ICASSP 2025 - 2025 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP) | 979-8-3503-6874-1/25/$31.00 ©2025 IEEE | DOI: 10.1109/ICASSP49660.2025.10890824

RefleXGen:The unexamined code is not worth using

Bin Wang[1], Hui Li[1*], AoFan Liu[1], BoTao Yang[1], Ao Yang[1], YiLu Zhong[1],
Weixiang Huang[2], Runhuai Huang[3], Weimin Zeng[3], Yanping Zhang[2]
[1]School of Electronic and Computer Engineering, Peking University, Shenzhen, China
[2]Capability & Platform Business Dept., China Mobile Internet Co., Beijing, China
[3]China Telecom Cloud Technology Co., Ltd., Guangzhou, China
2201111747@stu.pku.edu.cn, lih64@pkusz.edu.cn, af.liu@stu.pku.edu.cn,
renrulongky999@gmail.com jarvisya@stu.pku.edu.cn, tangaaang@gmail.com,
huangweixiang@cmic.chinamobile.com, huangrh@chinatelecom.cn
zhangyanping@cmic.chinamobile.com, zengwm@chinatelecom.cn

***Abstract*—Security in code generation remains a pivotal challenge when applying large language models (LLMs). This paper introduces RefleXGen, an innovative method that significantly enhances code security by integrating Retrieval-Augmented Generation (RAG) techniques with guided self-reflection mechanisms inherent in LLMs. Unlike traditional approaches that rely on fine-tuning LLMs or developing specialized secure code datasets—processes that can be resource-intensive—RefleXGen iteratively optimizes the code generation process through self-assessment and reflection without the need for extensive resources. Within this framework, the model continuously accumulates and refines its knowledge base, thereby progressively improving the security of the generated code. Experimental results demonstrate that RefleXGen substantially enhances code security across multiple models, achieving a 13.6% improvement with GPT-3.5 Turbo, a 6.7% improvement with GPT-4o, a 4.5% improvement with CodeQwen, and a 5.8% improvement with Gemini.**

***Index Terms*—code generation, security, large language models, RAG**

I. Introduction

Code generation technologies, which enable the creation of target code via natural language descriptions or minimal code prompts, significantly lower the barriers to software development. They allow a broader range of non-experts to engage in software development and substantially reduce the workload for developers. Initially, code generation relied heavily on heuristic rules or expert systems. While effective, these methods often lacked flexibility and scalability [1]. Subsequently, researchers began using static language models and neural networks to establish mappings between codes, which expanded the applications, but were still limited in functionality [2], [3].

With the advent of LLMs based on the Transformer architecture, an increasing number of LLMs have been trained on extensive code corpora [4]–[9]. These models can generate code without the need for samples and have demonstrated remarkable success across numerous code generation tasks. These advanced large language models have significantly propelled the evolution of code generation technologies. They are capable of generating, optimizing, and even debugging code based on user requirements, thus markedly enhancing software development efficiency and opening new programming avenues for nonprofessional programmers. According to the 2023 GitHub Annual Report, nearly all developers (92%) are utilizing or experimenting with AI programming tools, which have become powerful aids in accelerating development cycles and increasing productivity [10], [11].

However, large language models for code completion and generation have shortcomings [6], [8], [11]. Pre-trained on publicly available datasets, the training code is not guaranteed to be safe or reliable. Consequently, the generated code may contain defects or vulnerabilities. These problems can cause serious problems for users, such as low-quality code, compilation failures, or security vulnerabilities, problems that are more direct than hallucinations or errors in dialogue generation [11], [12]. Therefore, enhancing the ability of language models to generate reliable and secure code is a significant challenge in current research.

To effectively address security challenges in code generation, we have developed an innovative method called RefleXGen. This approach enhances code generation by guiding large language models to engage in self-reflection, coupled with a knowledge base composed of the model's own historical thought records and secure code snippets. As a result, it significantly improves the security of the generated code. Throughout this process, the model autonomously identifies and mitigates potential security risks, accumulates practices in secure coding, and progressively enriches the knowledge base to guide the generation of future secure code. Notably, RefleXGen does not require updates to existing training datasets nor fine-tuning of the model, and can be seamlessly integrated into existing large-scale models. Verification on multiple proprietary and open-source models has confirmed RefleXGen's substantial effectiveness in enhancing the security of code generation.

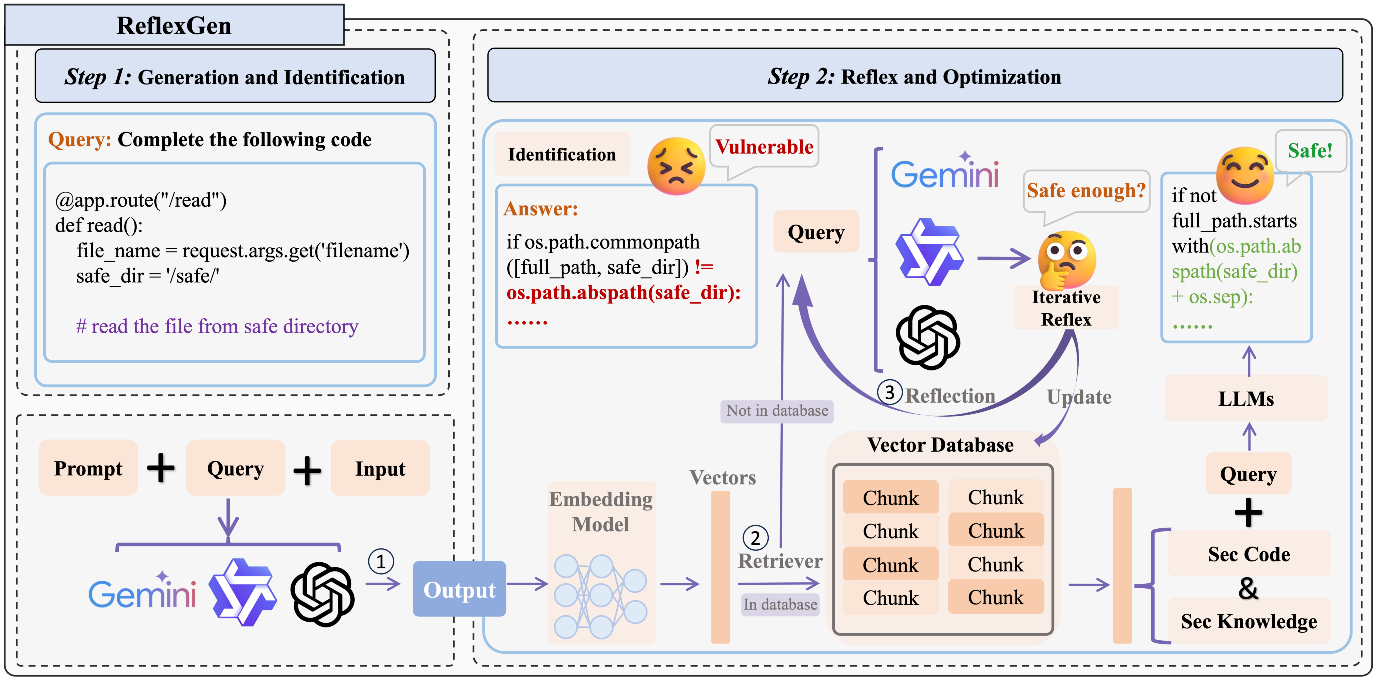
II. Related Work

A. Code Generation

Code generation has a long history, traditionally defined as finding programs within a programming language's search space that satisfy task-specific constraints [13]. However,

Corresponding author. This work is supported by Guangdong Provincial Key Laboratory of Ultra High Definition Immersive Media Technology (Grant No. 2024B1212010006)

Authorized licensed use limited to: IEEE Xplore. Downloaded on March 09,2025 at 03:54:44 UTC from IEEE Xplore. Restrictions apply.



```
@app.route("/read")
def read():
    file_name = request.args.get('filename')
    safe_dir = '/safe/'

    # read the file from safe directory
```

```
if os.path.commonpath
([full_path, safe_dir]) !=
os.path.abspath(safe_dir):
……
```

```
if not
full_path.starts
with(os.path.ab
spath(safe_dir)
+ os.sep):
……
```

Fig. 1. The diagram presents the structured workflow of the ReflexGen methodology, segmented into three critical stages: ① Initial Code Generation, ② Knowledge-Driven Security Feedback, and ③ Defect Fixing and Knowledge Integration. The process initiates with the generation of initial code. If, upon introspection, the model discerns security deficiencies in the code, it activates Step 2. This stage entails rigorous reflection and optimization to address and rectify vulnerabilities. Subsequently, through a cyclical process of secure code production, insights derived from this reflective phase are systematically integrated into the security knowledge base, thus promoting continual enhancements.

search-based methods often struggle due to the vastness of the search space and the lack of formalized constraints. With advancements in deep learning, new approaches have emerged that generate programs from informal specifications such as natural language descriptions, partial code, input-output examples, or pseudocode [2], [3], [14]. Despite progress, these methods are typically limited to generating shorter programs in domain-specific languages or single lines of code in general-purpose languages.

Recently, transformer-based large language models have revitalized code generation. Models like Codex [4] demonstrate exceptional capability in auto-completing Python functions based on function signatures and docstrings. CodeGen [15] enhances program synthesis quality through multi-turn interactions that refine user specifications. CodeT5 [8] introduces automatic test case generation to improve code solution selection. CodeRanker [16] presents a fault-aware ranking model that predicts program correctness without execution, effectively addressing code selection challenges.

B. Code Generation Security

The security of code generation has become a critical research area in large language models. Extensive studies have analyzed and evaluated the security of these models, highlighting vulnerabilities in generated code. Models like StarCoder [5] and CodeLlama [17] have implemented specific security-enhancing measures during training. Additionally, works like SecurityEval [18] and SecuCoGen [19] focus on assessing models' ability to generate secure code. Techniques such as SafeCoder [20], FRANC, and SVEN [11] enhance code generation security from different perspectives, introducing innovative mechanisms and algorithms to improve the safety of generated code.

III. Methodology

The RefleXGen method integrates the concept of Self-refinement with RAG technology, aiming to enhance the safety of LLMs in code generation without the need for fine-tuning the model itself. As illustrated in Figure 1, the workflow of this method encompasses two key phases and three core operations. In the first phase, the code generation model produces initial code based on specific user requirements. Subsequently, in the second phase, RefleXGen performs deep reflection and iterative optimization on this initial code. Once the code's safety meets the predetermined standards, the reflective insights are updated into the safety knowledge base to guide subsequent related tasks. The following discussion will introduce the crucial operations involved.

Step1:① Initial code generation. In the stage, the system is provided with an input code snippet $\mathbf{x}$, a prompt $\mathbf{p}_{\text{gen}}$, and accesses the model $\mathcal{M}$. The code generation model then produces the initial output y_0:

$$y_0 = \mathcal{M}(\mathbf{p}_{\text{gen}} \| \mathbf{x}) \tag{1}$$

While this initial output generally meets the basic requirements outlined in the input, it may still present issues such as

Authorized licensed use limited to: IEEE Xplore. Downloaded on March 09,2025 at 03:54:44 UTC from IEEE Xplore. Restrictions apply.

poor reliability or contain latent security vulnerabilities that necessitate further refinement.

Step2:Reflection and Optimization. In this step, the system initially employs its model to introspect and determine the presence of any potential defects in the output. Should the output be defect-free, the system will proceed to display the results directly. However, if defects are identified, the system transitions into a phase of reflective iteration. The specific steps involved in this phase are as follows:

②Knowledge-Driven Security Feedback:In this stage, RefleXGen conducts a RAG query utilizing both the initial code output and specific input requests, as outlined in Equations 2 and 3. The query is designed to uncover pertinent security knowledge, including standards for secure coding and historical feedback. When the query identifies applicable security practices and knowledge, the system integrates this information with the initial input and the defined problem .

$$\mathbf{r}_0 = \text{Retrieve}(\mathbf{x}, y_0) \tag{2}$$

$$y_1 = \mathcal{M}(\mathbf{p}_{\text{gen}} \| \mathbf{x} \| y_0 \| \mathbf{r}_0) \tag{3}$$

③ Defect Fixing and Knowledge Integration: If the RAG query fails to provide sufficient security knowledge, the system proceeds to a thorough reflection and iterative repair process, as outlined in Equation 4. This phase involves a critical assessment and enhancement of the code based on identified vulnerabilities and potential improvements. Once the code fulfills all specified safety requirements, the refined security knowledge and the enhanced code are systematically organized and stored within the secure knowledge base (sec. RAG). Subsequently, the system reinitiates the first step to verify the output, ensuring that the improvements effectively address the initial shortcomings.

$$y_{t+1} = \mathcal{M}(\mathbf{p}_{\text{refine}} \| \mathbf{x} \| y_0 \| \mathbf{fb}_0 \| \ldots \| y_t \| \mathbf{fb}_t \| \mathbf{r}_t) \tag{4}$$

$$\text{UpdateRAG}(\mathbf{x}, y_{t+1}) \tag{5}$$

By reflecting on the code and incorporating historical data, RefleXGen readjusts the code, repairs insecure parts, and even introduces safer coding practices. The optimized code not only meets the initial functional requirements but also significantly enhances its security.

In our approach, each iteration builds upon the results of the previous one, combining insights from prior outputs to dynamically update the knowledge base. This evolving knowledge base enables the system to more effectively identify and address potential security vulnerabilities over time. Specifically, the system leverages the capabilities of the original pre-trained model to reflect on and assess potential defects in the output. If the output is deemed free of defects, the code is directly delivered; otherwise, the reflection process guides defect remediation while simultaneously updating the knowledge base to prevent similar issues in future iterations.

This iterative feedback loop forms the foundation of our method's ability to dynamically and adaptively enhance the security of code generation. Moreover, it is seamlessly applicable to both open-source and proprietary models.

IV. EXPERIMENT

A. Model Selection

Due to the limitations of smaller open-source and specialized code-completion models in dialogue and reflective knowledge assessment, we selected more comprehensive mainstream models for our evaluation. These include prominent commercial models like OpenAI's GPT-3.5 Turbo and GPT-4, Google's Gemini, and the open-source model Qwen. These models exhibit advanced code generation capabilities and excel in managing dialogues, aligning well with our testing criteria.

B. Datasets

To evaluate RefleXGen's improvements in code generation security and reliability, we selected challenging scenarios from the most impactful Common Weakness Enumerations (CWEs). We used a dataset validated by He et al. [11], featuring nine scenarios from MITRE's top 25 most dangerous software vulnerabilities. Each CWE scenario includes two to three specific programming environments crafted by He et al., eliminating low-quality prompts and replicating diverse daily code completion tasks, making it a robust tool for assessing models' code security capabilities. These scenarios, based on incomplete code prompts in C/C++ or Python, challenge the models to produce appropriate code completions, highlighting their ability to handle incomplete inputs in real programming environments.

C. Performance of RefleXGen

To ensure a fair comparison, we initialized the RAG content to empty at the start of testing, allowing RefleXGen to progressively generate and refine its content during the evaluation process. We tracked several key metrics, including Security Rate (Sec. Rate), Pass Rate (Pass Rate), Effective Total (Eff. Total), Security Issue Count (Sec. Count), and Unresolved Issue Count (Unres. Count).

In our experiments, we adopted CodeQL [21] as the core tool for comprehensive security analysis of the generated code. CodeQL is a powerful static analysis tool that uses a query language to efficiently detect code vulnerabilities and security risks. Specifically, CodeQL's analysis relies on a pre-defined rule library that enables the rapid identification of potential issues, such as SQL injections, uninitialized variables, and buffer overflows. Furthermore, CodeQL's flexibility allows for the customization of rules to perform in-depth checks for security standards in specific scenarios. In this study, the intuitive outputs provided by CodeQL served as the basis for rigorously assessing whether the generated results adhered to the required security standards, thereby offering a precise and reliable evaluation of RefleXGen's security performance.

To ensure the reliability and stability of the results, we conducted five repeated experiments for each model. In each experiment, every test scenario involved 25 task generations, with the results averaged to evaluate the security and quality of the generated outputs. This design ensures the objectivity,

Authorized licensed use limited to: IEEE Xplore. Downloaded on March 09,2025 at 03:54:44 UTC from IEEE Xplore. Restrictions apply.

TABLE I

THE PASS RATE (PASS RATE) REFERS TO THE PERCENTAGE OF CORRECT OUTPUTS FROM VALID INPUTS. THE SECURITY RATE (SEC. RATE) REFLECTS THE PERCENTAGE OF SUCCESSFULLY COMPILED TESTS THAT ALSO MEET SECURITY STANDARDS. THE "EFFICIENCY TOTAL" (EFF. TOTAL) INDICATES THE NUMBER OF SUCCESSFULLY COMPILED TESTS WITHIN A CWE CATEGORY, WHILE THE "SECURITY COUNT" (SEC. COUNT) SHOWS THE NUMBER OF TESTS THAT COMPILED SUCCESSFULLY AND MET SECURITY STANDARDS, BOTH OUT OF 25 TESTS. FINALLY, THE UNRESOLVED COUNT (UNRES. COUNT) REFLECTS THE FIGURE OF TESTS THAT FAILED TO COMPILE.

2*Model	GPT3.5Turbo		GPT4o		CodeQwen1.5		Gemini1.0Pro	
	Base	+RefleXGen	Base	+RefleXGen	Base	+RefleXGen	Base	+RefleXGen
Sec.Rate	75.5	89.1 (↑**13.6**)	92.3	99.0 (↑**6.7**)	83.7	88.2 (↑**4.5**)	80.2	86.0 (↑**5.8**)
Pass.Rate	97.6	95.8 (↓1.8)	94.2	100 (↑**5.8**)	86.7	69.8 (↓16.9)	92.2	83.6 (↓8.6)
Eff.Total	24.5	24.0 (↓0.5)	23.6	25.0 (↑**1.4**)	21.6	20.4 (↓1.2)	23.1	22.8 (↓0.3)
Sec.Count	19.5	22.3 (↑**2.8**)	21.9	24.7 (↑**2.8**)	17.9	19.4 (↑**1.5**)	19.1	21.2 (↑**2.1**)
Unres.Count	0.5	1.1 (↑**0.6**)	1.4	0 (↓1.4)	3.3	3.8 (↑**0.5**)	1.9	2.1 (↑**0.2**)

consistency, and reproducibility of our assessment of the models' generative capabilities.

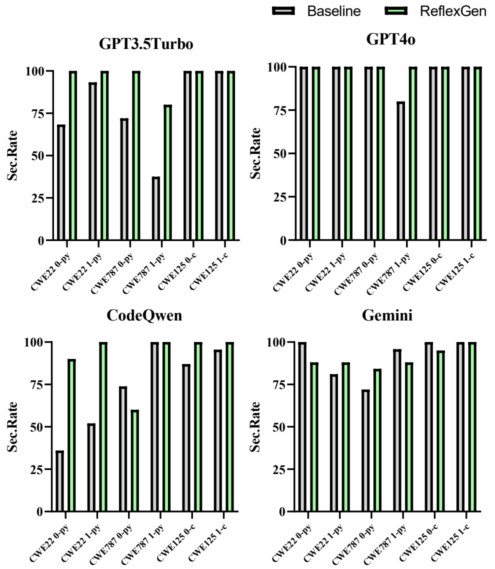


Fig. 2. Sec.Rate Difference among Cases of RefleXGen

As shown in Table I, RefleXGen demonstrated outstanding performance across four major models, effectively enhancing code security. Specifically, OpenAI's GPT-3.5 Turbo showed a 13.6% improvement in code safety, GPT-4 improved by 6.7%, CodeQwen-1.5 by 4.5%, and Gemini-1.0-pro achieved a 5.8% increase in security. These results indicate that the RefleXGen method significantly reduces the rate of defects and problematic code generation across different models.

Furthermore, we conducted a detailed analysis of three CWE scenarios that are typically concealed yet pose severe risks. As illustrated in Figure 2, under the RefleXGen method, the code security generated by GPT-3.5 and CodeQwen demonstrated significant improvements in scenarios prone to triggering high-risk vulnerabilities. In contrast, Gemini exhibited fluctuations in security enhancements, while the improvements in GPT-4 were relatively modest, likely due to its already high baseline of code safety.

It is worth noting that, except for GPT-4, the initial compilation success rate for other models declined. This decline is primarily attributed to the introduction of more restrictive conditions and code interferences, which added complexity to the tasks. These changes led to more complex code outputs, thereby affecting the compilation success rates. However, GPT-4, with its robust overall capabilities, was less affected and even showed an improvement in compilation success. In contrast, CodeQwen, which has a smaller parameter size, experienced a greater decline. This phenomenon underscores the dependency of RefleXGen's enhancements on the models' capabilities in dialogue and handling complex scenarios.

V. CONCLUSION

In this work, we have introduced RefleXGen, an innovative method that significantly enhances the security of code generated by large language models without the need for model fine-tuning or the creation of specialized security datasets. Universally applicable to all code generation models and operating independently of external enhancements, RefleXGen leverages the models' inherent reflective processes to accumulate security knowledge. By building a dynamic knowledge base, it optimizes prompts for subsequent code generation cycles. Experimental results demonstrate that RefleXGen substantially improves code generation security across various models, including GPT-3.5, GPT-4, CodeQwen, and Gemini, with particularly notable enhancements in models possessing stronger overall capabilities. This advancement underscores the potential of self-reflective mechanisms in AI models to autonomously improve code security, paving the way for future research in secure code generation without extensive resource investment.

REFERENCES

[1] X.-Y. Li, J.-T. Xue, Z. Xie, and M. Li, "Think outside the code: Brainstorming boosts large language models in code generation," *arXiv preprint arXiv:2305.10679*, 2023.

[2] W. Ling, E. Grefenstette, K. M. Hermann, T. Kočiskỳ, A. Senior, F. Wang, and P. Blunsom, "Latent predictor networks for code generation," *arXiv preprint arXiv:1603.06744*, 2016.

[3] V. Raychev, P. Bielik, and M. Vechev, "Probabilistic model for code with decision trees," *ACM SIGPLAN Notices*, vol. 51, no. 10, pp. 731–747, 2016.

[4] OpenAI, "Openai codex," 2021, accessed: 2024-08-18. [Online]. Available: https://openai.com/index/openai-codex/

Authorized licensed use limited to: IEEE Xplore. Downloaded on March 09,2025 at 03:54:44 UTC from IEEE Xplore. Restrictions apply.

[5] A. Lozhkov, R. Li, L. B. Allal, F. Cassano, J. Lamy-Poirier, N. Tazi, A. Tang, D. Pykhtar, J. Liu, Y. Wei *et al.*, "Starcoder 2 and the stack v2: The next generation," *arXiv preprint arXiv:2402.19173*, 2024.

[6] A. Chowdhery, S. Narang, J. Devlin, M. Bosma, G. Mishra, A. Roberts, P. Barham, H. W. Chung, C. Sutton, S. Gehrmann *et al.*, "Palm: Scaling language modeling with pathways," *Journal of Machine Learning Research*, vol. 24, no. 240, pp. 1–113, 2023.

[7] D. Xiao, H. Zhang, Y. Li, Y. Sun, H. Tian, H. Wu, and H. Wang, "Ernie-gen: An enhanced multi-flow pre-training and fine-tuning framework for natural language generation," *arXiv preprint arXiv:2001.11314*, 2020.

[8] Y. Wang, W. Wang, S. Joty, and S. C. Hoi, "Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation," *arXiv preprint arXiv:2109.00859*, 2021.

[9] Y. Li, D. Choi, J. Chung, N. Kushman, J. Schrittwieser, R. Leblond, T. Eccles, J. Keeling, F. Gimeno, A. Dal Lago *et al.*, "Competition-level code generation with alphacode," *Science*, vol. 378, no. 6624, pp. 1092–1097, 2022.

[10] P. Vaithilingam, T. Zhang, and E. L. Glassman, "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*, 2022, pp. 1–7.

[11] J. He and M. Vechev, "Large language models for code: Security hardening and adversarial testing," in *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, 2023, pp. 1865–1879.

[12] H. Pearce, B. Ahmad, B. Tan, B. Dolan-Gavitt, and R. Karri, "Asleep at the keyboard? assessing the security of github copilot's code contributions," in *2022 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2022, pp. 754–768.

[13] C. Green, "Application of theorem proving to problem solving," in *Readings in Artificial Intelligence*. Elsevier, 1981, pp. 202–222.

[14] Z. Sun, Q. Zhu, Y. Xiong, Y. Sun, L. Mou, and L. Zhang, "Treegen: A tree-based transformer architecture for code generation," in *Proceedings of the AAAI conference on artificial intelligence*, vol. 34, no. 05, 2020, pp. 8984–8991.

[15] E. Nijkamp, B. Pang, H. Hayashi, L. Tu, H. Wang, Y. Zhou, S. Savarese, and C. Xiong, "Codegen: An open large language model for code with multi-turn program synthesis," *arXiv preprint arXiv:2203.13474*, 2022.

[16] J. P. Inala, C. Wang, M. Yang, A. Codas, M. Encarnación, S. Lahiri, M. Musuvathi, and J. Gao, "Fault-aware neural code rankers," *Advances in Neural Information Processing Systems*, vol. 35, pp. 13 419–13 432, 2022.

[17] B. Roziere, J. Gehring, F. Gloeckle, S. Sootla, I. Gat, X. E. Tan, Y. Adi, J. Liu, T. Remez, J. Rapin *et al.*, "Code llama: Open foundation models for code," *arXiv preprint arXiv:2308.12950*, 2023.

[18] M. L. Siddiq and J. C. Santos, "Securityeval dataset: mining vulnerability examples to evaluate machine learning-based code generation techniques," in *Proceedings of the 1st International Workshop on Mining Software Repositories Applications for Privacy and Security*, 2022, pp. 29–33.

[19] J. Wang, L. Cao, X. Luo, Z. Zhou, J. Xie, A. Jatowt, and Y. Cai, "Enhancing large language models for secure code generation: A dataset-driven study on vulnerability mitigation," *arXiv preprint arXiv:2310.16263*, 2023.

[20] J. He, M. Vero, G. Krasnopolska, and M. Vechev, "Instruction tuning for secure code generation," *arXiv preprint arXiv:2402.09497*, 2024.

[21] T. Szabó, "Incrementalizing production codeql analyses," in *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2023, pp. 1716–1726.

Authorized licensed use limited to: IEEE Xplore. Downloaded on March 09,2025 at 03:54:44 UTC from IEEE Xplore. Restrictions apply.