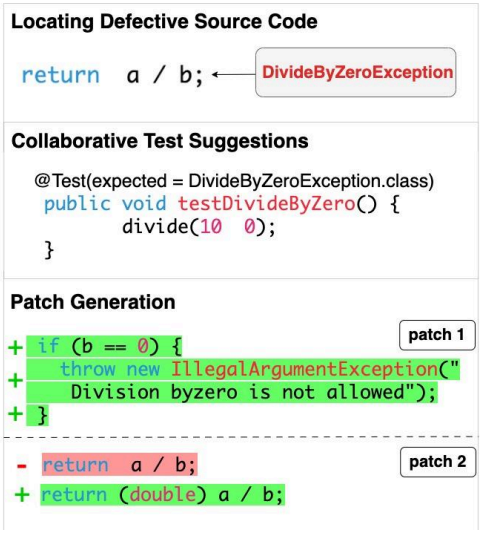
Research Statement

He Ye

My research in **Software Engineering (SE)** centers around constructing automated, effective, and trustworthy developer support systems for debugging and repairing software bugs and vulnerabilities with AI-augmented assistance. The rising demand for software products has put developers under growing pressure to repair software defects. While the common expectation is that developers should swiftly solve code defects, the reality is that bug location and repair consume a significant portion of developers' working hours, often surpassing the time dedicated to other requirements such as functionality and performance. Currently, the available developer support for bug locating and repairing is primarily manual, leading to ineffective software bug repairs and escalating overall development costs.

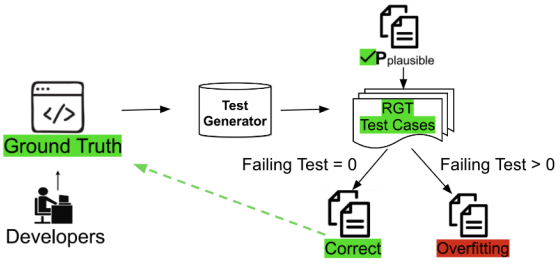
My goal is to offer one coherent solution for repairing software bugs and vulnerabilities at the source code level by integrating AI-augmented code assistance to offer easy-to-access bug-fixing support. Our research supports software debugging and repair in multiple aspects: 1) Locating Defective Source Code: Leveraging deep learning to identify flawed source code, and issuing warnings to developers. 2) Collaborative Test Suggestions: We offer collaborative test suggestions, facilitating a deeper understanding of program behaviour and aiding in specification processes. 3) Patch Generation: Our work generates timely patches, aiding developers in adhering to best practices and ensuring code quality.



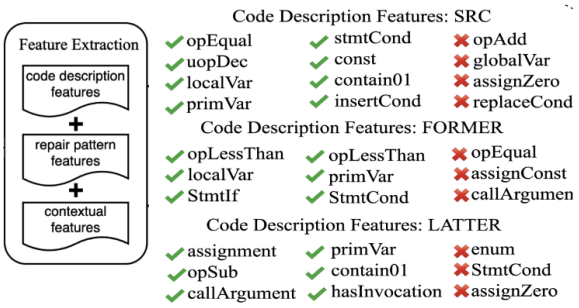
As a Software Engineering scholar, I do research to tackle fundamental software problems (e.g., bugs and testing) to ensure software quality and robustness, this research field is called automated program repair (APR). My research involves understanding bug characteristics with large-scale data analysis, and bug repairing with AI-based approaches. I conduct empirical studies using methods of data collection and data analysis to study the characteristics of defective code [1, 2]. I then design, build, evaluate, and deploy a series of techniques to repair software bugs and vulnerabilities in an automatic manner [3, 4, 5, 6, 7]. My research has yielded publications at top-tier SE venues, such as ICSE, ASE, TSE, FSE, and EMSE. My work also makes a broader impact through model release and outreach to the industry. I have successfully collaborated with researchers from KTH, CMU, TU Delft, Hong Kong Polytechnic University, and Universitat Politècnica de Catalunya-BarcelonaTech. My work has been recognized with an IEEE distinguished paper award to win the ASE 2023 Industry Challenge Competition.

Patch Assessment Approaches to Improve The Precision of APR by Ruling Out Incorrect Patches

As I set out on my research about providing high-quality patches to developers to automatically repair software bugs in 2019, most prior APR generates a great number of candidate patches and many of them are incorrect. APR considers a set of the test suite as program behavior specifications that aims to select those patches that pass all test specifications, which results in many candidate patches being selected and most of them merely overfitting to the



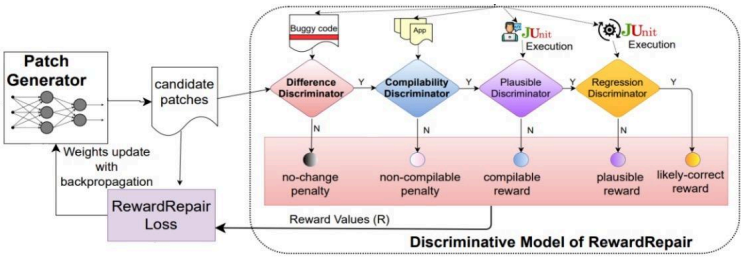
test specifications yet incorrect. This affects both scientific evaluation and practical usage of APR research. For scientific evaluation, researchers rely on manual analysis to evaluate the correctness of patches generated by their own models, which could be difficult, biased, and time-consuming. For practical usage, providing way more patches to developers will waste developers' time and their patience with APR research. In our EMSE 2021 paper [3], I first identified and proved the bias problem exists in manual patch assessment with large-scale patch assessment results to confirm the misunderstandings from individual authors, then we propose to assess the correctness of patches with automatically generated independent test suite via Evosuite and Randoop based on Random Ground Truth (RGT) tests. With this approach, 72% of manual patch assessment work could be replaced automatically. We have open-sourced RGT test cases generated for six projects and have seen the tests gaining adoption by researchers in assessing the correctness of APR patches [1].



Automatically generating an independent test suite is effective for scientific evaluation to rule out incorrect patches, yet, this approach requires a ground truth program to construct oracles for test generation. For practical usage of APR, such a ground truth patch would not be available. To address the real-world problem that too many incorrect patches are generated and provided to developers, in our TSE 2022 paper [4], we study the code features and train a probability model ODS to distinguish correct patches from incorrect patches. The hypothesis of this approach is based on code features capturing universal correctness properties for classifying APR patches. Our evaluation shows that with the learning code feature, the learned model can largely identify incorrect patches and rule them out. We have open-sourced ODS [2].

Execution-Aware Neural Program Repair Model

I started to realize generating high-quality patches may be the first important pre-step of patch correctness assessment, after doing two works of patch assessment. In the initial work of adapting AI for code (AI4CODE), they considered source code as natural languages and surprisingly little research looked into the execution information of generated code to improve the effectiveness of program repair models. The prior literature applies neural machine translation (NMT) and aims to transform buggy code snippets into the correct code snippets. Yet, these approaches largely neckage the nature of programming languages (e.g., structures and dependencies). Our philosophy is "code is not a string, and code can be executed at runtime". A minor difference in static tokens will make a big difference during runtime. In our ICSE 2022 paper [5], we identified a crucial challenge in the AI4Code repair model: unawareness of execution information that leads to low compilable rate and low precision of generated patch. These issues suggest the need for embedding execution information of generated patches as the feedback of the repair model, which inspired our work, RewardRepair, to help repair models aware of the



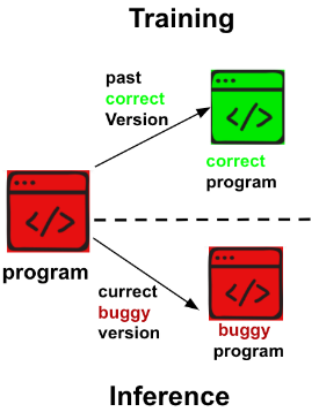
[1] https://github.com/ASSERT-KTH/drr
[2] https://github.com/ASSERT-KTH/ODSExperiment

execution states of their generated patch [5]. RewardRepair consists of two components: a patch generator and a patch discriminator, where a patch generator is responsible for generating fix code to correct buggy programs and a patch discriminator compiles and executes the generated patches, and eventually offers real-time patch quality feedback based on the execution results. Over comprehensive experiment shows RewardRepair improved the overall quality of patch generation by being aware of the execution status of AI-generated code. We have open-sourced RewardRepair [3].

Leveraging Project-Specific Knowledge To Improve Program Repair Model

During our previous work, we observed that Large language models (LLM) are good at learning the general knowledge of code transformation from real-world bug-fix pair data that exist in diverse open-source projects. However, they do not capture domain knowledge for specific projects during the testing phase, where the domain knowledge is specific project logic at a higher level and customize classes, expressions, and variables at the code structure level. There exists a distribution difference between the learned general knowledge during the training phase and the project-specific knowledge testing phase.

In our ASE 2022 paper [6], we mitigate such a distribution gap between general knowledge and domain-specific knowledge by including training samples artificially generated from testing projects. Our work proposed a self-supervised training approach called SelfAPR, which generates training samples by injecting bugs into *a previous version* of the program being repaired, and trains a deep-learning model to identify and fix the injected bugs. The learned model is used to repair a future buggy version of the same project, enforcing the neural model to capture project-specific knowledge. In addition, SelfAPR executes all training samples and extracts and encodes test execution diagnostics into the input representation, steering the neural model to fix the kind of fault. SelfAPR improves existing work from two perspectives: 1) by including the application domain of the program being repaired, and 2) by including the fault type being repaired. This is different from the previous work based on purely mined past commits from diverse open-source projects. We have open-sourced SelfAPR. We have open-sourced SelfAPR [4].



Iterative Neural Repair For Multi-Location Patches

LLMs yield remarkable results in AI4Code tasks. However, like humans, LLMs do not always produce the best outputs on their first try. This motivates us to design an approach to refine the initial outputs of LLMs iteratively based on the feedback from previous attempts. In our ICSE 2024 paper, we present ITER [7], an approach that enhances the initial outputs of LLMs through iterative feedback and refinement. The core concept of ITER involves generating an initial output with an LLM, using the same model to provide feedback on its output and then refining itself in an iterative manner. In the task of APR, ITER proves effective in enhancing generated patches for both single-location bugs and multi-location bugs. In the case of single-location bugs, ITER operates on the principle of refining partial patches until they become plausible and correct. For multi-location bugs, ITER employs an iterative process of re-executing fault localization based on partial patches. This allows it to continually update the bug status, ensuring a more accurate and up-to-date understanding of the buggy state of programs.

---

[3] https://github.com/ASSERT-KTH/rewardrepair
[4] https://github.com/ASSERT-KTH/SelfAPR

Future Research Agenda

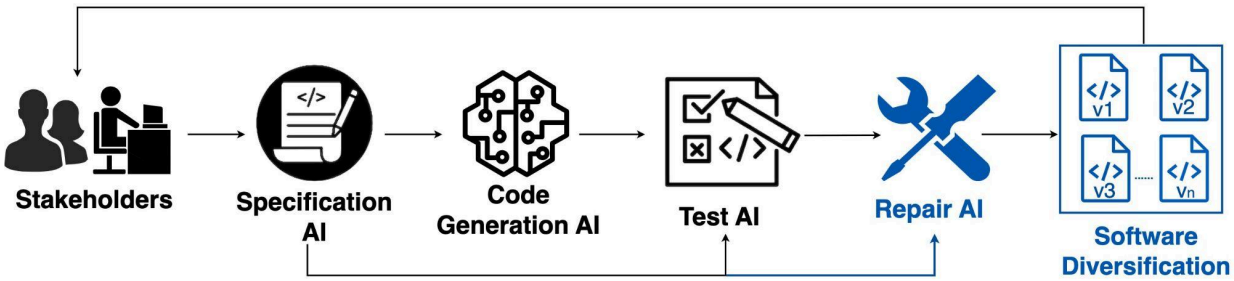
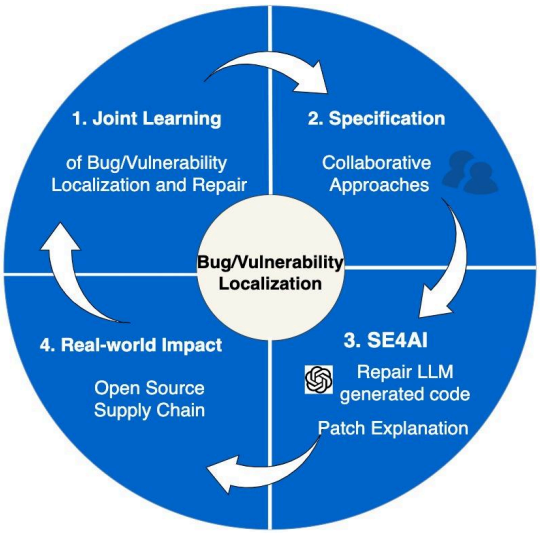


Figure 1: An overview of the future research agenda in the AI-integrated software development process

I intend to leverage my core strengths in software engineering (SE) to play a significant role in developing robust and efficient software support. My approach will shift from thinking solely about the AI model to considering its role in an entire complex system and working with multiple components. My goal is to support novel AI-integrated software development processes in the future, as shown in Figure 1. I believe that much of the design, requirement analysis, and prototyping work could be replaced by AI, focusing on specification elicitation (Specification AI) and working closely with stakeholders. Additionally, a significant amount of coding work can be done by code generation AI. Despite their power, our research focuses on the extent to which we can trust the AI-generated code and use it in production. To achieve this, our future work will focus on how to repair the AI-generated code to make it trustworthy. In another aspect, future code generation AI may generate the same or similar code for all end users, posing a threat of security attacks. Our work focuses on creating software diversification to protect against these security attacks. In summary, my future work focuses on 1) building a **Repair AI** to ensure the quality of AI-generated code and 2) implementing **Software Diversification** to enhance security and privacy.

Repair AI

Joint learning of fault localization and repair. The precise identification of faulty locations is a critical step in program repair. However, existing approaches often treat fault localization and repair as separate tasks, considering fault localization merely as a preprocessing step that generates a ranked list of suspicious statements. Unfortunately, this methodology offers no guarantee that the faulty code snippets are consistently ranked at the top. As highlighted in our ICSE 2024 paper [7], the separation of fault localization and repair proves inefficient, particularly in addressing multi-location bugs. The first step of our future work of Repair AI aims to integrate these two components into a joint learning model, thereby enhancing the precision of fault detection and repair.



Specification. Repair AI requires clear specifications to define the intended program behavior accurately, ensuring its correctness. While test cases have been commonly used as specifications in existing work, they often prove inadequate in the real world, leading to the generation of many overfitting patches [3, 4]. To enhance patch precision, we aim to explore collaborative approaches to enrich program specifications comprehensively. This enrichment could involve input from various

sources such as stakeholders, Specification AI, bug reports, discussion information present in pull requests, and historical data. A robust specification greatly aids in guiding the program toward the correct behavior. The fundamental idea is to revisit all information from the inception of design and requirements collaboratively with stakeholders.

SE4AI and enabling patch explanation. AI for code generation represents the future trajectory of software development. LLMs like CodePilot and ChatGPT offer substantial support in this domain. By leveraging the provided function descriptions, LLMs can effortlessly produce professional code. However, one fundamental issue arises. Can humans fully trust the quality of code generated by AI? To address this concern, our upcoming work proposes employing program repair techniques to ensure the trustworthiness of AI-generated code — Software Engineering (SE) for AI. In addition, we also call for explaining the behaviors of patches to developers or stakeholders, involving humans in the AI process. With explanations, humans can place greater trust in the model's suggestions or corrections when they comprehend the underlying logic. Explanations also assist developers in identifying errors or inaccuracies in the model's output, enabling them to diagnose and rectify issues more effectively by gaining insights into the model's decision-making process.

The real-world impact of Repair AI. Recent program repair work limits itself to benchmark evaluation. In my future work, I aim to showcase the real-world impact of repair AI on open-source software supply chains and other domains. There are several avenues we can explore. For instance, repairing open source projects in collaboration with OSS-Fuzz (Open Source Software Fuzzing). OSS-Fuzz automates the fuzz testing process, a dynamic analysis technique involving injecting random or unexpected inputs into a software application to detect vulnerabilities. As of the current writing, OSS-Fuzz has identified 31,000 bugs and vulnerabilities, with only approximately 6,000 being manually addressed by developers. This highlights a significant gap in addressing and repairing the remaining bugs and vulnerabilities. Integrating our research with OSS-Fuzz to automatically repair bugs and vulnerabilities holds significant potential in reducing software development time for developers.

AI-based Software Diversification

In the era of AI, when all developers use LLMs to generate code, the generated code becomes predictable, and even identical code snippets may be generated for different developers. This poses potential security risks and makes the software vulnerable to attacks. One aspect of our future work will explore methods to generate diverse code and investigate the extent to which their diversities and quality can be increased with Optimization AI.

Building on prior work [5], a Code Repair AI can be easily transformed into an Optimization AI based on the number of test cases that pass and fail. Optimization AI ensures the same program behavior for two software versions, contributing to Software Diversification. Subsequently, all these diverse software versions are deployed in production. In such a scenario, if one version is attacked, other versions remain operational, ensuring the successful running of the software and avoiding a complete software crash.

REFERENCES

[1] He Ye, Matias Martinez, Thomas Durieux, and Martin Monperrus. A Comprehensive Study of Automatic Program Repair on the QuixBugs Benchmark. Journal of Systems and Software (JSS), Volume 171, 2021.
[2] He Ye, Zimin Chen, and Claire Le Goues. PreciseBugCollector: Extensible, Executable and Precise Bug-fix Collection. Accepted to the 38th IEEE/ACM International Conference on Automated Software Engineering (ASE) Industry Challenge Competition, pages 1899-1910, 2023.
[3] He Ye, Matias Martinez, and Martin Monperrus. Automated Patch Assessment for Program Repair at Scale. Empirical Software Engineering (EmSE), Volume 26, No. 2, 38 pages, 2021.
[4] He Ye, Jian Gu, Matias Martinez, Thomas Durieux, and Martin Monperrus. Automated Classification of Overfitting Patches With Statically Extracted Code Features. In IEEE Transactions on Software Engineering (TSE), vol. 48, no. 8, pages 2920-2938. 2022.
[5] He Ye, Matias Martinez, and Martin Monperrus. Neural Program Repair with Execution-based Backpropagation. In Proceedings of the 44th International Conference on Software Engineering (ICSE), pages 1506–1518, 2022.
[6] He Ye, Matias Martinez, Xiapu Luo, Tao Zhang, and Martin Monperrus. SelfAPR: Self-supervised Program Repair with Test Execution Diagnostics. In Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering (ASE). Article 92, pages 1–13, 2022.
[7] He Ye and Martin Monperrus. ITER: Iterative Neural Repair for Multi-Location Patches. Accepted to the 46th International Conference on Software Engineering (ICSE), 2024.