statements without bug reports and synthesizing fixes that preserve program semantics. Take fault localization as an example, vanilla approaches require bug-triggering test cases which are not always available in early development phases. I asked the question – **Can we exploit the redundancy in program structures and learn from historical patches to repair bugs automatically?** Yes. Prior work has explored this direction using sequence-to-sequence models from the NLP community and built deep learning-based repair tools capable of fixing simple bugs. However, these approaches struggle with fault localization without test cases and context identification for patch generation, which I refer to as the dual challenge in Automated Program Repair [ISSRE 2024].

I have made two core contributions to deep learning-based Automated Program Repair (APR). Despite its promise of automating bug fixing, APR has trade-offs among fault localization accuracy, patch generation quality, and generalization to **real-world bugs**. My first contribution is a dual deep learning framework RATCHET that addresses both fault localization and patch generation without requiring bug-triggering test cases or bug reports [ISSRE 2024]. The fault localization component (Ratchet-FL) uses a BiLSTM model to learn features directly from code structure, achieving 39.8-96.4% accuracy without additional artifacts. The patch generation component (Ratchet-PG) employs a retrieval-augmented transformer that learns from historical patches, achieving 18.4-46.4% repair accuracy. This has brought practical promise to the APR community that deep learning systems can localize and fix bugs in realistic development scenarios where test cases are unavailable.

My second contribution is the first large-scale executable benchmark Defects4C for C/C++ program repair [ASE 2025]. Despite C/C++'s widespread use in safety-critical systems, research on C/C++ repair has been severely constrained by the lack of high-quality benchmarks comparable to Defects4J for Java. The idea is to construct a comprehensive dataset from real-world repositories with executable test cases, enabling rigorous evaluation and retraining of learning-based approaches. I curated 248 high-quality buggy functions and 102 vulnerable functions from 9M bug-relevant commits, all paired with test cases for reproduction. Using Defects4C, I conducted the first comprehensive evaluation of 24 state-of-the-art LLMs on C/C++ repair, revealing significant performance gaps compared to Java and highlighting the critical need for better semantic understanding in current models.

I have also contributed to understanding the limitations of current APR approaches: an in-the-wild dataset Ratchet-DS demonstrating the challenges of real-world bug complexity [ISSRE 2024], and empirical findings showing that existing LLMs struggle with C/C++ repair due to insufficient semantic reasoning capabilities rather than mere lack of training data [ASE 2025].

AI-Generated Code Detection | Identifying Untrusted Code with Deep Learning

Intrigued by the rapid adoption of AI coding assistants and their impact on software quality, I asked the question – Can we reliably detect AI-generated code to enable targeted quality reviews and ensure compliance? Yes. With comprehensive evaluation of existing detectors and domain-specific fine-tuning strategies [ASE 2024]. This question is inspired by real-world concerns about academic integrity, Stack Overflow policies against AI-generated answers, and the need for licensing compliance when AI models may have memorized copyrighted code. Like code plagiarism detection, AI-generated code detection does not have ground truth labels in production environments and must generalize across different code generation models and software activities.

I conducted the first comprehensive empirical study evaluating AIGC detectors on code-related content [ASE 2024]. I selected three state-of-the-art code LLMs (GPT-3.5, WizardCoder, CodeLlama) and created a dataset of 2.23M samples across popular software activities: Q&A (150K), code summarization (1M), and code generation (1.1M). I evaluated 13 AIGC detectors comprising 6 commercial and 7 open-source solutions. My findings reveal that AIGC detectors perform significantly worse on code-related data than on natural language, with detection accuracy dropping substantially when applied to code tasks. Despite limitations in current detectors, I demonstrate that fine-tuning can enhance detector performance, especially for content within the same domain, though generalization across different software activities remains a fundamental challenge.

From the practical impact side, my goal to provide empirical guidance for developing better AI-generated code detectors directly addresses industry needs for code review automation and compliance verification. From the research methodology side, my systematic evaluation framework establishes best practices for assessing detector performance across multiple dimensions: different generation models, various software activities, and domain adaptation scenarios.

Semantic Enhancement for Code LLMs | Beyond Pattern Matching to Program Understanding

Carrying the momentum of improving automated program repair through better semantic understanding, I am curious whether we can enhance LLMs with dynamic semantic information to achieve genuine program comprehension. This has led to a systematic investigation of execution trace-based information for Code LLMs, where I designed a generic framework for integrating semantic signals into both training and inference [EMNLP 2025 Findings].

I apply the same methodology of data-driven learning in automated program repair and adapt it to the semantic enhancement problem. Specifically, I view execution traces as dynamic semantic signals that reveal actual runtime behavior, complementing the static code that LLMs typically process. To systematically study the usefulness of semantic information, I designed a generic framework that supports integrating execution traces into task-relevant prompts. This framework enables controlled experiments across different integration strategies: during supervised fine-tuning (SFT) to improve model capabilities, and during inference for test-time scaling. One unique characteristic is that my framework disagrees with previous optimistic claims about semantic information, revealing through comprehensive experiments that trace-based information has surprisingly limited usefulness for both SFT and test-time scaling of Code LLMs. This finding suggests that current models lack the architectural capacity to effectively utilize semantic information, rather than simply needing more training data.

Future Research

My research goal of building trustworthy and semantically-grounded automated program repair systems has far-reaching prospects in novel datasets, fundamental reasoning methods, large language models, and applications across software engineering domains and even beyond. Notably, fields such as program synthesis, software testing, and code understanding share analogous challenges with automated program repair, utilizing similar deep learning and semantic reasoning strategies across various tasks. I plan to maintain a medium-sized research group. I will mentor them to ask valuable questions grounded in software engineering fundamentals and conduct impactful research for software reliability and AI safety.

Software Engineering Fundamentals | Dataset Construction and Semantic Reasoning from First Principles

From first principles, my goal of building trustworthy automated program repair can be decomposed into sub-problems in datasets (benchmark construction and curation), reasoning (semantic understanding and program comprehension), and applications (security, reliability, and compliance).

For the dataset side, I plan to unify benchmark construction across programming languages through systematic curation from real-world repositories. With the growing availability of open-source code and bug-fixing commits, there is an opportunity to construct comprehensive benchmarks that enable reproducible evaluation. This is achieved by developing automated mining pipelines that extract high-quality bugs with executable test cases, filtering criteria that ensure bug reproducibility, and validation procedures that verify patch correctness. This unified dataset construction methodology can be directly applied to other software engineering problems such as vulnerability detection and code review automation. Besides, following my trend of addressing increasingly challenging programming languages, I will continue exploring opportunities for under-resourced languages such as Rust, Go, and domain-specific languages used in embedded systems. For instance, constructing a comprehensive benchmark for Rust would enable evaluating whether modern memory-safe languages benefit from different repair strategies.

For the reasoning side, I plan to unify semantic representations across static and dynamic program analysis. Using execution traces, program slices, and symbolic execution as complementary views of program behavior, neural networks can effectively capture semantic information in a task-agnostic manner. The insight is to embed structured semantic information into model architectures, such that the latent representations capture genuine program understanding rather than surface-level patterns. Additionally, I plan to leverage large language models as generic "program reasoners" and use them as strong semantic priors for downstream tasks. I am exploring neurosymbolic approaches that combine LLM reasoning with formal verification, bridging the gap between statistical learning and logical correctness.

Repair-inspired AI for Software Engineering | Semantic Understanding for Multi-Task Code Intelligence

Large language models trained on millions of code repositories have demonstrated impressive capabilities in code generation. For comprehensive software engineering assistance, understanding across multiple tasks and modalities is desired. On one hand, such multi-task understanding requires expensive annotation; on the other hand, training multi-task models suffers from the same curse of capacity as the repair problem. Can we use the strategy of semantic enhancement to improve multi-task code understanding?

Imagine all code-related tasks benefit from shared semantic representations that capture program behavior. Note that the semantic space can still be efficiently represented through execution traces and program analysis. Recall that my work on semantic enhancement reveals limitations of current approaches [EMNLP 2025 Findings], we can use these insights to design better architectures that explicitly model semantic information rather than treating it as auxiliary input. This idea is intriguing because like automated program repair, multi-task code understanding requires genuine comprehension of what programs do, not just pattern matching on syntax. The encoder part can incorporate structured semantic representations through specialized attention mechanisms, and the decoder part can be conditioned on semantic constraints to ensure output correctness.

After training such a semantically-enhanced multi-task model, we get improved performance across code generation, program repair, vulnerability detection, and code summarization as important byproducts. Additionally, the same idea works for any code understanding task with executable specifications: with or without test cases, static or dynamic analysis. We can get a unified model that handles multiple tasks and shares semantic understanding across them. If the semantic representations are differentiable, the model architecture can be co-optimized along the training process since we can back-propagate gradients through both code and execution paths.

AI for Software Security and Reliability | Trustworthy AI-Assisted Development

As a researcher working on practical software engineering problems, I am excited to use my expertise and collaborate with industry partners to bring real-world impact and advance software quality. In particular, my research on AI-generated code detection and automated program repair can contribute to building safer and more reliable software systems, which falls into the category of trustworthy AI for software engineering.

Conventional software development processes usually trade off development speed for code quality, such as rushed releases versus thorough testing, or rapid prototyping versus security audits. Using the above-mentioned unified semantic reasoning framework, compromised software quality can be restored by incorporating automated quality checks and repair mechanisms into the development pipeline. There are challenges to be tackled by working with industry partners, especially how to balance automation with developer control and ensure that automated repairs preserve program intent.

Take AI-assisted coding as an example, conventional workflows generate code snippets and leave quality assurance entirely to developers. That means developers must review potentially hundreds of AI-generated lines, and the cognitive load is 10-100 times higher than traditional code review. This high review burden is similar to inspecting dozens of patches in automated program repair. It can be addressed by developing targeted detection methods that flag AI-generated code for extra scrutiny [ASE 2024], and automated repair tools that fix common bugs before human review [ISSRE 2024, ASE 2025]. I foresee these techniques will dramatically improve software reliability in the age of AI-assisted development and enable safer adoption of AI coding tools in safety-critical domains.

References

[EMNLP 2025] **Jian Wang**, Xiaofei Xie, Qiang Hu, Shangqing Liu, and Yi Li. Do Code Semantics Help? A Comprehensive Study on Execution Trace-Based Information for Code Large Language Models. In Findings of the Association for Computational Linguistics: EMNLP, 2025.

[ASE 2025] **Jian Wang**, Xiaofei Xie, Qiang Hu, Shangqing Liu, Jiongchi Yu, Jiaolong Kong, and Yi Li. Defects4C: Benchmarking Large Language Model Repair Capability with C/C++ Bugs. In Proceedings of the 40th IEEE/ACM International Conference on Automated Software Engineering (ASE), 2025.

[ASE 2024] **Jian Wang**, Shangqing Liu, Xiaofei Xie, and Yi Li. An Empirical Study to Evaluate AIGC Detectors on Code Content. In Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering (ASE), 2024.

[ISSRE 2024] **Jian Wang**, Shangqing Liu, Xiaofei Xie, Jing Kai Siow, Kui Liu, and Yi Li. RATCHET: Retrieval Augmented Transformer for Program Repair. In Proceedings of the 35th International Symposium on Software Reliability Engineering (ISSRE), 2024.

[ACM LCTES 2024] Shangqing Liu, Wei Ma, **Jian Wang**, Xiaofei Xie, Ruitao Feng, and Yang Liu. Enhancing code vulnerability detection via vulnerability-preserving data augmentation. Proceedings of the ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES), 2024.

[ACM TOSEM 2023] Tianlin Li, Xiaofei Xie, **Jian Wang**, Qing Guo, Aishan Liu, Lei Ma, and Yang Liu. Faire: repairing fairness of neural networks via neuron condition synthesis. ACM Transactions on Software Engineering and Methodology (TOSEM), 2023.

[ACM TOSEM 2022] Xiaofei Xie, Tianlin Li, **Jian Wang**, Lei Ma, Qing Guo, Felix Juefei-Xu, and Yang Liu. NPC: neuron path coverage via characterizing decision logic of deep neural networks. ACM Transactions on Software Engineering and Methodology (TOSEM), 2022.

[ICML 2021] Xiaofei Xie, Wenbo Guo, Lei Ma, Wei Le, **Jian Wang**, Linjun Zhou, Yang Liu, and Xinyu Xing. Automatic RNN Repair via Model-based Analysis. International Conference on Machine Learning (ICML), 2021.

[NeurIPS 2020] Qing Guo, Felix Juefei-Xu, Xiaofei Xie, Lei Ma, **Jian Wang**, Bing Yu, Wei Feng, and Yang Liu. Watch out! Motion is Blurring the Vision of Your Deep Neural Networks. Advances in Neural Information Processing Systems (NeurIPS), 2020.

[IJCAI 2020] Run Wang, Felix Juefei-Xu, Lei Ma, Xiaofei Xie, Yihao Huang, **Jian Wang**, and Yang Liu. FakeSpotter: A Simple yet Robust Baseline for Spotting AI-Synthesized Fake Faces. International Joint Conference on Artificial Intelligence (IJCAI), 2020