

Can AI Fix Buggy Code? Exploring the Use of Large Language Models in Automated Program Repair

Lan Zhang, *Northern Arizona University, Flagstaff, AZ, 86005, USA*

Anoop Singhal, *National Institute of Standards and Technology, Gaithersburg, MD, 20899, USA*

Qingtian Zou, *University of Texas Southwestern Medical Center, Dallas, TX, USA*

Xiaoyan Sun, *Worcester Polytechnic Institute, Worcester, MA, 01609, USA*

Peng Liu, *Pennsylvania State University, State College, PA, 16803, USA*

Abstract: LLMs are becoming increasingly used to help programmers fix buggy code due to their remarkable capabilities. This article reviews the current human-LLM collaboration approach to bug fixing and points out the research directions towards (the development of) autonomous program repair AI agents.

INTRODUCTION

The field of software engineering has witnessed a paradigm shift with the advent of large language models (LLMs). These sophisticated AI systems have demonstrated remarkable versatility across various software development tasks, including code generation, bug detection, and code review [1, 2, 3]. The potential of LLMs to revolutionize software development practices has sparked broad interest within both academic and industry circles, prompting a surge of research into their capabilities and limitations.

A recent breakthrough in this domain came with the introduction of Devin, an LLM-powered AI system capable of autonomously completing 13.8% of real-world coding tasks [4]. These tasks encompass a range of complex operations, from diagnosing and fixing bugs to conducting comprehensive code reviews. However, the relatively modest success rate of 13.8% in real-world scenarios raises a critical question that forms the core of our investigation: Are we truly prepared to leverage LLMs for repairing buggy complex programs? This question is not merely academic but has far-reaching implications for the future of software development and maintenance practices.

To address this fundamental quest, our study focuses on two modes of LLM-supported program repair:

Human-LLM Collaboration: This approach examines the synergistic relationship between human software

engineers and LLMs in the bug repair process [5]. It encompasses both interactive, dialogue-based methodologies and more integrated solutions such as real-time code completion and suggestion systems.

Autonomous AI Agent Repair: This mode investigates the potential for LLMs to independently identify and rectify bugs without direct human intervention, representing a more ambitious vision of automated program repair.

By examining the efficacy of LLMs across diverse programming contexts, e.g., C/C++, Java, Python, we aim to provide a nuanced understanding of their current capabilities and limitations in addressing complex software bugs. Our findings reveal a nuanced landscape of LLM-supported program repair. For the Human-LLM Collaboration mode, we observed that results could be significantly improved when humans provide additional contextual knowledge. This includes information about variable contexts, relevant data structures, related functions, and even the underlying logic of the code. This synergy between human expertise and LLM capabilities shows promise for enhancing bug repair processes in complex software systems. In contrast, the Autonomous AI Agent Repair mode presents a more challenging frontier. Our research indicates that we are still far from achieving reliable automatic code repair using LLMs alone. The complexity of real-world software systems, coupled with the nuanced understanding required for effective bug repair, continues to pose significant challenges for fully autonomous LLM-based solutions.

Human-LLM Collaboration

GitHub Copilot's ROBIN system represents a significant advancement in human-LLM collaboration for debugging [6]. It uses multiple AI agents to analyze code context, exception information, and user queries, guiding developers through systematic debugging steps. ROBIN leverages LLMs as reasoning engines to pro-

TABLE 1: LLM-based Program Repair Across Languages and Methodologies

	Dataset Type	Methodologies	Paper
C/C++	Synthetic programs	Human supported dialogue	[7, 8]
	Real-world projects	Human supported dialogue	[2, 6, 9, 8]
Java	Synthetic programs	Human and static tools supported dialogue	[10, 11, 12]
	Real-world projects	Human supported dialogue	[9]
Python	Synthetic programs	Human and static tools supported dialogue	[13, 14, 12]
	Real-world projects	Human and static tools supported dialogue	[15, 16]

vide interactive and collaborative debugging assistance through a chat-based interface. It analyzes exception information, code context, and user queries, guiding developers through a series of steps to explore potential hypotheses, gather more information, and utilize IDE debugging tools to fix issues. This industrial work demonstrates the potential for more effective collaboration between developers and AI in software debugging tasks.

To understand the current state and potential of Human-LLM collaboration in program repair, we conducted a comprehensive review of existing research across multiple programming languages and methodologies. Table 1 summarizes our findings, categorizing studies based on programming language (C/C++, Java, Python), dataset type (synthetic programs and real-world projects), and methodology.

Our analysis reveals a clear trend across all three programming languages: the performance of LLM-assisted repair techniques tends to decrease as the complexity of the dataset increases [2, 7, 9, 15, 8]. For example, Yang et al. [7] and Pearce et al. [8] investigated human-supported dialogue approaches with synthetic programs, achieving remarkably high success rates - up to 100% in some cases. However, when this methodology was extended to real-world projects by Lan Zhang et al. [2], Kulsum et al. [9], and again by Pearce et al. [8] the performance dropped dramatically to less than 20%.

EXAMPLE 1: Syntax difference in Java and C++

Java:

```
String str = "Hello";
List<Integer> numbers = new
ArrayList<>();
System.out.println(str.length());
```

C++:

```
std::string str = "Hello";
int* numbers = new int(5);
std::cout << (*numbers) << std::endl;
delete numbers;
```

While the general trends are consistent across

C/C++, Java, and Python, some language-specific nuances emerged. For instance, as depicted in Example 1, Java manages references to objects without explicit pointers, while C++ allows direct memory manipulation through pointers ('str'). Moreover, Java employs automatic memory management through garbage collection, where 'numbers' is automatically deallocated when it's no longer referenced or goes out of scope. In C++, we must manually allocate memory with 'new' and then explicitly deallocate it with 'delete' to prevent memory leaks. Similar to Java, Python uses automatic memory management. Its dynamic typing and high-level abstractions can simplify certain programming tasks, potentially making some types of repairs more straightforward. For example, Python shows the highest success rate at 38.80% [16], while C/C++ lags behind at 16.5% [2]. The lower performance in C/C++ can be partially attributed to the complexity introduced by manual memory management and pointer manipulation.

Our analysis of the Human-LLM collaboration in program repair leads to one key conclusion: human expertise continues to play a critical role in the bug repair process. Results improve substantially when humans provide additional contextual knowledge [5, 2]:

Context of Variables: Understanding the context of variables is crucial for LLMs in program repair for several reasons. The *scope* of a variable, whether it's global, local, or class-level, determines where it can be accessed and modified. LLMs need to understand this to avoid introducing bugs by incorrectly accessing or modifying variables. Knowing the *range of possible values* a variable can take helps in identifying potential edge cases or unexpected inputs that could lead to vulnerabilities. Understanding how a variable is *typically used* within the code, such as a loop counter, a flag, or to store intermediate results, helps LLMs generate more appropriate and context-aware fixes. Tracking how the value of a *variable changes* throughout the program's execution is essential for identifying the root cause of bugs and proposing effective solutions. In dynamically typed languages, inferring the *type of a variable* from its usage context is crucial for generating type-safe patches.

External Elements: Knowledge of external functions, data structures, and variables is vital for LLMs in program repair. LLMs need to understand the *correct usage of external APIs*, including function signatures, return values, and potential side effects. For languages with manual *memory management*, understanding how external functions allocate and deallocate memory is crucial for preventing memory leaks and buffer overflows. Knowledge of how *external functions* report

errors, such as through return codes or exceptions, is necessary for implementing proper error checking and handling in patches. Understanding whether external functions are *thread-safe* is important when generating patches for multi-threaded applications.

Logic of the Vulnerable Code: Comprehending the logic of vulnerable code is essential for effective program repair. Understanding *what the code is supposed to do* is crucial for ensuring that patches fix the vulnerability without breaking the intended functionality. Analyzing the *control flow* of the vulnerable code helps in identifying potential logical errors or improper handling of edge cases. Understanding how different parts of the *code depend on each other's outputs* helps in ensuring that patches maintain the correct data flow. Grasping the *underlying algorithms* used in the code is necessary for proposing optimizations or alternative implementations that might resolve the vulnerability. Identifying *implicit security assumptions* in the code, such as trust boundaries and input validation, is important for generating patches that enhance the overall security posture.

In case study 1, providing comprehensive information about the vulnerable code's logic and the relationship between the variables $p->tokenpos$ and $p->tokenlen$ greatly helps LLMs generate effective patches. This additional knowledge should involve explaining the fundamental logic behind the vulnerability, emphasizing the potential disparity between $p->tokenpos$ and $p->tokenlen$, and clarifying the proper boundary conditions. Specifically, elucidating that $p->tokenbuf$ is capable of holding $p->tokenlen$ elements, making the maximum index $p->tokenpos$ should be $p->tokenlen - 1$.

CASE STUDY 1: Understanding the range of a variable

I will provide a code that has bug: CWE-119...(shorten to save space)

```
1 static void tokenadd(struct jv_parser *p,
2     char c) {
3     assert(p->tokenpos <= p->tokenlen);
4     /**<start-bug>*/
5     if (p->tokenpos == p->tokenlen) {
6         /**<end-bug>*/
7         p->tokenlen = p->tokenlen * 2 + 256;
8         p->tokenbuf = jv_mem_realloc(p->
9             tokenbuf, p->tokenlen);
10    }
11    assert(p->tokenpos < p->tokenlen);
12    p->tokenbuf[p->tokenpos++] = c;
13 }
```

Reply from ChatGPT (shorten to save space):

The issue here is that there's a potential to write outside of the allocated memory for *tokenbuf* in the last line of code.

```
1 ...
2 /**<start-bug>*/
3 if (p->tokenpos >= p->tokenlen) {
4     p->tokenlen = p->tokenlen*2+256 ;
5     char* newbuf = jv_mem_realloc(p->
6         tokenbuf, p->tokenlen);
7     if (newbuf == NULL) {
8         fprintf(stderr, "Memory allocation
9             failed\n");
10    return;
11 }
12 p->tokenbuf = newbuf;
13 }
14 /**<end-bug>*/
15 assert (p->tokenpos<p->tokenlen);
16 ...
```

Additional knowledge: *Off-by-one error in the tokenadd function allows remote attackers to cause a denial of service (crash), which triggers a heap-based buffer overflow. If $p->tokenpos$ is more than or equal to the end of buffer, resize the buffer. The end of buffer should be $p->tokenlen - 1$ not $p->tokenlen$.*

Reply from ChatGPT (shorten to save space):

```
1 ...
2 /**<start-bug>*/
3 if (p->tokenpos>=p->tokenlen-1) {
4     p->tokenlen = p->tokenlen*2+256 ;
5     char* newbuf = jv_mem_realloc(p->
6         tokenbuf, p->tokenlen);
7     if (newbuf == NULL) {
8         fprintf(stderr, "Memory allocation
9             failed\n");
10    return;
11 }
12 p->tokenbuf = newbuf;
13 }
14 /**<end-bug>*/
15 ...
```

AI Agent Automatic Repair

Recent advancements in artificial intelligence have led to the development of increasingly sophisticated coding assistants, such as Codeium [17], Devin [4], Cursor [18], Magic [19], Replit [20], and Cody [21]. Devin AI represents a significant advancement in AI-powered software engineering, demonstrating impressive capabilities in autonomously resolving GitHub issues. This system can build and deploy applications end-to-end, encompassing tasks such as project cloning, exploration of structures related to vulnerable functions, test case updating based on compiler error messages, generation of new test cases through brute force methods, and bug identification and re-

pair. Devin's ability to resolve 13.8% of issues in the SWE-bench benchmark [22], outperforming GPT-4 by a factor of three, is a notable technical achievement. While tools like GitHub Copilot, Codeium, and Cody primarily focus on code completion and generation, their underlying technologies contribute to the broader field of automatic code repair. These systems leverage large language models trained on vast corpora of code, enabling them to understand code context and suggest fixes for common errors. However, it is crucial to contextualize this success within the broader landscape of software development. While capabilities of automatic AI agents are impressive, the full realization of automatic repair in practical, large-scale software development environments remains a challenging goal that will require further advancements in AI technology and software engineering practices.

Fully Automatic AI Agent are In Early Stage

Despite the promising advancements, automatic code repair using AI face several significant challenges:

Program Comprehension: While Devin has demonstrated enhanced code context understanding compared to previous systems, it still encounters difficulties when faced with complex projects involving numerous interdependent components. For instance, in the case of scikit-learn-11542 [23], Devin identified only two instances with inconsistent default values. However, a comprehensive analysis starting from the RandomForestClassifier would reveal five such instances. This discrepancy highlights the need for more sophisticated algorithms capable of traversing and understanding complex dependency graphs in large-scale software projects.

Verification and Testing: Devin's current approach for verification and testing relies heavily on human-written test cases and brute-force input generation techniques. While this methodology can be effective for simple problems, it falls short when dealing with the complexities of real-world software systems. For example, in scikit-learn-25744 [24], Devin correctly identified that the issue stemmed from the `min_samples_split` parameter and implemented an error message for integer values less than 2. However, it failed to verify the error condition for float values of `min_samples_split`, which should be constrained between 0.0 and 1.0 (exclusive) when representing a percentage. This oversight underscores the potential dangers of incomplete verification, particularly in critical systems where such oversights could lead to severe consequences.

Contextual Understanding: Automatic repair systems must not only fix the immediate bug but also ensure

that the repair aligns with the broader context of the software, including design patterns, coding standards, and project-specific requirements. This level of contextual understanding remains a significant hurdle for current AI systems.

Future Directions in Automatic Code Repair

Recent research in AI-driven program repair has shown promising results, particularly in addressing well-defined programming tasks of limited scope [25, 26, 27]. These works have made significant strides by letting AI agents leverage static and dynamic analysis tools to examine compilation information and code output. This integrated approach guides AI agents in their repair efforts, improving the accuracy and reliability of the generated fixes. However, ensuring the correctness and reliability of AI-generated repairs remains a critical challenge, particularly as we move towards more complex systems. The field of AI-driven program repair continues to evolve, with several promising areas for future research:

Advanced Program Understanding: While recent models have improved in understanding code context, they still struggle with grasping the full scope of a program, including external dependencies, project-specific conventions, and broader architectural considerations. Developing more sophisticated techniques to capture semantic information and programmer intent is crucial for the future of AI-driven program repair. This may involve leveraging advanced natural language processing techniques to better interpret code comments and documentation [28, 29]. Incorporating program dependency analysis could enhance the AI's understanding of the context and potential impact of repairs [2]. Additionally, utilizing machine learning models trained on vast codebases could help in recognizing common patterns and idioms in software design [4, 6].

Rigorous Verification and Testing: One of the most significant challenges is ensuring the correctness of AI-generated patches. While AI models can generate plausible fixes, they may introduce new bugs or fail to fully address the underlying issue. Developing robust verification mechanisms for AI-generated patches remains an open problem. This involves integrating formal verification techniques with AI-generated repairs to provide mathematical guarantees of correctness [25]. Developing specialized testing frameworks that can automatically generate comprehensive test suites for AI-repaired code would help ensure the reliability of the fixes [30]. Additionally, utilizing symbolic execution and model checking techniques would allow for systematic exploration of the state space of repaired

programs [29].

Multi-level Software Reasoning: Enhancing AI models' ability to reason about software at various levels of abstraction is essential for comprehensive program repair. For example, GPT-o1 can reason through complex tasks and solve harder problems than previous models in science, coding, and math [31]. Future work could focus on developing hierarchical models that can simultaneously consider low-level code logic and high-level software system architectures [32]. Exploring reinforcement learning approaches might allow AI agents to learn from the consequences of their repair decisions across different abstraction levels [33, 34, 35]. By improving the AI's ability to reason at multiple levels, we can expect more sophisticated repairs that consider both local code improvements and their global impact on the system.

Explainability and Transparency: From the perspective of AI agents, even as they are expected to work autonomously, the role of human supervision remains crucial, especially in the development and maintenance of critical systems. This underscores the importance of explainability and transparency in AI-driven program repair. Motivated by the need to bridge the gap between AI capabilities and human oversight, future work in this area should focus on several key aspects. SHAP values could quantify the importance of different code features (e.g., specific lines, functions, or dependencies) in the AI's decision to make a particular repair. This would allow human supervisors to understand which parts of the code most influenced the AI's choice of repair strategy [36]. Developing sophisticated attention mechanisms could highlight specific parts of the code that influence the AI's repair decisions, providing insight into the agent's focus and reasoning process [37].

By addressing these key areas, researchers aim to bridge the gap between current capabilities and the vision of fully autonomous AI agents capable of general-purpose program repair. While this goal remains distant, ongoing advancements in these areas continue to push the boundaries of what's possible in AI-driven software development and bug fixing.

Conclusion

Our investigation into the capabilities of LLMs in program repair reveals a nuanced landscape with significant implications for software engineering. In the realm of Human-LLM Collaboration, our findings demonstrate a promising synergy, where human expertise in providing contextual knowledge significantly enhances LLMs' effectiveness in bug repair processes. This collabora-

tive approach shows great potential for improving software development and bug fixing practices, particularly in complex systems. However, the results for Autonomous AI Agent Repair indicate that we are still far from achieving reliable, fully autonomous code repair using LLMs alone. These findings lead us to conclude that while LLMs represent a powerful tool in software engineering, they are not yet ready to replace human expertise in program repair. The most promising path forward appears to be a hybrid approach that leverages the strengths of both human developers and LLMs. As we move forward, it is crucial to focus on enhancing LLMs' contextual understanding, developing more sophisticated Human-LLM interfaces, and improving LLMs' ability to reason about and verify their proposed solutions. By maintaining a balanced perspective and working towards solutions that harmoniously combine human expertise and artificial intelligence, we can continue to advance the field of software development.

Disclaimer

Commercial products are identified in order to adequately specify certain procedures. In no case does such identification imply recommendation or endorsement by the National Institute of Standards and Technology, nor does it imply that the identified products are necessarily the best available for the purpose.

Acknowledgment

Peng Liu was supported by NSF CNS-2019340, NSF ECCS-2140175, and NIST 60NANB22D144. Xiaoyan Sun is supported by NSF DGE2105801.

REFERENCES

1. Wang, Z.; Zhang, L.; Liu, P. ChatGPT for Software Security: Exploring the Strengths and Limitations of ChatGPT in the Security Applications. *arXiv preprint arXiv:2307.12488* **2023**,
2. Zhang, L.; Zou, Q.; Singhal, A.; Sun, X.; Liu, P. Evaluating Large Language Models for Real-World Vulnerability Repair in C/C++ Code. Proceedings of the 10th ACM International Workshop on Security and Privacy Analytics. New York, NY, USA, 2024; p 49–58.
3. Joshi, H.; Sanchez, J. C.; Gulwani, S.; Le, V.; Radiček, I.; Verbruggen, G. Repair Is Nearly Generation: Multilingual Program Repair with LLMs. *Proceedings of the 37th AAAI Conference on Artificial Intelligence, AAAI 2023* **2023**, *37*, 5131–5140.
4. SiebeRozendal Breakthrough in AI agents? <https://forum.effectivealtruism.org/posts/>

- [3brE2Mt6qC72cQvzL/breakthrough-in-ai-agents-on-devin-the-zvi-linkpost](https://arxiv.org/abs/2024.04113), 2024; [Online; accessed 10-Apr-2024].
5. Sellen, A.; Horvitz, E. The Rise of the AI Co-Pilot: Lessons for Design from Aviation and Beyond. 2023; <https://arxiv.org/abs/2311.14713>.
 6. Bajpai, Y.; Chopra, B.; Biyani, P.; Aslan, C.; Gulwani, S.; Coleman, D.; Parnin, C.; Radhakrishna, A.; Soares, G. Let's Fix this Together: Conversational Debugging with GitHub Copilot. IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC). 2024.
 7. Yang, B.; Tian, H.; Pian, W.; Yu, H.; Wang, H.; Klein, J.; Bissyandé, T. F.; Jin, S. CREF: An LLM-based Conversational Software Repair Framework for Programming Tutors. 2024; <https://arxiv.org/abs/2406.13972>.
 8. Pearce, H.; Tan, B.; Ahmad, B.; Karri, R.; Dolan-Gavitt, B. Examining zero-shot vulnerability repair with large language models. 2023 IEEE Symposium on Security and Privacy (SP). 2023; pp 2339–2356.
 9. Kulsum, U.; Zhu, H.; Xu, B.; d'Amorim, M. A Case Study of LLM for Automated Vulnerability Repair: Assessing Impact of Reasoning and Patch Validation Feedback. Proceedings of the 1st ACM International Conference on AI-Powered Software. New York, NY, USA, 2024; p 103–111.
 10. Kang, S.; Yoon, J.; Yoo, S. Large Language Models are Few-shot Testers: Exploring LLM-based General Bug Reproduction. **2022**,
 11. Xia, C. S.; Zhang, L. Keep the Conversation Going: Fixing 162 out of 337 bugs for \$0.42 each using ChatGPT. *arXiv preprint arXiv:2304.00385* **2023**,
 12. Wadhwa, N.; Pradhan, J.; Sonwane, A.; Sahu, S. P.; Natarajan, N.; Kanade, A.; Parthasarathy, S.; Rajamani, S. Frustrated with Code Quality Issues? LLMs can Help! *arXiv preprint arXiv:2309.12938* **2023**,
 13. Lemieux, C.; Priya Inala, J.; Lahiri, S. K.; Sen, S. CODAMOSA: Escaping Coverage Plateaus in Test Generation with Pre-trained Large Language Models. *Icse* 1–13.
 14. Cao, J.; Li, M.; Wen, M.; Cheung, S.-c. *Proceedings of arXiv*; Association for Computing Machinery, 2023; Vol. 1.
 15. Parasaram, N.; Yan, H.; Yang, B.; Flahy, Z.; Qudsi, A.; Ziaber, D.; Barr, E.; Mechtav, S. The Fact Selection Problem in LLM-Based Program Repair. 2024; <https://arxiv.org/abs/2404.05520>.
 16. Jimenez, C. E.; Yang, J.; Wettig, A.; Yao, S.; Pei, K.; Press, O.; Narasimhan, K. R. SWE-bench: Can Language Models Resolve Real-world Github Issues? The Twelfth International Conference on Learning Representations. 2024.
 17. <https://codeium.com/>.
 18. <https://anysphere.inc/>.
 19. <https://magic.dev/>.
 20. <https://replit.com/>.
 21. <https://meetcody.ai/>.
 22. Jimenez, C. E.; Yang, J.; Wettig, A.; Yao, S.; Pei, K.; Press, O.; Narasimhan, K. SWE-bench: Can Language Models Resolve Real-World Github Issues? 2024; <https://arxiv.org/abs/2310.06770>.
 23. https://github.com/CognitionAI/devin-swebench-results/blob/main/output_diffs/fail/scikit-learn__scikit-learn-11542-diff.txt.
 24. https://github.com/CognitionAI/devin-swebench-results/blob/main/output_diffs/fail/scikit-learn__scikit-learn-25744-diff.txt.
 25. Fan, Z.; Gao, X.; Mirchev, M.; Roychoudhury, A.; Tan, S. H. Automated Repair of Programs from Large Language Models. **2022**,
 26. Bouzenia, I.; Devanbu, P.; Pradel, M. RepairAgent: An Autonomous, LLM-Based Agent for Program Repair. 2024; <https://arxiv.org/abs/2403.17134>.
 27. Qian, C.; Li, J.; Dang, Y.; Liu, W.; Wang, Y.; Xie, Z.; Chen, W.; Yang, C.; Zhang, Y.; Liu, Z.; Sun, M. Iterative Experience Refinement of Software-Developing Agents. 2024; <https://arxiv.org/abs/2405.04219>.
 28. Shinyama, Y.; Arahori, Y.; Gondow, K. Analyzing code comments to boost program comprehension. 2018 25th Asia-Pacific Software Engineering Conference (APSEC). 2018; pp 325–334.
 29. Zhou, W.; Zhang, L.; Guan, L.; Liu, P.; Zhang, Y. What Your Firmware Tells You Is Not How You Should Emulate It: A Specification-Guided Approach for Firmware Emulation. Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security. New York, NY, USA, 2022; p 3269–3283.
 30. Oliveira, C.; Aleti, A.; Grunske, L.; Smith-Miles, K. Mapping the effectiveness of automated test suite generation techniques. *IEEE Transactions on Reliability* **2018**, *67*, 771–785.
 31. <https://openai.com/index/introducing-openai-o1-preview/>.
 32. Muccini, H.; Inverardi, P.; Bertolino, A. Using software architecture for code testing. *IEEE Transactions on Software Engineering* **2004**, *30*, 160–171.
 33. Zhang, L.; Liu, P.; Choi, Y.-H.; Chen, P. Semantics-preserving reinforcement learning attack against graph neural networks for malware detection. *IEEE Transactions on Dependable and Secure Comput-*

- ing **2022**, *20*, 1390–1402.
34. Islam, N. T.; Khoury, J.; Seong, A.; Parra, G. D. L. T.; Bou-Harb, E.; Najafirad, P. LLM-Powered Code Vulnerability Repair with Reinforcement Learning and Semantic Reward. *arXiv preprint arXiv:2401.03374* **2024**,
 35. Gupta, R.; Kanade, A.; Shevade, S. Deep reinforcement learning for syntactic error repair in student programs. Proceedings of the AAAI Conference on Artificial Intelligence. 2019; pp 930–937.
 36. Zou, Q.; Zhang, L.; Singhal, A.; Sun, X.; Liu, P. Using Explainable AI for Neural Network Based Network Attack Detection. *IEEE Computer* **2024**, *57*.
 37. Niu, Z.; Zhong, G.; Yu, H. A review on the attention mechanism of deep learning. *Neurocomputing* **2021**, *452*, 48–62.

Lan Zhang Dr. Zhang (lan.zhang@nau.edu) is an Assistant Professor at Northern Arizona University's School of Informatics, Computing, and Cyber Systems (SICCS). She earned a Ph.D. in Information Sciences and Technology (IST) from Pennsylvania State University. Her research lies at the intersection of Artificial Intelligence and cybersecurity, with a focus on developing innovative solutions to address emerging challenges in AI-driven security systems.

Anoop Singhal Dr. Singhal (anoop.singhal@nist.gov) is a Senior Computer Scientist in the Computer Security Division at the National Institute of Standards and Technology (NIST). He received his Ph.D. in Computer Science from Ohio State University. His research interests are in network security, cloud computing security, network forensics, security metrics and data mining systems.

Qingtian Zou Dr. Zou (qingtian.mill.zou@gmail.com) is a Postdoctoral Researcher at the University of Texas, Southwestern Medical Center. He earned his Ph.D. in Information Sciences and Technology from Pennsylvania State University in 2023. His research focuses on deep learning and its diverse applications, particularly in advancing innovative solutions to complex problems.

Xiaoyan (Sherry) Sun Dr. Sun (xsun7@wpi.edu) is an Associate Professor with Department of Computer Science, Worcester Polytechnic Institute. She received her Ph.D. degree in Information Sciences and Technology from the Pennsylvania State University in 2016. Her research interests lie in cybersecurity and digital forensics.

Peng Liu Dr. Liu (pxl20@psu.edu) holds the prestigious title of Raymond G. Tronzo, MD Professor of Cybersecurity and serves as the Director of the Cyber Security Lab at The Pennsylvania State University, located in State College, PA.