Chapter 9
Vulnerabilities Introduced by LLMs Through Code Suggestions

Sebastiano Panichella

Abstract Code suggestions from generative language models like ChatGPT contain vulnerabilities as they often rely on older code and programming practices, over-represented in the older code libraries the LLMs rely on for their coding abilities. Advanced attackers can leverage this by injecting code with known but hard-to-detect vulnerabilities in the training datasets. Mitigation can include user education and engineered safeguards such as LLMs trained for vulnerability detection or rule-based checking of codebases. Analysis of LLMs' code generation capabilities, including formal verification and source training dataset (code-comment pairs) analysis, is necessary for effective vulnerability detection and mitigation.

9.1 Introduction

The landscape of software development has been revolutionized by the emergence of generative language models such as ChatGPT and GitHub Copilot, which offer code recommendations and suggestions to developers. However, while these models provide tremendous convenience and productivity gains, a latent concern exists surrounding the security implications of their outputs. This chapter delves into the relationship between generative language models (LLMs) and the security of the generated code, shedding light on the vulnerabilities that can arise.

Unlike natural language-generating LLMs, where counterfactual text generation ("hallucinations") is a major concern, code-generating LLM output undergoes either compilation or interpretation. As such, code that does not sufficiently adhere to examples in the model's training dataset does not pose as much risk, given that it will most likely fail to execute or lead to an immediately detectable wrong behavior. Because of that, a much bigger risk for code-generating LLMs is the

S. Panichella (✉)
Zurich University of Applied Sciences, Zurich, Switzerland
e-mail: panc@zhaw.ch

A. Kucharavy et al. (eds.), *Large Language Models in Cybersecurity*,
https://doi.org/10.1007/978-3-031-54827-7_9

problematic code in their training dataset. Code-generating LLMs are trained from historical codebases and programming practices, which might be outdated or even include several vulnerabilities. As a result, the code snippets generated by LLMs could inadvertently incorporate these vulnerabilities, posing a potential threat to the security of the resultant software. LLMs tend to favor older code libraries and repositories for learning, leading to an over-representation of deprecated and potentially risky coding paradigms.

The vulnerabilities introduced by LLM-generated code open up opportunities for advanced attackers to exploit the weaknesses in the software. These attackers can strategically inject malicious code leveraging well-concealed vulnerabilities in the training datasets. Detecting and countering these vulnerabilities pose significant challenges due to their elusive nature. Notably, these vulnerabilities might be identified by humans/developers only with considerable effort, making their identification a non-trivial task.

Addressing these security concerns necessitates a multifaceted approach. One avenue for mitigation involves enhancing user education about the potential risks inherent in relying blindly on LLM-generated code. Additionally, the integration of engineered safeguards, such as LLMs specialized in vulnerability detection or rule-based assessments of codebases, can provide an extra layer of protection. However, the complexity of LLMs and the subtlety of vulnerabilities they introduce necessitate a more thorough exploration. An in-depth analysis of LLMs' code generation capabilities is crucial, encompassing methods like formal verification and exhaustive examination of the source training datasets, including code-comment pairs. Such analyses will pave the way for effective vulnerability detection and mitigation strategies.

Looking ahead, the chapter also delineates future research prospects in the realm of LLMs and security. Researchers and practitioners are poised to delve deeper into devising techniques for accurately identifying vulnerabilities in LLM-generated code and methodologies for generating secure code without stifling the models' creative capabilities. Exploring techniques to fine-tune LLMs using security-focused datasets could also yield models more adept at producing secure code snippets.

In summary, this chapter not only exposes the risks and challenges associated with security in the context of LLM-generated code but also sheds some light on the potential opportunities for enhancing software security through vigilant research, innovative techniques, and proactive safeguarding measures. It is a compass for researchers and practitioners navigating the intricate landscape where the promise of LLMs intersects with the imperative of secure software development.

9.2 Relationship Between LLMs and Code Security

The software development state of the practice has undergone a remarkable transformation with the advent of LLMs like ChatGPT and GitHub Copilot.

These cutting-edge LLMs have introduced a revolutionary shift by providing developers with an array of code recommendations and insightful suggestions [1]. This innovative advancement has effectively transformed the way software is created and refined. Through their sophisticated capabilities, ChatGPT and GitHub Copilot have emerged as pivotal tools that empower developers with enhanced efficiency and creativity, ushering in a new era of collaborative and accelerated software development processes, including coding [1, 2] and code documentation activities [3].

9.2.1 Vulnerabilities and Risks Introduced by LLM-Generated Code

An important and significant risk associated with the utilization of such a model arises from the fundamental premise that they are trained using historical codebases and programming practices [4, 5]. This aspect brings to light a multifaceted concern, wherein the historical context might potentially render the acquired knowledge outdated or obsolete. It could inadvertently encompass numerous vulnerabilities and security loopholes within its framework [6, 7].

The crux of this risk lies in the inherent nature of LLMs, which learn from the vast repository of programming examples that have been amassed over time. While this repository undoubtedly offers a treasure trove of insights into the evolution of coding paradigms, it also implies that LLMs are exposed to a wide array of programming techniques that have potentially been rendered obsolete due to advancements in technology, shifts in best practices, or the identification of security flaws [7]. Furthermore, the historical codebases upon which LLMs are trained might inadvertently harbor vulnerabilities that were unknown or less prioritized in the past but have since emerged as critical points of concern in contemporary software development [8]. If ingrained within the model's learned patterns, these vulnerabilities could propagate into the code it generates, leading to inadvertent security breaches or susceptibility to cyberattacks. LLMs tend to favor older code libraries and repositories for learning, leading to an over-representation of deprecated and potentially risky coding paradigms.

In the rapidly evolving landscape of technology and cybersecurity, relying solely on historical programming knowledge to shape the capabilities of LLMs can be likened to building upon a risky and antiquated foundation. As software development methodologies adapt to new security standards, coding practices, and emerging paradigms, the risk of generating code that adheres to outdated or insecure practices becomes increasingly possible [6, 8].

An additional critical factor that warrants careful consideration is the inherent vulnerability of LLMs to adversarial attacks. These attacks, which exploit the intricate nuances of the model's behavior, raise significant concerns regarding

the model's robustness and reliability in real-world applications [9, 10].[1] The susceptibility of LLMs to such attacks underscores the necessity for rigorous testing [11–16] and fortification of these models to ensure their resilience in the face of diverse adversarial strategies. Adversarial attacks targeting LLMs involve subtly manipulating input data that may seem inconsequential to human observers but can lead to significant distortions in the model's outputs. This vulnerability stems from the intricate nature of language understanding and generation, where slight perturbations can cause LLMs to produce misleading or erroneous results. Consequently, this susceptibility poses a multifaceted challenge encompassing not only the theoretical understanding of these vulnerabilities but also the practical implementation of effective defense mechanisms. The intricate interplay between LLMs and adversarial attacks introduces a multifaceted challenge that demands concerted efforts from researchers, practitioners, and policymakers alike [10]. By delving deeper into the vulnerabilities inherent to these models and collaborating across disciplines, I can pave the way for the development of LLMs that not only excel in their linguistic capabilities but also stand resilient against the ever-evolving landscape of adversarial threats [17, 18].

In essence, while LLMs present remarkable potential in enhancing developer productivity and catalyzing innovation, a judicious approach to their usage must be adopted. This involves acknowledging the limitations inherent in training these models on historical data and proactively addressing the challenges posed by outdated practices and vulnerabilities. Through a concerted and vigilant effort, the benefits of LLMs can be harnessed while minimizing the inherent risks, ultimately leading to a more secure and robust software development landscape. In the next section, I discuss more in detail potential mitigation strategies for such problems.

9.3 Mitigating Security Concerns With LLM-Generated Code

Secure LLM-Based Programming with Static, Code, and Change Analysis Previous work discusses the challenges and benefits of using static analysis tools to ensure secure programming practices, highlighting the importance of tools and techniques in identifying vulnerabilities in code, which aligns with the challenges of detecting vulnerabilities in LLM-generated code [19, 20]. Researchers and practitioners are called into devising static analysis-based techniques for accurately identifying vulnerabilities in LLM-generated code, as well as methodologies for generating secure code without stifling the models' creative capabilities. Complementary, exploring techniques to fine-tune LLMs using security-focused datasets could yield models that are more adept at producing secure code snippets [21, 22].

[1] https://github.com/llm-attacks/llm-attacks.

In a closely related direction, recent studies proposed code-based or static metadata-based vulnerability detection or prediction techniques in the context of code written by developers of open source and mobile applications [23, 24], providing an overview of techniques that can be applied to assessing LLM-generated code. Here, the challenge is to study and investigate how much it is possible to generalize them to LLMs' generated software. In particular, the concept of *vulnerability-proneness* [24] of software applications created on top of LLMs could contribute to the understanding of vulnerabilities and the potential risks introduced by its code, which is relevant to the security concerns discussed in the chapter. This notion can be combined with more exhaustive and expensive techniques from the state-of-the-art vulnerability detection [23, 25].

Another relevant direction for mitigating security issues with LLM-generated code concerns the adaptation of change analysis [26–29] and code analysis [30–34] strategies, to enact monitoring and testing automation for LLM generated-code behavior [35–38]. Specifically, while such previous research was very timely and relevant for software and cyber-physical systems, such approaches are intrinsically insufficient to deal with the evolving, dynamic, and safety-critical nature of code generated and modified with the support of LLMs. Once security concerns with such adapted techniques, researchers could explore the opportunity to investigate *code clone* techniques [39, 40], which typically target the identification (or monitoring) of code clones that involve subtle changes or variations of existing similar code, and that presents vulnerabilities/security risks or issues.

Automated Code Review for LLMs The potential risks associated with using outdated or vulnerable codebases for training contribute to the need for *Modern Code Review* (MCR) practices to address these issues [29, 41, 42].[2,3] MCR is a key process in software development aimed at inspecting (code inspection done typically by developers) for identifying and rectifying programming and vulnerability issues, which is relevant to the topic of identifying vulnerabilities and code-related issues introduced in LLM-generated code. In this, context, recent research proposed approaches to automate the code review process [31, 43–52], as well as proposed methods to evaluate them [53]. Hence, similarly to previous empirical research, this chapter suggests the investigation of MCR practices that are suited for LLM-generated code. Compared to the previous studies, researchers in the field are required to manually and/or automatically analyze MCR changes [29, 41, 42].

Monitoring of Adversarial Attacks and Formal verification of LLMs As the application domains of LLMs continue to expand, ranging from automated content generation to personalized assistance, it is crucial to establish robust evaluation benchmarks that account for their susceptibility to adversarial attacks and general

[2] https://medium.com/@andrew_johnson_4/the-role-of-large-language-models-in-code-review-2b74598249ab.

[3] https://paperswithcode.com/paper/lever-learning-to-verify-language-to-code/review/?hl=100085.

security risks. These benchmarks should encompass a wide array of potential attack vectors, spanning from syntactic manipulations to more sophisticated semantic distortions. By subjecting LLMs to a battery of rigorous tests, I can benchmark their performance under different adversarial scenarios and iteratively refine their architectures to enhance their defense mechanisms. To mitigate LLMs-related risks, it becomes imperative to implement comprehensive validation and verification processes that scrutinize the code generated by LLMs for adherence to current security standards and best practices. This entails not only ensuring the functional correctness of the code but also conducting thorough security audits to identify and rectify potential vulnerabilities that might have been inadvertently woven into the resulting generated code.

To address the risk of adversarial attacks comprehensively, fostering collaboration between the research community and industry stakeholders is imperative. By coordinating the research and expertise from diverse fields, including machine learning, cybersecurity, linguistics, and cognitive science, I can devise innovative strategies to enhance the resilience of LLMs [17, 18]. These efforts might involve the development of novel training or repairing techniques [54] that can expose models to a broader spectrum of adversarial examples during their learning process, thereby augmenting their ability to discern subtle deviations and generate accurate responses. Complementary, addressing these security concerns necessitates a multifaceted approach, encompassing methods such as formal verification and exhaustive examination of the source training datasets [55–57], including code-comment analysis, evolution and consistency [58].

Explainability and Testing in the Era of LLMs An additional crucial challenge that arises pertains to the intricate realm of explainability, particularly when utilizing empirical software engineering methodologies [59]. Within the expansive landscape of Language Model technologies, like LLMs, the task of elucidating their decision-making processes becomes a paramount concern. The endeavor to decipher and articulate the rationales behind the outcomes generated by these models becomes increasingly intricate, requiring sophisticated techniques that can fathom the complex inner workings of these advanced systems.

Simultaneously, an equally significant facet that necessitates thorough consideration is the rigorous testing of LLMs [11, 60], often referred to as the *oracle problem* [60]. This predicament underscores the difficulty of establishing a reliable and comprehensive benchmark or reference for evaluating the accuracy and effectiveness of these models' outputs. Given language's dynamic and ever-evolving nature, the challenge of devising a definitive gold standard against which these models can be measured presents an ongoing obstacle. In essence, the intersection of these challenges underscores the multidimensional nature of working with LLMs within the context of software engineering [11, 60]. Addressing the issues of explainability and testing entails delving into the intricacies of these models, reconciling their outputs with human logic and language nuances, and crafting methodologies that can reliably gauge their performance in a field where definitive truths are often elusive.

9.4 Conclusion and The Path Forward

In conclusion, this chapter has delved deep into the intricate relationship between LLMs and the security of the code they produce. The evolution of software development, catalyzed by ChatGPT and GitHub Copilot, brings immense advantages in terms of efficiency and productivity. However, the security implications inherent in the outputs of these LLMs cannot be ignored.

As highlighted throughout this chapter, the vulnerabilities that can seep into LLM-generated code present significant challenges for software security. Integrating outdated programming practices and potential vulnerabilities from historical codebases raises concerns about the robustness of the resulting software. The chapter underscores the inherent risks of relying blindly on LLM-generated code, emphasizing the need for heightened user education and awareness.

The solutions proposed here are multifaceted. Engineered safeguards, tailored LLMs for vulnerability detection, and rule-based assessments of codebases can offer an extra layer of protection against exploitable weaknesses. Nevertheless, the complexity of LLMs and the subtle nature of vulnerabilities necessitate a more profound investigation. This entails enhanced vulnerability detection and a comprehensive exploration of techniques to generate secure code without stifling the creative capabilities of these models.

In essence, this chapter acts as a guiding light for those navigating the dynamic landscape where LLMs intersect with the imperatives of software security. It emphasizes the importance of proactive research and safeguarding measures, all of which are essential to harnessing the potential of LLMs while mitigating the inherent risks. By taking these insights to heart and advancing the proposed research directions, the field stands to elevate software security to new heights in an era defined by transformative linguistic technologies.

References

1. Gustavo Sandoval et al. Lost at c: A user study on the security implications of large language model code assistants, 2023.
2. Alec Radford et al. Language models are unsupervised multitask learners. *OpenAI blog*, 1(8):9, 2019.
3. Toufique Ahmed and Premkumar T. Devanbu. Few-shot training llms for project-specific code-summarization. In *37th IEEE/ACM International Conference on Automated Software Engineering, ASE 2022, Rochester, MI, USA, October 10–14, 2022*, pages 177:1–177:5. ACM, 2022.
4. Sameera Horawalavithana et al. Mentions of security vulnerabilities on reddit, twitter and github. In Payam M. Barnaghi, Georg Gottlob, Yannis Manolopoulos, Theodoros Tzouramanis, and Athena Vakali, editors, *WI*, pages 200–207. ACM, 2019.
5. David Glukhov et al. Llm censorship: A machine learning challenge or a computer security problem?, 2023.

6. Ahmed Zerouali, Tom Mens, Alexandre Decan, and Coen De Roover. On the impact of security vulnerabilities in the npm and rubygems dependency networks. *Empir. Softw. Eng.*, 27(5):107, 2022.
7. Muhammad Shumail Naveed Abdul Malik. Analysis of code vulnerabilities in repositories of github and rosettacode: A comparative study. *International Journal of Innovations in Science & Technology*, 4(2):499–511, Jun. 2022.
8. Mansooreh Zahedi, Muhammad Ali Babar, and Christoph Treude. An empirical study of security issues posted in open source projects. In Tung Bui, editor, *HICSS*, pages 1–10. ScholarSpace / AIS Electronic Library (AISeL), 2018.
9. Andy Zou, Zifan Wang, J. Zico Kolter, and Matt Fredrikson. Universal and transferable adversarial attacks on aligned language models, 2023.
10. Erik Derner and Kristina Batistič. Beyond the safeguards: Exploring the security risks of chatgpt, 2023.
11. Junjie Wang et al. Software testing with large language model: Survey, landscape, and vision, 2023.
12. Sebastiano Panichella, Alessio Gambi, Fiorella Zampetti, and Vincenzo Riccio. Sbst tool competition 2021. In *2021 IEEE/ACM 14th International Workshop on Search-Based Software Testing (SBST)*, pages 20–27, 2021.
13. Christian Birchler et al. Machine learning-based test selection for simulation-based testing of self-driving cars software. *Empir. Softw. Eng.*, 28(3):71, 2023.
14. Sajad Khatiri et al. Machine learning-based test selection for simulation-based testing of self-driving cars software. *CoRR*, abs/2111.04666, 2021.
15. Andrea Stocco and Paolo Tonella. Confidence-driven weighted retraining for predicting safety-critical failures in autonomous driving systems. *J. Softw. Evol. Process.*, 34(10), 2022.
16. Sajad Khatiri, Sebastiano Panichella, and Paolo Tonella. Simulation-based test case generation for unmanned aerial vehicles in the neighborhood of real flights. In *International Conference on Software Testing, Verification and Validation*, 2023.
17. Jiongxiao Wang et al. Adversarial demonstration attacks on large language models, 05 2023.
18. Alexander Wan, Eric Wallace, Sheng Shen, and Dan Klein. Poisoning language models during instruction tuning, 2023.
19. R. E. Strom and S. Yemini. Typestate: A programming language concept for enhancing software reliability. *IEEE Transactions on Software Engineering*, SE-12(1):157–171, January 1986.
20. Henning Perl et al. Vccfinder: Finding potential vulnerabilities in open-source projects to assist code audits. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, page 426–437, New York, NY, USA, 2015. Association for Computing Machinery.
21. Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation, 2023.
22. Norbert Tihanyi et al. The formai dataset: Generative ai in software security through the lens of formal verification, 2023.
23. Nima Shiri Harzevili et al. A Survey on Automated Software Vulnerability Detection Using Machine Learning and Deep Learning. *arXiv e-prints*, page arXiv:2306.11673, 05 2023.
24. Andrea Di Sorbo and Sebastiano Panichella. Exposed! A case study on the vulnerability-proneness of google play apps. *Empir. Softw. Eng.*, 26(4):78, 2021.
25. Yi Wu, Nan Jiang, Hung Viet Pham, Thibaud Lutellier, Jordan Davis, Lin Tan, Petr Babkin, and Sameena Shah. How effective are neural networks for fixing security vulnerabilities. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*. ACM, jul 2023.
26. Sebastiano Panichella et al. How developers' collaborations identified from different sources tell us about code changes. In *30th IEEE International Conference on Software Maintenance and Evolution, Victoria, BC, Canada, September 29 - October 3, 2014*, pages 251–260, 2014.

27. Y. Zhou et al. User review-based change file localization for mobile applications. *IEEE Transactions on Software Engineering*, pages 1–1, 2020.
28. Sebastiano Panichella. Summarization techniques for code, change, testing, and user feedback (invited paper). In *2018 IEEE Workshop on Validation, Analysis and Evolution of Software Tests, VST@SANER 2018, Campobasso, Italy, March 20, 2018*, pages 1–5, 2018.
29. Sebastiano Panichella and Nik Zaugg. An empirical investigation of relevant changes and automation needs in modern code review. *Empir. Softw. Eng.*, 25(6):4833–4872, 2020.
30. Sebastiano Panichella, Gerardo Canfora, Massimiliano Di Penta, and Rocco Oliveto. How the evolution of emerging collaborations relates to code changes: an empirical study. In *22nd International Conference on Program Comprehension, ICPC 2014, Hyderabad, India, June 2–3, 2014*, pages 177–188, 2014.
31. Sebastiano Panichella, Venera Arnaoudova, Massimiliano Di Penta, and Giuliano Antoniol. Would static analysis tools help developers with code reviews? In *22nd IEEE International Conference on Software Analysis, Evolution, and Reengineering, SANER 2015, Montreal, QC, Canada, March 2–6, 2015*, pages 161–170, 2015.
32. Carol V. Alexandru, Sebastiano Panichella, Sebastian Proksch, and Harald C. Gall. Redundancy-free analysis of multi-revision software artifacts. *Empirical Software Engineering*, 24(1):332–380, 2019.
33. Carol V. Alexandru, Sebastiano Panichella, and Harald C. Gall. Replicating parser behavior using neural machine translation. In *Proceedings of the 25th International Conference on Program Comprehension, ICPC 2017, Buenos Aires, Argentina, May 22–23, 2017*, pages 316–319, 2017.
34. Carmine Vassallo et al. How developers engage with static analysis tools in different contexts. *Empirical Software Engineering*, 2019.
35. Andrea Di et al. Sorbo. Automated identification and qualitative characterization of safety concerns reported in UAV software platforms. *ACM Trans. Softw. Eng. Methodol.*, 2022.
36. Gunel Jahangirova, Andrea Stocco, and Paolo Tonella. Simulation-based test case generation for unmanned aerial vehicles in the neighborhood of real flights. In *International Conference on Software Testing, Verification and Validation, ICST*, 2023.
37. Fiorella Zampetti et al. Continuous integration and delivery practices for cyber-physical systems: An interview-based study. *ACM Trans. Softw. Eng. Methodol.*, 2022.
38. Fiorella Zampetti, Ritu Kapur, Massimiliano Di Penta, and Sebastiano Panichella. An empirical characterization of software bugs in open-source cyber–physical systems. *Journal of Systems and Software*, 192:111425, 2022.
39. Martin White, Michele Tufano, Christopher Vendome, and Denys Poshyvanyk. Deep learning code fragments for code clone detection. In David Lo, Sven Apel, and Sarfraz Khurshid, editors, *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, ASE 2016, Singapore, September 3–7, 2016*, pages 87–98. ACM, 2016.
40. Liuqing Li et al. Cclearner: A deep learning-based clone detection approach. In *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 249–260, 2017.
41. Daoguang Zan et al. Large language models meet NL2Code: A survey. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 7443–7464, Toronto, Canada, 07 2023. Association for Computational Linguistics.
42. Ying Yin, Yuhai Zhao, Yiming Sun, and Chen Chen. Automatic code review by learning the structure information of code graph. *Sensors*, 23(05):2551, 2023.
43. Mike Barnett, Christian Bird, João Brunet, and Shuvendu K. Lahiri. Helping developers help themselves: Automatic decomposition of code review changesets. In *37th IEEE/ACM International Conference on Software Engineering, ICSE 2015, Florence, Italy, May 16–24, 2015, Volume 1*, pages 134–144, 2015.
44. Tianyi Zhang, Myoungkyu Song, Joseph Pinedo, and Miryung Kim. Interactive code review for systematic changes. In *37th IEEE/ACM International Conference on Software Engineering, ICSE 2015, Florence, Italy, May 16–24, 2015, Volume 1*, pages 111–122, 2015.

45. Vipin Balachandran. Reducing human effort and improving quality in peer code reviews using automatic static analysis and reviewer recommendation. In *35th International Conference on Software Engineering, ICSE '13, San Francisco, CA, USA, May 18–26, 2013*, pages 931–940, 2013.
46. Motahareh Bahrami Zanjani, Huzefa H. Kagdi, and Christian Bird. Automatically recommending peer reviewers in modern code review. *IEEE Trans. Software Eng.*, 42(6):530–543, 2016.
47. Ali Ouni, Raula Gaikovina Kula, and Katsuro Inoue. Search-based peer reviewers recommendation in modern code review. In *2016 IEEE International Conference on Software Maintenance and Evolution, ICSME 2016, Raleigh, NC, USA, October 2–7, 2016*, pages 367–377, 2016.
48. Christoph Hannebauer, Michael Patalas, Sebastian Stünkel, and Volker Gruhn. Automatically recommending code reviewers based on their expertise: an empirical comparison. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, ASE 2016, Singapore, September 3–7, 2016*, pages 99–110, 2016.
49. Patanamon Thongtanunam et al. Who should review my code? A file location-based code-reviewer recommendation approach for modern code review. In *22nd IEEE International Conference on Software Analysis, Evolution, and Reengineering, SANER 2015, Montreal, QC, Canada, March 2–6, 2015*, pages 141–150, 2015.
50. Carmine Vassallo et al. Context is king: The developer perspective on the usage of static analysis tools. In *25th International Conference on Software Analysis, Evolution and Reengineering, SANER 2018, Campobasso, Italy, March 20–23, 2018*, pages 38–49, 2018.
51. Robert Chatley and Lawrence Jones. Diggit: Automated code review via software repository mining. In *25th International Conference on Software Analysis, Evolution and Reengineering, SANER 2018, Campobasso, Italy, March 20–23, 2018*, pages 567–571, 2018.
52. Shu-Ting Shi et al. Automatic code review by learning the revision of source code. In *The Thirty-Third AAAI Conference on Artificial Intelligence, AAAI 2019, The Thirty-First Innovative Applications of Artificial Intelligence Conference, IAAI 2019, The Ninth AAAI Symposium on Educational Advances in Artificial Intelligence, EAAI 2019, Honolulu, Hawaii, USA, January 27 - February 1, 2019*, pages 4910–4917. AAAI Press, 2019.
53. Martin Höst and Conny Johansson. Evaluation of code review methods through interviews and experimentation. *Journal of Systems and Software*, 52(2–3):113–120, 2000.
54. H. Pearce, B. Tan, B. Ahmad, R. Karri, and B. Dolan-Gavitt. Examining zero-shot vulnerability repair with large language models. In *2023 IEEE Symposium on Security and Privacy (SP)*, pages 2339–2356, Los Alamitos, CA, USA, may 2023. IEEE Computer Society.
55. Yiannis Charalambous et al. A new era in software security: Towards self-healing software via large language models and formal verification, 2023.
56. Susmit Jha et al. Dehallucinating large language models using formal methods guided iterative prompting. In *2023 IEEE International Conference on Assured Autonomy (ICAA)*, pages 149–152, 2023.
57. Yuhuai Wu, Albert Qiaochu Jiang, Wenda Li, Markus Rabe, Charles Staats, Mateja Jamnik, and Christian Szegedy. Autoformalization with large language models. In S. Koyejo, S. Mohamed, A. Agarwal, D. Belgrave, K. Cho, and A. Oh, editors, *Advances in Neural Information Processing Systems*, volume 35, pages 32353–32368. Curran Associates, Inc., 2022.
58. Pooja Rani et al. A decade of code comment quality assessment: A systematic literature review. *J. Syst. Softw.*, 195:111515, 2023.
59. Yunfan Gao et al. Chat-rec: Towards interactive and explainable llms-augmented recommender system, 2023.
60. Gunel Jahangirova. Oracle problem in software testing. In Tevfik Bultan and Koushik Sen, editors, *ISSTA*, pages 444–447. ACM, 2017.