



Celebrating Signal Processing

# 2025 IEEE INTERNATIONAL CONFERENCE ON ACOUSTICS, SPEECH, AND SIGNAL PROCESSING (ICASSP 2025)

**April 06 – 11, 2025** **Hyderabad, India**



## **RefleXGen: The unexamined code is not worth using**

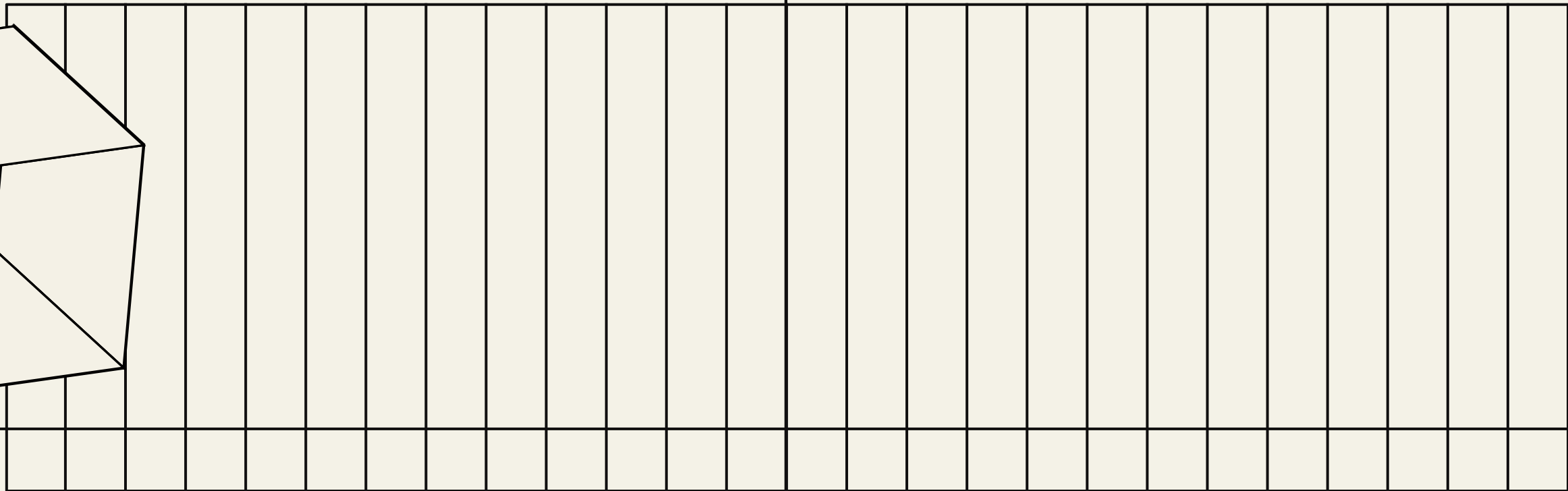

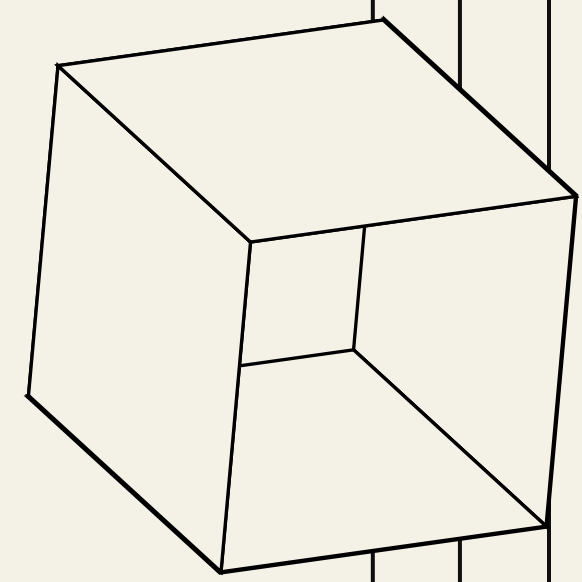
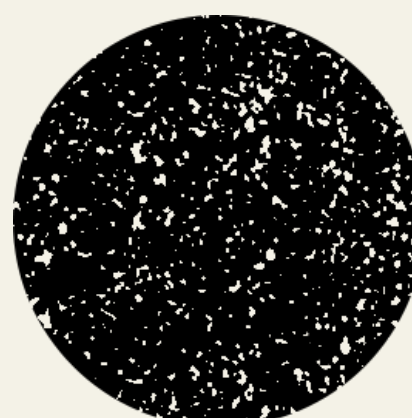
Bin Wang, Hui Li, AoFan Liu, BoTao Yang, Ao Yang, YiLu Zhong, Weixiang Huang, Runhuai Huang, Weimin Zeng, Yanping Zhang





# **ReflexGen: The unexamined code is not worth using**

Bin Wang, Hui Li, AoFan Liu, BoTao Yang, Ao  
Yang, YiLu Zhong, Weixiang Huang, Runhuai  
Huang, Weimin Zeng, Yanping Zhang



# Research Background

## RefleXGen: The Unexamined Code Is Not Worth Using

Bin Wang, Hui Li, AoFan Liu, BoTao Yang, Ao Yang, YiLu Zhong, Weixiang Huang,  
Runhuai Huang, Weimin Zeng, Yanping Zhang

Problem	LLM-generated code often contains security vulnerabilities due to unsafe training data.
Existing Solution	Fine-tuning LLMs or creating secure datasets → Resource-intensive. Limited focus on iterative self-improvement.
RefleXGen	Integrates <b>RAG (Retrieval-Augmented Generation)</b> with <b>self-reflection</b> mechanisms.
Advantage	<b>Iteratively optimizes code security without fine-tuning.</b>
Key Terms	Code Security, LLMs, RAG, Self-Reflection, CodeQL

Enhancing Code Security via **RAG** and **Self-Reflection**

**Affiliation:** Peking University

# Related Work

## RefleXGen: The Unexamined Code Is Not Worth Using

Bin Wang, Hui Li, AoFan Liu, BoTao Yang, Ao Yang, YiLu Zhong, Weixiang Huang, Runhuai Huang, Weimin Zeng, Yanping Zhang

### Related LLM Research

#### Traditional Rule-Based Methods

Relied on heuristic rules or expert systems to map requirements to code.

Inflexible and unscalable due to rigid rule sets. Limited to narrow domains (e.g., template-based code generation).

#### Early Deep Learning Approaches

Trained on code corpora to predict code sequences (e.g., [2], [3]). Advancements: Enabled mapping natural language descriptions to code snippets.

**However, these are not so effective!**

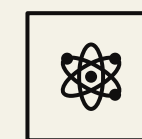
##### ☐ Limitaion

Ceres is located in the main asteroid belt

##### ☒ SATURN

Saturn is a gas giant and has several rings

#### Transformer-Based LLMs



Codex



CodeGen



CodeT5

#### Code Generation Security

☐ Secure Training Practices:  
StarCoder [5] and CodeLlama [17]:

☐ Filtered training data to exclude vulnerable code snippets.

☐ Added security-focused pretraining objectives.

☐ Limitation:

Security guarantees depend on data quality, which is hard to ensure at scale.



# RefleXGen's Innovation

## Core Idea:

- 1. Self-Reflection:** Guides LLMs to iteratively critique and repair code.
- 2. RAG Integration:** Dynamically retrieves secure coding practices and historical fixes.

## Related Work

There are various code generation models based on the Transformer architecture, such as Codex, CodeGen, CodeT5, etc., as well as research on evaluating and improving code security, such as StarCoder, CodeLlama, SafeCoder, etc.



WE RECOMMEND  
**RefleGen!**

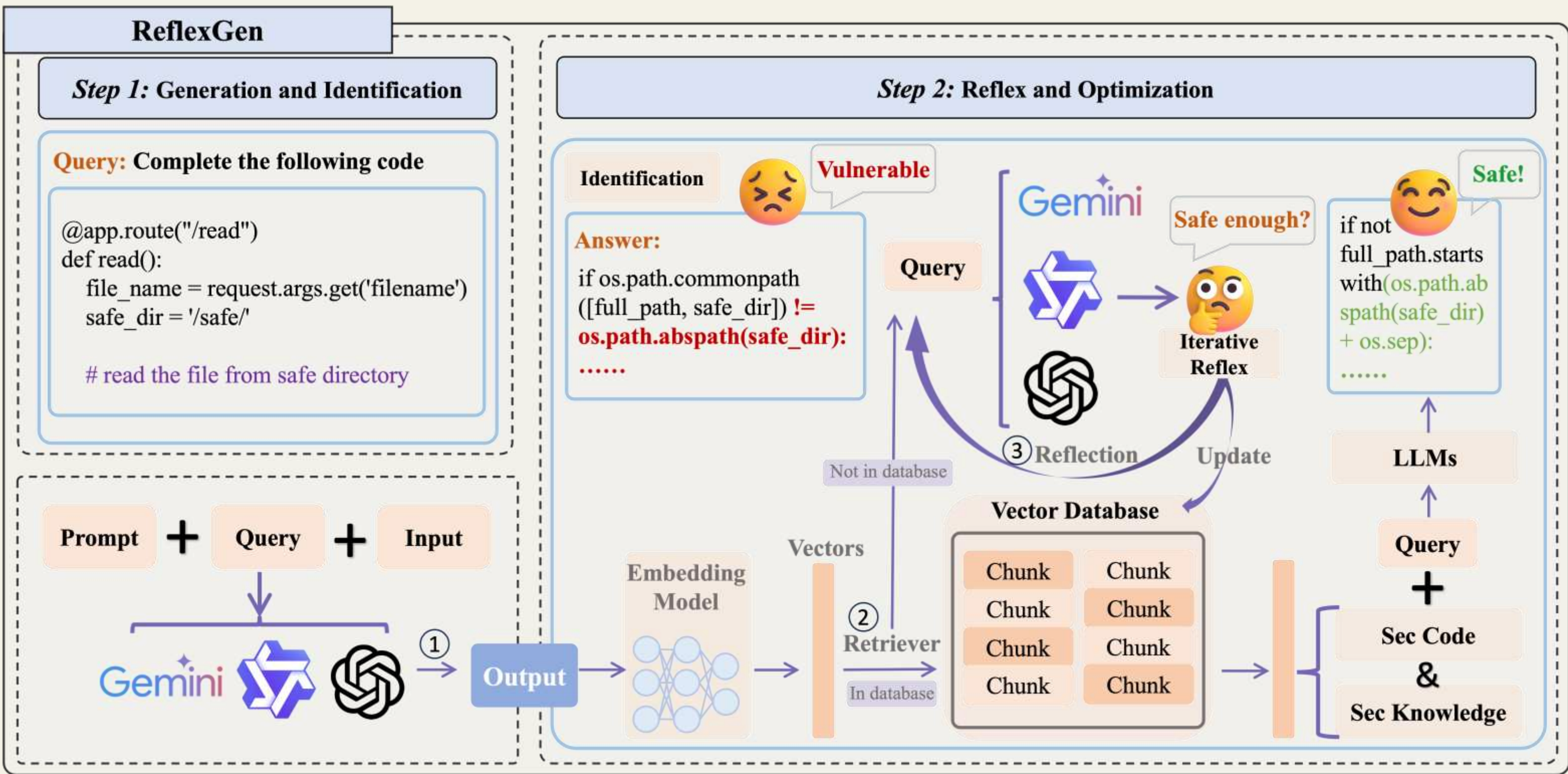


# Methodology

## RefleXGen: The Unexamined Code Is Not Worth Using

Bin Wang, Hui Li, AoFan Liu, BoTao Yang, Ao Yang, YiLu Zhong, Weixiang Huang, Runhuai Huang, Weimin Zeng, Yanping Zhang

### Architecture



### Method

RefleXGen combines self-refinement and RAG to improve LLM-generated code security without model fine-tuning. Its workflow follows a two-phase process:

**Phase 1:** Generate initial code from user requirements.

**Phase 2:** Iteratively optimize code through security-focused reflection.

- Validated secure code updates a dynamic safety knowledge base.
- Future tasks leverage accumulated insights for enhanced security.

**Key advantages:** No training overhead, adaptive knowledge integration, and broad model compatibility.

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.



# Methodology

## Detailed Steps

### Step 1: Initial Code Gen

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

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

### Step 2: 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.

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

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

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.

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

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.

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

# Experiment

## Experiment Settings

### 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.

### Dataset

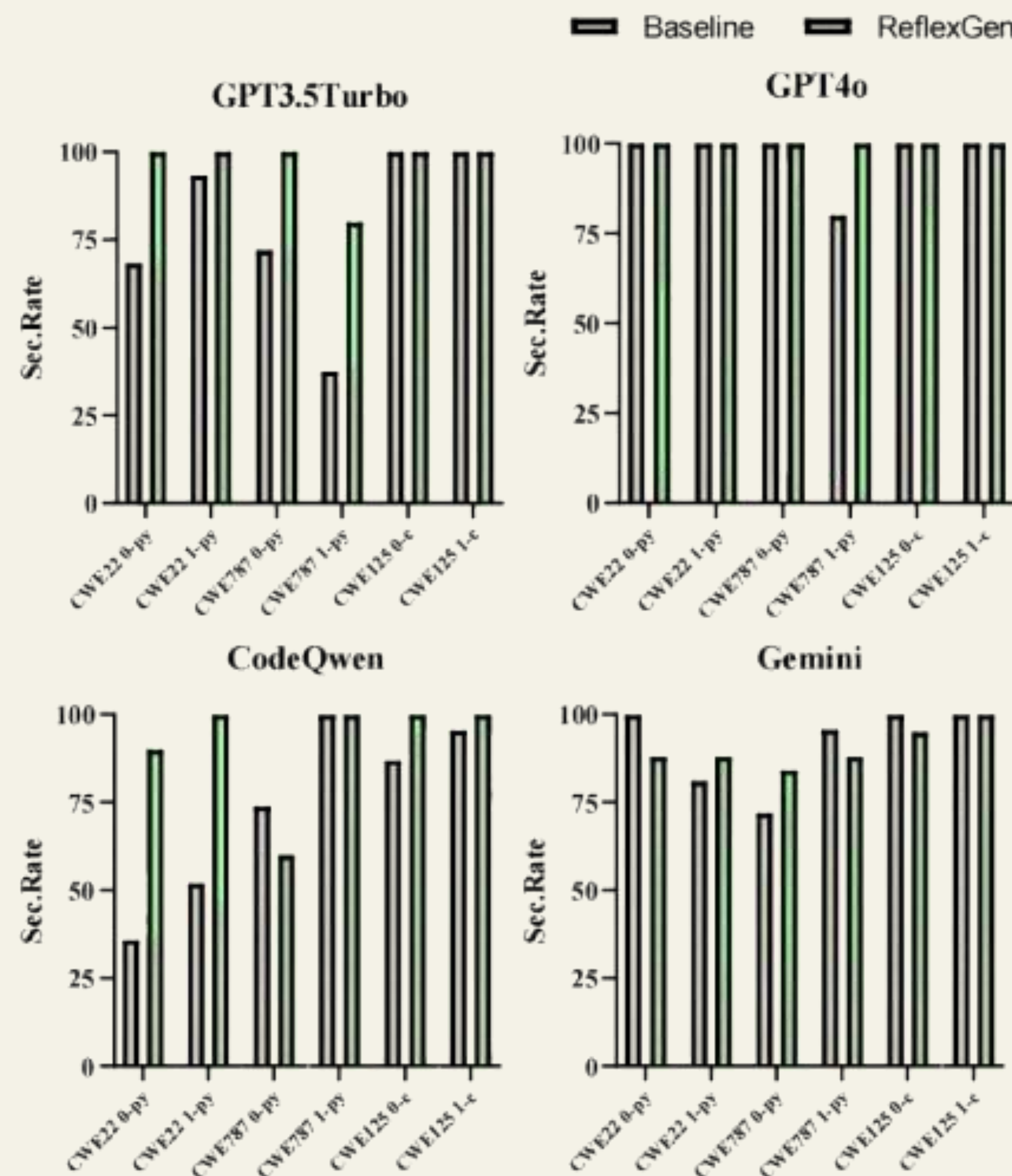
To evaluate **RefleXGen's** improvements in code generation security and reliability, we selected challenging scenarios from the most impactful **Common Weakness Enumerations** (CWEs). We used a **dataset validated by He et al.**, featuring nine scenarios from MITRE's top 25 most dangerous software vulnerabilities.

Model	Sec.Rate	Pass.Rate	Eff.Total	Sec.Count	Unres.Count
GPT3.5Turbo (Base)	75.5	97.6	24.5	19.5	0.5
GPT3.5Turbo (+RefleXGen)	89.1	95.8	24.0	22.3	1.1
GPT4o (Base)	92.3	94.2	23.6	21.9	1.4
GPT4o (+RefleXGen)	99.0	100.0	25.0	24.7	0.0
CodeQwen1.5 (Base)	83.7	86.7	21.6	17.9	3.3
CodeQwen1.5 (+RefleXGen)	88.2	69.8	20.4	19.4	3.8
Gemini1.0Pro (Base)	80.2	92.2	23.1	19.1	1.9
Gemini1.0Pro (+RefleXGen)	86.0	83.6	22.8	21.2	2.1



# Experiment

## Experiment Result



To ensure a fair comparison, we initially set the RAG content to empty, allowing **RefleXGen** to progressively generate content during testing.

We tracked several metrics: **Sec. Rate** , **Pass Rate**, **Eff. Total**, **Sec. Count**, **Unres. Count**. To obtain reliable data, we conducted five repeated experiments for each model. **CodeQL** was utilized to perform security analysis and assessment.

In each experiment, every scenario was subjected to **25 task** generations to average the results, ensuring an objective assessment of each model's generative capabilities.

# Conclusion

## RefleXGen: The Unexamined Code Is Not Worth Using

Bin Wang, Hui Li, AoFan Liu, BoTao Yang, Ao Yang, YiLu Zhong, Weixiang Huang, Runhuai Huang, Weimin Zeng, Yanping Zhang

### What we have done...

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.

### Not done...

- ☐ Optimize real-time performance: reduce the number of iterations or introduce parallelization strategies.
- ☐ Expand language support: adapt security rules of languages such as Rust and JavaScript.
- ☐ Dynamic analysis integration: integrate fuzzing or symbolic execution.
- ☐ Pre-trained security knowledge base: provide a pre-built security code base to accelerate cold start.
- ☐ Model capability enhancement: design lightweight fine-tuning or prompt engineering techniques for security tasks.



## Reference

- [1] Xin-Ye Li, Jiang-Tian Xue, Zheng Xie, and Ming Li, “Think outside the code: Brainstorming boosts large language models in code generation,” arXiv preprint arXiv:2305.10679, 2023.
- [2] Wang Ling, Edward Grefenstette, Karl Moritz Hermann, Tomáš Kočíšek, Andrew Senior, Fumin Wang, and Phil Blunsom, “Latent predictor networks for code generation,” arXiv preprint arXiv:1603.06744, 2016.
- [3] Veselin Raychev, Pavol Bielik, and Martin 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.
- [5] Anton Lozhkov, Raymond Li, Loubna Ben Allal, Federico Cassano, Joel Lamy-Poirier, Nouamane Tazi, Ao Tang, Dmytro Pykhtar, Jiawei Liu, Yuxiang Wei, et al., “StarCoder 2 and the stack v2: The next generation,” arXiv preprint arXiv:2402.19173, 2024.
- [6] Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham, Hyung Won Chung, Charles Sutton, Sebastian Gehrmann, et al., “Palm: Scaling language modeling with pathways,” Journal of Machine Learning Research, vol. 24, no. 240, pp. 1–113, 2023.
- [7] Dongling Xiao, Han Zhang, Yukun Li, Yu Sun, Hao Tian, Hua Wu, and Haifeng Wang, “Ernie-gen: An enhanced multi-flow pre-training and fine-tuning framework for natural language generation,” arXiv preprint arXiv:2001.11314, 2020.
- [8] Yue Wang, Weishi Wang, Shafiq Joty, and Steven CHoi, “Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation,” arXiv preprint arXiv:2109.00859, 2021.
- [9] Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, et al., “Competition-level code generation with alphacode,” Science, vol. 378, no. 6624, pp. 1092–1097, 2022.
- [10] Priyan Vaithilingam, Tianyi Zhang, and Elena 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] Jingxuan He and Martin 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] Hammond Pearce, Baleegh Ahmad, Benjamin Tan, Brendan Dolan-Gavitt, and Ramesh 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] Cordell Green, “Application of theorem proving to problem solving,” in Readings in Artificial Intelligence, pp. 202–222. Elsevier, 1981.
- [14] Zeyu Sun, Qihao Zhu, Yingfei Xiong, Yican Sun, Lili Mou, and Lu Zhang, “TreeGen: A tree-based transformer architecture for code generation,” in Proceedings of the AAAI conference on artificial intelligence, 2020, vol. 34, pp. 8984–8991.
- [15] Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong, “Codegen: An open large language model for code with multi-turn program synthesis,” arXiv preprint arXiv:2203.13474, 2022.
- [16] Jeevana Priya Inala, Chenglong Wang, Mei Yang, Andres Codaş, Mark Encarnación, Shuvendu Lahiri, Madanlal Musuvathi, and Jianfeng Gao, “Fault-aware neural code rankers,” Advances in Neural Information Processing Systems, vol. 35, pp. 13419–13432, 2022.