Lineage-Aware Security for AI-Assisted Software Development

Rafael Araújo Rodrigues
Pontifical Catholic University of Rio Grande do Sul
Porto Alegre, Brazil
rafael.araujo11@edu.pucrs.br

Avelino Zorzo
Pontifical Catholic University of Rio Grande do Sul
Porto Alegre, Brazil
avelino.zorzo@pucrs.br

Elder de Macedo Rodrigues
Universidade Federal do Pampa (UNIPAMPA)
Alegrete, Brazil
elderrodrigues@unipampa.edu.br

Lauren Silva Rolan Sampaio
HP Inc. Brazil
Porto Alegre, Brazil
lauren.silva.rolan.sampaio1@hp.com

Abstract

AI-assisted software development is becoming part of everyday software engineering, from code generation and review to testing, refactoring, and pull request creation. This shift challenges security practices focused mainly on final code artifacts. In AI-assisted workflows, insecure code may emerge from prompts, repository history, retrieved context, developer decisions, agent actions, dependencies, configurations, and build artifacts. This position paper argues for lineage-aware security as a complement to DevSecOps. We propose a layered perspective connecting historical, technical, socio-technical, AI-contextual, and agent/toolchain/supply-chain evidence. Rather than replacing SAST, DAST, SCA, or LLM-based detection, this perspective integrates their evidence into an auditable explanation of how risky code was produced, accepted, and propagated. Such lineage evidence supports traceability, auditing, reproducibility, and prevention in secure AI-assisted software development.

Keywords

AI-assisted software development, Lineage-aware security, DevSecOps, Vulnerability provenance

1 Introduction

AI-assisted programming tools are now embedded in everyday development, from code completion and test generation to refactoring, documentation, and bug fixing [14, 22, 29]. Research has also shown that these tools can produce insecure code, be difficult to validate, and influence developers in ways not fully visible in repository traces [1, 6, 17, 26]. This creates a mismatch between AI-assisted development and dominant security analysis practices.

Much software security work still focuses on the code artifact after the fact. Static and dynamic analysis, taint analysis, graph-based vulnerability mining, and LLM-based classifiers provide important capabilities [15, 21, 30, 33], but usually begin from a code snapshot, diff, or execution trace. They often answer where a problem is, but not why the code came to exist under a particular combination of human decisions, AI-generated suggestions, retrieved context, agent actions, and toolchain constraints. This explanatory gap matters because the attack surface is no longer limited to the model or final code.

This paper argues that modern software security needs a broader unit of analysis: the lineage of vulnerable code. Vulnerabilities may emerge not only because a model "made a mistake", but also through unsafe retrieved examples, misleading specifications, reused anti-patterns, rushed reviews, over-trusted suggestions, or agent and toolchain actions. For instance, an AI assistant may implement an authentication endpoint using an old internal example with weak token validation. The patch may pass functional tests and PR review, while a later SAST alert flags the unsafe flow. A lineage-aware analysis would connect the alert to the retrieved example, historical pattern, review decision, and AI interaction.

We argue for a layered lineage perspective with five complementary views: historical, technical, socio-technical, AI-contextual, and agent/toolchain/supply-chain. Rather than replacing code analysis, it embeds it in a broader framework that traces vulnerabilities across repository evolution, code evidence, human decisions, AI-facing context, and toolchain artifacts. As a position paper, our goal is not to validate a complete framework empirically, but to argue why lineage evidence should become an important concern in secure AI-assisted software development.

2 Background and Motivation

AI-assisted development is shifting software security from code-level vulnerability detection toward lineage-aware explanation, traceability, auditing, and prevention. In DevSecOps workflows, security evidence supports continuous feedback, automated testing, review, and release decisions rather than remaining a late-stage activity [5, 19]. However, AI-assisted code generation and integration introduce evidence spanning prompts, retrieved context, repository history, generated patches, reviews, dependencies, configurations, and build artifacts. This motivates analyzing risky code not only as a final snapshot, but as the outcome of a process shaped by human, technical, and AI-mediated evidence.

Prior work in mining software repositories provides a foundation for this view. SZZ-style analyses, history slicing, version-history mining, and change-coupling studies show that defects and vulnerabilities are often embedded in software evolution rather than isolated revisions [12, 16, 23, 34, 35]. These techniques suggest that vulnerable code may have a traceable history involving previous fixes, reused patterns, and co-evolving artifacts.

Program-analysis research complements this historical perspective by localizing risk in code. Program slicing, data-flow tracking, taint analysis, and code property graphs (CPG) reconstruct relationships relevant to security reasoning [21, 28, 30]. However, they generally provide code-level localization rather than lineage explanation. They can identify unsafe flows or vulnerable slices, but not whether risk was inherited, reproduced from historical patterns,

introduced by AI-generated code, missed during review, or shaped by context, agents, dependencies, or toolchain decisions.

Socio-technical research adds another layer. Developer–module networks, socio-technical congruence, PR dynamics, and review linkages show that software quality depends on coordination, review practices, and decision processes [2, 7, 9, 18, 25, 32]. In AI-assisted development, these traces become security-relevant because developers may over-trust plausible suggestions, reviewers may miss AI-introduced risks, and small PRs may hide broader contextual dependencies.

AI assistance further expands the relevant evidence. Studies of AI programming assistants show broad use across development activities alongside challenges in control, validation, understanding, and contextual fit [1, 14, 22, 26]. Prompts are also becoming software artifacts, despite immature lifecycle, versioning, and debugging practices [13]. Meanwhile, indirect prompt injection and context-attribution research shows that external context can influence model outputs in ways that are difficult to observe or faithfully attribute [3, 4, 8, 20, 27, 31].

Toolchain and supply-chain research shows that security evidence also extends beyond source code and human review. Modern software relies on external packages, build pipelines, configurations, generated artifacts, and automated tools. Supply-chain attacks demonstrate how dependencies, distribution, build steps, and development infrastructure create opportunities for compromise and propagation [11]. AI agents amplify this concern by suggesting dependencies, modifying configurations, invoking tools, and producing build artifacts.

Taken together, these strands motivate viewing AI-assisted development as a socio-technical, tool-mediated environment where vulnerabilities may emerge from interactions among repository history, code evidence, developer workflows, AI context, agent actions, and supply-chain/toolchain traces.

3 The Limits of Code-Only Vulnerability Detection

Code-only vulnerability detection remains necessary. Static analysis can identify suspicious API use, unsafe flows, dangerous patterns, and classes of weakness at scale. Dynamic analysis can expose failures under concrete executions. CPG and related graph representations can compactly encode semantic structure for vulnerability discovery [30]. LLM-based detectors and vulnerability reasoners may also help prioritize inspection or broaden coverage [15, 33], and remain valuable.

This limitation is more critical in AI-assisted development. A vulnerable artifact may result from a multi-stage interaction involving issue descriptions, prompts, retrieved files, IDE suggestions, developer edits, review comments, AI-generated revisions, agent actions, dependencies, configurations, and build artifacts [1]. If the question is only "is this code vulnerable?", code-level analysis may suffice. If it is "how did this vulnerability emerge, through which evidence, and how can similar recurrences be prevented?", code-only detection is insufficient.

The LLM should not be treated as the only attack surface. The broader AI-assisted development environment is a socio-technical attack surface spanning prompts, specifications, retrieved files, prior commits, PRs, code reviews, documentation, dependencies, configurations, developer decisions, context windows, toolchain integrations, and model-generated suggestions. A vulnerability may therefore emerge from the context pipeline rather than the model alone.

4 Toward a Layered Lineage Perspective

We propose a five-layer lineage model for reasoning about vulnerabilities in AI-assisted development. The layers are not sequential stages, but complementary views connected through shared development artifacts and events. Evidence from one layer can therefore contextualize or strengthen another. The model is conceptual but actionable, organizing evidence to move from isolated vulnerability detection toward traceable, auditable, and preventive security explanations. Table 1 summarizes the layers, their guiding questions, and their role in secure AI-assisted development.

Historical layer. The historical layer reconstructs whether a risky pattern previously appeared in the repository. Evidence may include previous commits, fixes, history slices, reused examples, and co-evolving artifacts. Its role is to explain whether a vulnerability is new, inherited, or reintroduced from earlier development history.

Technical layer. The technical layer anchors the explanation in code-level evidence. It identifies the vulnerable flow, slice, dependency, or semantic relation that makes the artifact security-relevant. Without this layer, lineage risks becoming speculative; with it, historical, socio-technical, and AI-contextual claims can be tied to concrete technical evidence.

Socio-technical layer. The socio-technical layer reconstructs how the vulnerable change was discussed, reviewed, and accepted. PRs, review comments, approvals, and developer–artifact relations help explain whether the risk passed through superficial review, fragmented discussion, unclear ownership, or delivery pressure.

AI-context layer. The AI-context layer examines what the model saw and what may have influenced its output. This includes prompts, retrieved repository artifacts, examples, generated suggestions, tool outputs, and attribution traces. Its role is to make prompt-driven and context-mediated generation auditable rather than opaque.

Agent/toolchain/supply-chain layer. The agent/toolchain/supply-chain layer captures non-code artifacts and automated actions shaping AI-assisted development. Evidence may include agent plans, tool calls, dependency changes, configurations, CI/CD logs, SBOM-like metadata, and build artifacts. This layer matters because vulnerabilities may arise not only from generated code, but also from unsafe configurations, dependencies, tool misuse, or compromised supply-chain steps.

The model's value lies in composition. A vulnerability explanation becomes stronger when code-level evidence is connected to historical origins, review decisions, AI-facing context, and toolchain traces. For example, a static warning may be linked to an unsafe historical pattern, rushed review, incomplete retrieved context, and an agent action introducing a vulnerable dependency or configuration, yielding a more actionable explanation that supports detection, traceability, auditing, and prevention.

Table 1: A layered lineage model for vulnerabilities in AI-assisted software development.

Layer	Guiding question	Lineage evidence	Security use
1 - Historical	Where did the risky pattern come from?	Commits, fixes, history slices, reused examples, change coupling	Identify recurring unsafe patterns and prevent reintroduction
2 - Technical	Where is the risk in the code?	Unsafe flows, slices, data dependencies, CPGs	Localize, validate, and prioritize vulnerability findings
3 - Socio-technical	How was the decision accepted?	PR, review comments, approvals, developer–artifact links	Explain review outcomes and improve accountability
4 - AI-context	What did the AI see and use?	Prompts, retrieved files, generated suggestions, attribution traces	Audit context influence and prompt-driven risks
5 - Agent/toolchain/supply-chain	Which tools and artifacts shaped the result?	Agent plans, tool calls, dependencies, configurations, CI/CD logs, build artifacts	Trace agent actions, toolchain effects, and supply-chain risks

5 Research Challenges and Agenda

The layered lineage perspective reframes vulnerability analysis as an explanatory and preventive problem, but making this perspective usable in AI-assisted DevSecOps raises several research challenges.

Incomplete observability and audit trails. This calls for selective, privacy-aware audit trails that capture security-relevant provenance, such as generated suggestions, retrieved artifacts, agent actions, configuration changes, and review decisions, extending the use of provenance as a basis for explanation and accountability in automated systems [10].

False attribution and evidential uncertainty. AI assistance does not imply that a vulnerability was caused by AI. Risky code may originate from legacy commits, internal examples, human error, review pressure, dependencies, or toolchain decisions. AI-generated code detectors may indicate whether a fragment was likely produced by an LLM, but recent evidence shows limited generalizability and they should not be treated as causal evidence [24]. A credible lineage approach must therefore treat attribution as uncertain and evidence-based. This motivates controlled-provenance benchmarks using synthetic repositories, instrumented scenarios, red-team exercises, and sanitized industrial datasets where the origins of prompts, retrieved context, agent actions, and vulnerable changes are partially known.

Cross-layer alignment. Lineage evidence spans different granularities, including statements, functions, commits, PRs, reviews, prompts, retrieved chunks, dependencies, configurations, CI/CD events, and build artifacts. These units do not naturally align. Existing graph-based representations capture parts of this space, such as CPG and review linkage graphs [9, 30]. However, AI-assisted development requires extending these partial representations into a heterogeneous developer–artifact–AI–toolchain graph, connecting developers, code artifacts, reviews, prompts, generated suggestions, tool calls, dependencies, and builds through relations such as authorship, modification, retrieval, generation, review, execution, dependency, and temporal precedence.

Agentic and supply-chain complexity. AI-assisted development is moving beyond code completion toward agentic workflows. Agents may inspect files, call tools, modify configurations, add dependencies, execute tests, trigger builds, and create PRs. Vulnerabilities may arise not only from generated code, but also from unsafe dependencies, configurations, excessive permissions, compromised build steps, or tool misuse. Future work should treat agent actions, toolchain integrations, dependencies, and build artifacts as first-class lineage evidence.

Making Lineage Actionable. A lineage-aware DevSecOps workflow could therefore support PR review, SAST/SCA triage, CI/CD gates, audit reports, and recurrence prevention. The central question is not only how to detect vulnerable code, but how to explain why it emerged, decide whether it should be accepted, and prevent similar risks from reappearing in future AI-assisted development.

6 Conclusion

This paper argued that securing AI-assisted software development requires moving beyond code-only vulnerability detection toward lineage-aware explanation. Static and dynamic analysis, graph-based methods, SCA, and LLM-based detectors remain essential, but offer limited insight into how vulnerable code is produced, accepted, and propagated. Our central position is that vulnerabilities in AI-assisted environments should be examined across five complementary lineage layers: repository history, code-level evidence, socio-technical decisions, AI-facing context, and agent/toolchain/supply-chain traces. This perspective treats the broader development environment, not just the LLM or final code, as security-relevant. Prompts, retrieved files, reviews, generated suggestions, dependencies, configurations, tool calls, and build artifacts may all contribute to vulnerabilities.

Although a lineage-aware framework cannot eliminate uncertainty, it can make security evidence more traceable, auditable, and actionable. By connecting vulnerability findings to historical origins, review trajectories, AI-contextual influences, and toolchain effects, it can support DevSecOps practices that explain risk emergence, inform acceptance decisions, and help prevent recurrence.

Artifact Availability

No artifacts are available for this position paper.

Acknowledgements

Research supported by HP Brasil Indústria e Comércio de Equipamentos Eletrônicos Ltda. using financial incentives of IPI refund regarding the Law (Law nº 8.248 of 1991) and financially supported

by Conselho Nacional de Desenvolvimento Científico e Tecnológico - Brazil (INCT SiMAI, CNPq 408330/2024-4). Generative AI tools were used only for language editing and clarity. The authors reviewed the final text and remain responsible for its content.

References

[1] Shraddha Barke, Michael B. James, and Nadia Polikarpova. 2023. Grounded Copilot: How Programmers Interact with Code-Generating Models. *Proceedings of the ACM on Programming Languages* 7, OOPSLA1, Article 78 (2023), 27 pages. doi:10.1145/3586030

[2] Christian Bird, Nachiappan Nagappan, Harald Gall, Brendan Murphy, and Premkumar Devanbu. 2009. Putting It All Together: Using Socio-Technical Networks to Predict Failures. In *Proceedings of the 20th International Symposium on Software Reliability Engineering*. IEEE, 109–119. doi:10.1109/ISSRE.2009.17

[3] Yung-Sung Chuang, Benjamin Cohen-Wang, Zejiang Shen, Zhaofeng Wu, Hu Xu, Xi Victoria Lin, James R. Glass, Shang-Wen Li, and Wen-tau Yih. 2025. SelfCite: Self-Supervised Alignment for Context Attribution in Large Language Models. In *Proceedings of the 42nd International Conference on Machine Learning (Proceedings of Machine Learning Research, Vol. 267)*. PMLR, 10839–10858.

[4] Benjamin Cohen-Wang, Harshay Shah, Kristian Georgiev, and Aleksander Madry. 2024. ContextCite: Attributing Model Generation to Context. In *Proceedings of the 38th International Conference on Neural Information Processing Systems* (Vancouver, BC, Canada) *(NIPS '24)*. Curran Associates Inc., Red Hook, NY, USA, Article 3035, 44 pages. doi:10.52202/079017-3035

[5] Clarisse Feio, Nuno Santos, Nelson Escravana, and Bernardo Pacheco. 2024. An Empirical Study of DevSecOps Focused on Continuous Security Testing. In *2024 IEEE European Symposium on Security and Privacy Workshops (EuroSPW)*. IEEE, Vienna, Austria, 610–617. doi:10.1109/EuroSPW61312.2024.00074

[6] Yujia Fu, Peng Liang, Amjed Tahir, Zengyang Li, Mojtaba Shahin, Jiaxin Yu, and Jinfu Chen. 2025. Security Weaknesses of Copilot-Generated Code in GitHub Projects: An Empirical Study. *ACM Transactions on Software Engineering and Methodology* 34, 8, Article 218 (2025), 34 pages. doi:10.1145/3716848

[7] Georgios Gousios, Martin Pinzger, and Arie van Deursen. 2014. An Exploratory Study of the Pull-Based Software Development Model. In *Proceedings of the 36th International Conference on Software Engineering*. ACM, 345–355. doi:10.1145/2568225.2568260

[8] Kai Greshake, Sahar Abdelnabi, Shailesh Mishra, Christoph Endres, Thorsten Holz, and Mario Fritz. 2023. Not What You've Signed Up For: Compromising Real-World LLM-Integrated Applications with Indirect Prompt Injection. In *Proceedings of the 16th ACM Workshop on Artificial Intelligence and Security*. ACM, 79–90. doi:10.1145/3605764.3623985

[9] Toshiki Hirao, Shane McIntosh, Akinori Ihara, and Kenichi Matsumoto. 2019. The Review Linkage Graph for Code Review Analytics: A Recovery Approach and Empirical Study. In *Proceedings of the 27th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, 578–589. doi:10.1145/3338906.3338949

[10] Trung Dong Huynh, Niko Tsakalakis, Ayah Helal, Sophie Stalla-Bourdillon, and Luc Moreau. 2021. Addressing Regulatory Requirements on Explanations for Automated Decisions with Provenance—A Case Study. *Digital Government: Research and Practice* 2, 2, Article 16e (2021), 14 pages. doi:10.1145/3436897

[11] Piergiorgio Ladisa, Henrik Plate, Matias Martinez, and Olivier Barais. 2023. SoK: Taxonomy of Attacks on Open-Source Software Supply Chains. In *2023 IEEE Symposium on Security and Privacy (SP)*. IEEE, San Francisco, CA, USA, 1509–1526. doi:10.1109/SP46215.2023.10179304

[12] Yi Li, Chenguang Zhu, Julia Rubin, and Marsha Chechik. 2016. Precise Semantic History Slicing through Dynamic Delta Refinement. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*. ACM, 495–506. doi:10.1145/2970276.2970336

[13] Jenny T. Liang, Melissa Lin, Nikitha Rao, and Brad A. Myers. 2025. Prompts Are Programs Too! Understanding How Developers Build Software Containing Prompts. *Proceedings of the ACM on Software Engineering* 2, FSE, Article FSE072 (2025), 24 pages. doi:10.1145/3729342

[14] Jenny T. Liang, Chenyang Yang, and Brad A. Myers. 2024. A Large-Scale Survey on the Usability of AI Programming Assistants: Successes and Challenges. In *Proceedings of the IEEE/ACM International Conference on Software Engineering*. Article 52, 13 pages. doi:10.1145/3597503.3608128

[15] Andrew A. Mahyari. 2024. Harnessing the Power of LLMs in Source Code Vulnerability Detection. In *Proceedings of the IEEE Military Communications Conference*. IEEE, 251–256. doi:10.1109/MILCOM61039.2024.10774025

[16] Gustavo A. Oliva and Marco A. Gerosa. 2015. Change Coupling Between Software Artifacts: Learning from Past Changes. In *The Art and Science of Analyzing Software Data*, Christian Bird, Tim Menzies, and Thomas Zimmermann (Eds.). Morgan Kaufmann, 285–323. doi:10.1016/B978-0-12-411519-4.00011-2

[17] Hammond Pearce, Baleegh Ahmad, Benjamin Tan, Brendan Dolan-Gavitt, and Ramesh Karri. 2022. Asleep at the Keyboard? Assessing the Security of GitHub Copilot's Code Contributions. In *Proceedings of the IEEE Symposium on Security and Privacy*. IEEE, 754–768. doi:10.1109/SP46214.2022.9833571

[18] Martin Pinzger, Nachiappan Nagappan, and Brendan Murphy. 2008. Can Developer-Module Networks Predict Failures?. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 2–12. doi:10.1145/1453101.1453105

[19] Luís Prates and Rúben Pereira. 2025. DevSecOps Practices and Tools. *International Journal of Information Security* 24, 1 (Feb. 2025), 11. doi:10.1007/s10207-024-00914-z

[20] Jirui Qi, Gabriele Sarti, Raquel Fernández, and Arianna Bisazza. 2024. Model Internals-Based Answer Attribution for Trustworthy Retrieval-Augmented Generation. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics, 6037–6053. doi:10.18653/v1/2024.emnlp-main.347

[21] Edward J. Schwartz, Thanassis Avgerinos, and David Brumley. 2010. All You Ever Wanted to Know about Dynamic Taint Analysis and Forward Symbolic Execution (but Might Have Been Afraid to Ask). In *2010 IEEE Symposium on Security and Privacy*. IEEE, 317–331. doi:10.1109/SP.2010.26

[22] Agnia Sergeyuk, Yaroslav Golubev, Timofey Bryksin, and Iftekhar Ahmed. 2025. Using AI-Based Coding Assistants in Practice: State of Affairs, Perceptions, and Ways Forward. *Information and Software Technology* 178 (2025), 107610. doi:10.1016/j.infsof.2024.107610

[23] Francisco Servant and James A. Jones. 2012. History Slicing: Assisting Code-Evolution Tasks. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*. ACM, Cary, North Carolina, USA, 1–11. doi:10.1145/2393596.2393646

[24] Hyunjae Suh, Mahan Tafreshipour, Jiawei Li, Adithya Bhattiprolu, and Iftekhar Ahmed. 2025. An Empirical Study on Automatically Detecting AI-Generated Source Code: How Far Are We?. In *Proceedings of the IEEE/ACM 47th International Conference on Software Engineering* (Ottawa, Ontario, Canada) *(ICSE '25)*. IEEE Press, 859–871. doi:10.1109/ICSE55347.2025.00064

[25] Jason Tsay, Laura Dabbish, and James Herbsleb. 2014. Influence of Social and Technical Factors for Evaluating Contribution in GitHub. In *Proceedings of the 36th International Conference on Software Engineering*. ACM, 356–366. doi:10.1145/2568225.2568315

[26] Priyan Vaithilingam, Tianyi Zhang, and Elena L. Glassman. 2022. Expectation vs. Experience: Evaluating the Usability of Code Generation Tools Powered by Large Language Models. In *CHI Conference on Human Factors in Computing Systems Extended Abstracts*. ACM. doi:10.1145/3491101.3519665

[27] Jonas Wallat, Maria Heuss, Maarten de Rijke, and Avishek Anand. 2025. Correctness is not Faithfulness in Retrieval Augmented Generation Attributions. In *Proceedings of the 2025 International ACM SIGIR Conference on Innovative Concepts and Theories in Information Retrieval* (Padua, Italy) *(ICTIR '25)*. Association for Computing Machinery, New York, NY, USA, 22–32. doi:10.1145/3731120.3744592

[28] Mark Weiser. 1981. Program Slicing. In *Proceedings of the 5th International Conference on Software Engineering* (San Diego, California, USA) *(ICSE '81)*. IEEE Press, 439–449.

[29] Justin D. Weisz, Shraddha Kumar, Michael Muller, Karen-Ellen Browne, Arielle Goldberg, Ellice Heintze, and Shagun Bajpai. 2025. Examining the Use and Impact of an AI Code Assistant on Developer Productivity and Experience in the Enterprise. In *Extended Abstracts of the CHI Conference on Human Factors in Computing Systems*. ACM, Article 673, 13 pages. doi:10.1145/3706599.3706670

[30] Fabian Yamaguchi, Nico Golde, Daniel Arp, and Konrad Rieck. 2014. Modeling and Discovering Vulnerabilities with Code Property Graphs. In *2014 IEEE Symposium on Security and Privacy*. IEEE, 590–604. doi:10.1109/SP.2014.44

[31] Jingwei Yi, Yueqi Xie, Bin Zhu, Emre Kiciman, Guangzhong Sun, Xing Xie, and Fangzhao Wu. 2025. Benchmarking and Defending against Indirect Prompt Injection Attacks on Large Language Models. In *Proceedings of the 31st ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, Vol. 1. ACM, 1809–1820. doi:10.1145/3690624.3709179

[32] Weiqiang Zhang, Shing-Chi Cheung, Zhenyu Chen, Yuming Zhou, and Bin Luo. 2019. File-Level Socio-Technical Congruence and its Relationship with Bug Proneness in OSS Projects. *Journal of Systems and Software* 156 (2019), 21–40. doi:10.1016/j.jss.2019.05.030

[33] Xin Zhou, Ting Zhang, and David Lo. 2024. Large Language Model for Vulnerability Detection: Emerging Results and Future Directions. In *Proceedings of the 2024 ACM/IEEE 44th International Conference on Software Engineering: New Ideas and Emerging Results* (Lisbon, Portugal) *(ICSE-NIER '24)*. Association for Computing Machinery, New York, NY, USA, 47–51. doi:10.1145/3639476.3639762

[34] Thomas Zimmermann, Peter Weißgerber, Stephan Diehl, and Andreas Zeller. 2005. Mining Version Histories to Guide Software Changes. *IEEE Transactions on Software Engineering* 31, 6 (2005), 429–445. doi:10.1109/TSE.2005.72

[35] Jacek Śliwerski, Thomas Zimmermann, and Andreas Zeller. 2005. When Do Changes Induce Fixes?. In *Proceedings of the International Workshop on Mining Software Repositories*. ACM, 1–5. doi:10.1145/1082983.1083147