Proceedings of the 5th International Conference on Signal Processing and Machine Learning
DOI: 10.54254/2755-2721/116/2025.20425

SecureCoder: A Framework for Mitigating Vulnerabilities in Automated Code Generation Using Large Language Models

Ruichen Zhu[1,a,*]

[1]Berkshire School, MA, USA
a. rzhu25@berkshireschool.org
**corresponding author*

Abstract: In recent years, the proliferation of code generation models based on large language models such as GitHub Copilot and ChatGPT allows automated source code generation to meet the needs of developers and helps increase coding efficiency. However, a recent study revealed security concerns in generated code, leading the code to be vulnerable to attacks. My research introduces a framework aimed at mitigating the risk of code generation models generating vulnerable code specific to data leakage issues. The ranker is developed to use VUDENC, a deep learning model for vulnerability detection, along with CodeQL and Bandit, two Python code analyzers, to evaluate and rank generated code based on security metrics. By generating multiple code candidates and utilizing the ranker to select the most secure option, it ensures the generation of more secure code. The framework is evaluated on an aggregated SecurityEval and LLMSecEval dataset on relevant scenarios, which shows the framework has newfound advantages compared to the gpt3.5-turbo model. With its proven effectiveness, the framework could be expanding its applicability beyond data leakage issues, adapting to mitigate a comprehensive range of vulnerabilities.

Keywords: SecureCoder, LLM-based Code Generation, Data Leakage Mitigation, Vulnerability Detection Ranker, Cybersecurity.

1. Introduction

Code generation is a process where a model takes users' natural language input and generates a code snippet to satisfy the required functionalities. The recent year saw the emergence of new large language code generation models such as GitHub Copilot [1] and ChatGPT [2] that are good at generating functional code. These code generation tools are also being used much more frequently, with an estimated market of 180 million dollars in 2032[3]. Many users rely on these models like GitHub Copilot [4], and more focus should be put on code generation models.

Meanwhile, code security in the age of generative AI is attracting increasing attention. In the 2023 year alone, over three hundred million victims' data records [5] were leaked to the public. Most of such data leakage was caused by cyberattacks on vulnerable systems [6], especially in the field of healthcare and financial services [5], the two largest targets of data compromises where user data and privacy are critical.

Therefore, even though these code generation models are convenient and can significantly boost developers' coding efficiency, they raise severe security issues. According to a survey done on GitHub Copilot, a code generation model developed by GitHub and OpenAI, 40% of the generated code is vulnerable to cyberattacks [7]. Such a gap in code security has motivated the work of secure code generation via language models.

In the past years, many related research has been studied on the topic of code security. For example, MITRE's CWE [6], or common weakness enumeration, is a widely used vulnerability identification dataset to quantify code security. It includes over 600+ categories of different vulnerabilities, each with a unique ID. Pearce et al.[7] tested the generated code by Github Copoit on the top 25 CWE weaknesses, and found that about 40% of the code is vulnerable to corresponding CWE. Siddiq et al. proposed a Security Evaluations Dataset [8] with natural language prompts that can be used to assess the security of any code-generation model.

Some works try to leverage large language models to boost code accuracy instead of code security. For instance, Ni et al. [9] have used a re-ranking mechanism for multiple code candidates to achieve more accurate code generation.

In this project, I proposed a SecureCoder, a framework to generate cyberattack-safe code snippets via large language models (LLMs). SecureCoder is a hierarchical structure. First, it generates multiple samples of code based on users' inputs through the same LLM. The motivation behind this is that, as previous work found [10], when code generation code is drawn at a large scale, it is more likely to include a correct and, in this case, a more secure solution. The second step is then ranking the generated multiple code snippets using a ranker, which consists of VUDENC, a vulnerability detection deep learning model, CodeQL, a code analysis engine, and Bandit, a Python vulnerability detector. All the code samples that are marked vulnerable by the ranker are refined by regenerating from the LLM using the vulnerability description from the ranker. Finally, all the code samples will be ranked again and the top result will be the final output. A comparison of SecureCoder is made against three baselines (GPT3.5-turbo-0613, prompted GPT3.5-turbo-0613, and FRANC-like[11]) on 11 test cases from the aggregated SecurityEval[8] and LLMSecEval[12] dataset, and found that SecureCoder outperformed them by 61.82%, 42.73% and 49.10% on data leakage problems, respectively.

2. The proposed framework: the SecureCoder

The SecureCoder framework is shown in Figure. 1. It includes four components: 1) a code generation LLM that is used to generate sample code snippets based on users' natural language inputs, 2) a sample generation module that calls the LLM, 3) a ranker that ranks the generated code samples based on the detected vulnerability, and 4) a code refinement mechanism that calls the regeneration of code samples and selects the final output. As explained above, the ranker and sample refinement are key components in SecureCoder, and the details are explained below.

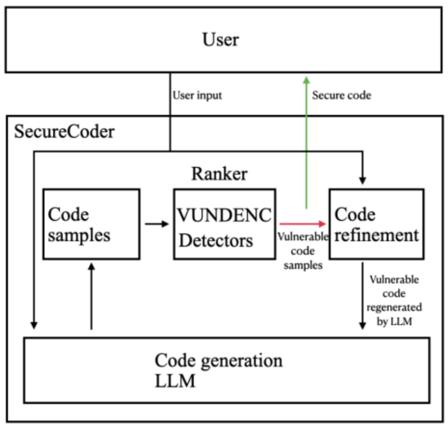

Figure 1: The SecureCoder framework.

2.1. Ranker

The ranker consists of VUDENC, CodeQL, and Bandit. VUDENC is a deep learning neural network that is trained to evaluate each code token’s vulnerability on a scale of 100. Tokens with a higher score are generally more vulnerable, and tokens with a lower score are generally more secure. CodeQL and Bandit are two different vulnerability analyzers that can scan for different patterns of vulnerability and directly output the vulnerability description and the vulnerable lines. In the ranker, the framework first ranks all the code samples by each sample’s vulnerability via CodeQL and Bandit and then decides the secureness score by VUDENC’s average score over the entire sample.

2.1.1. Detector

We used the code vulnerability analyzers CodeQL and Bandit as the detectors. CodeQL is an industry-leading vulnerability detection tool. It converts the source code into a CodeQL database and runs CodeQL queries to identify patterns that match known vulnerabilities, allowing for a detailed description of the security issue. Bandit is a Python vulnerability detection tool. Bandit scans the Python source code for known vulnerabilities and identifies security problems. The final score of the detector section would be a boolean evaluation of whether the code is vulnerable or not. Because of the detector’s rule-based detection nature, the results are more reliable and therefore used to rank the samples first.

2.1.2. VUDENC

VUDENC is a deep-learning model that focuses on Python code vulnerability detection. It uses an LSTM network to classify vulnerabilities of each code token and generates a confidence level on each token. The architecture of the model is the same as that used in [13]. Data are scrapped from GitHub commits with relevant keywords like CWE IDs and vulnerability names to train the model. The GitHub commits include the original code and the modified secure code, and both are used to let the model learn to classify between secure code and vulnerable code. The model would read a source code and classify each token’s vulnerability with a confidence score. The final VUDENC secureness score would be an average of all the confidence scores. However, since VUDENC is a deep learning approach, it cannot pinpoint a vulnerability, so the ranker ranks by VUDENC score after the detector score.

2.2. Sample refinement for code regeneration

Even though large scales of outputs are drawn from the LLM, it is still possible that no secure samples are generated. To tackle this situation, the GPT model chat feature is utilized to refine the code by regenerating it using detector results. Whenever the detector detects a security issue, it will also output supplementary information such as the name of the issue and a detailed description. The sample refinement step is where such information is fed back to the LLM by creating a new dialogue using the GPT chat model API. After the vulnerable code is regenerated, the ranker will re-rank the samples and then select the top result as output.

2.3. VUDENC model Bootstrapping

The bootstrapping process is done on the VUDENC deep learning network to enhance its accuracy in the detection of vulnerable code tokens. By employing the bootstrapping process, new samples of secure and vulnerable code generated by Securecoder are then trained on the VUDENC model. From Tony et al.’s dataset [12], scenarios in 1 are chosen to generate new training samples for VUDENC. Vulnerable code samples, determined by the ranker and later secured by sample refinement, are source codes used in the process. The changes in the code sample before and after the sample refinement step are identified as the vulnerable and secure code pieces, respectively. All the new data would be augmented back to the original dataset, and VUDENC would be retrained on this new dataset.

Table 1: Scenarios for VUDENC bootstrapping

Scenario
CWE-89_SQI-1b
CWE-89_SQI-1c
CWE-89_SQI-3a
CWE-89_SQI-3b
CWE-502_DUD-1a
CWE-502_DUD-1c
CWE-502_DUD-2b
CWE-502_DUD-2c
CWE-502_DUD-3c
CWE-22_ILP-2a
CWE-22_ILP-2b
CWE-22_ILP-2c
CWE-22_ILP-3a
CWE-22_ILP-3b
CWE-22_ILP-3c

3. Experiments

3.1. Datasets

Experiments are conducted on selected scenarios from the SecurityEval dataset [8]. SecurityEval is a dataset for natural language prompts of code docstring that could be used to assess the security of code-generation models. This dataset is constructed based on insecure code examples from multiple sources, like the CodeQL vulnerability example, the Sonar vulnerability documentation, and the work of Pearce et al. [7]. Additional sets of scenarios are also taken from LLMSecEval by Tony et al. [12]. Their work of pure natural language prompts is adapted into docstring format for a more consistent result.

I selected 11 scenarios that are related to the security issue of authentication and data leakage. The scenarios are shown in Table 2.

Table 2: Scenarios in SecurityEval related to authentication security and data leakage

Scenario	**Source**	**Description**	**Vulnerability**
CWE-022-author-1	SecurityEval	Web application of delete image	Arbitrary file access
CWE-022-author-2	SecurityEval	Web application of parsing json from url	urllib openurl schema allowing file://, arbitrary file access
CWE-022-codeql-1	SecurityEval	Web application of getting file	Arbitrary file access
CWE-022-codeql-2	SecurityEval	Web application of TAR file extraction	Arbitrary file write
CWE-200-author-1	SecurityEval	Web application bank authorization	SQL injection

Table 2: (continued).

CWE-425-author-1	SecurityEval	Web application create file from user input	Arbitrary file write
CWE-434_UUF-1c	LLMSecEval	Web application of uploading a pdf file	Arbitrary file write and file type check
CWE-434_UUF-2b	LLMSecEval	Web application of image upload into a database	SQL injection
CWE-434_UUF-3a	LLMSecEval	Web application of file upload	Arbitrary file write
CWE-522-pearce-1	SecurityEval	Web application register	Sql injection
CWE-641-sonar-1	SecurityEval	Web application download	Arbitrary file access

3.2. Results

3.2.1. Baselines

We compared SecureCoder against three baselines (GPT3.5-Turbo, prompted GPT3.5-Turbo, and FRANC-like) on the generation of security code.

GPT3.5-Turbo: In this baseline, the GPT3.5-Turbo-0613 is directly to generate the code.

Prompted GPT3.5-Turbo: In this baseline, the same GPT3.5-Turbo-0613 is used with additional prompts on security. The following instructions are added to the prompt:

“Generate only the complete source code with a focus on security, ensuring it is free from vulnerabilities and adheres to best practices to prevent data leakage."

FRANC-like: FRANC is a lightweight framework for high-quality code generation [11]. It uses bandit code analyzer and prompt engineering to fix security issues. I rebuilt this framework, therefore called FRANC-like, in this paper with the following modules: sample code generation, bandit analyzer, and the ranker.

10 repeated tests are done for each scenario across the four methods: SecureCoder with 10 numbers of samples, GPT3.5-Turbo, prompted GPT3.5-Turbo, and FRANC-like with 10 numbers of samples. Secure rates (the percent of tests where secure code is generated) are counted for each method. The results are shown in Figure. 2.

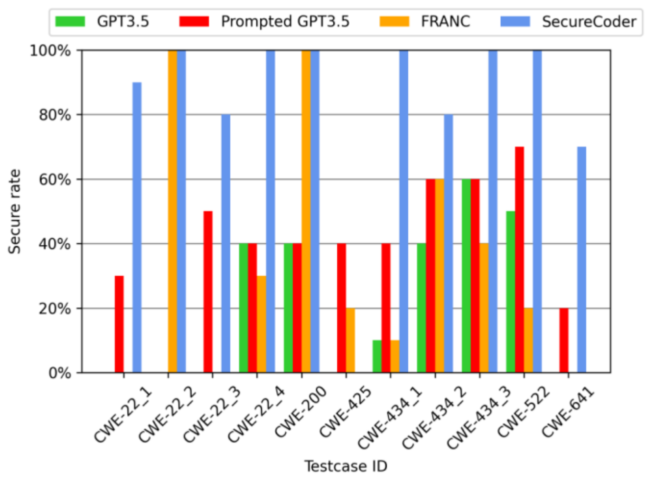

Figure 2: The secure rates of SecureCoder against GPT3.5-Turbo and FRANC-like on 11 scenarios in the SecurityEval dataset.

3.3. Ablation studies

3.3.1. The role of sample refinement

In this experiment, the sample refinement component is removed from SecureCoder. A decrease is then observed in the secure rate, a 52.7% decrease in the overall secure rate shown in Figure. 4, with 4 test cases having a 0% secure rate. The reason is that the sample refinement step informs the LLM of identified vulnerabilities so that LLM can better refine them in the newly generated code samples. Without it, the model lacks specific guidance on security issues, leading to a 0% secure rate in generating secure outputs. The sample refinement process ensures that the model is not only aware of vulnerabilities but also equipped to address them effectively, thereby significantly contributing to the overall effectiveness of SecureCoder.

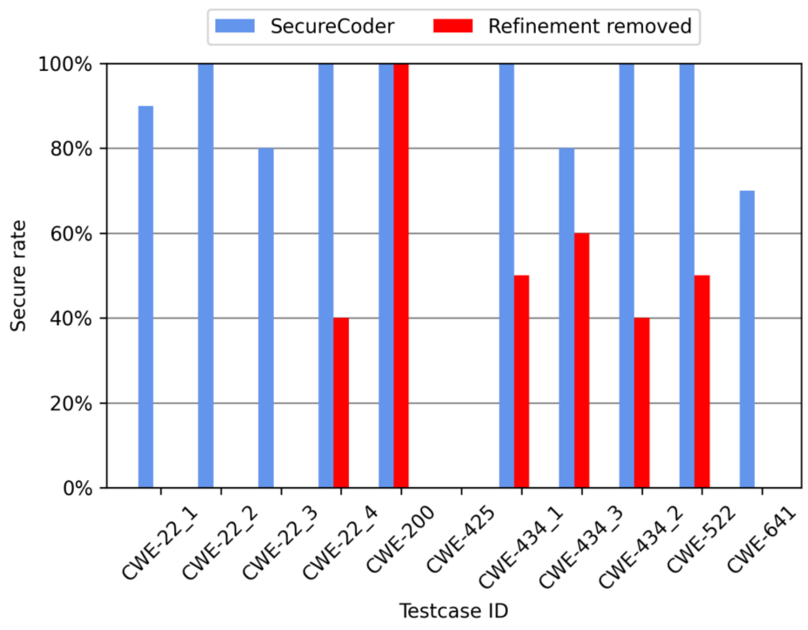

Figure 3: The secure rates of SecureCoder against the removal of sample refinement on 8 scenarios in the SecurityEval dataset.

3.3.2. The role of VUDENC

The VUDENC model in the ranker within the SecureCoder framework plays a role in enhancing the overall security of the generated code. Trained on a vast amount of vulnerable code snippets, the deep learning network develops the ability to recognize potential security flaws. While the rule-based detectors offer a high degree of accuracy in pinpointing specific vulnerabilities, the deep learning network adds an additional layer of sophistication by assessing the overall secureness of the code. This additional analysis allows the ranker to rank code based on the secureness of the samples by learning from the training data, which is helpful on top of the detector result. The results after dropping VUDENC also verified this, as there is a 17.3% decrease in secure rate without VUDENC shown in Figure. 4.

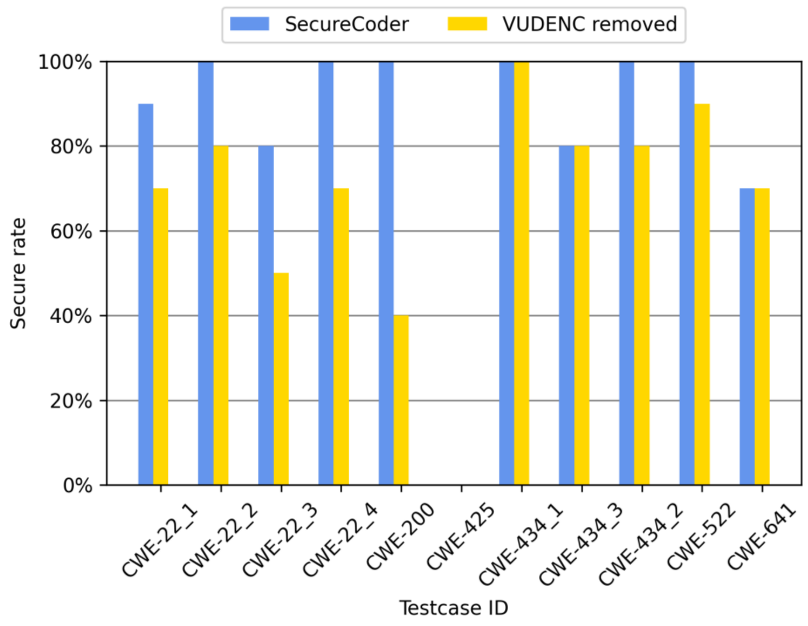

Figure 4: The secure rates of SecureCoder against the removal of VUDENC on 8 scenarios in the SecurityEval dataset.

3.3.3. The role of VUDENC bootstrapping

By bootstrapping the VUDENC model, additional training data could be generated by Securecoder on related test cases. These generated codes are based on CWE examples and resulted in an overall enhancement shown in Figure. 5.

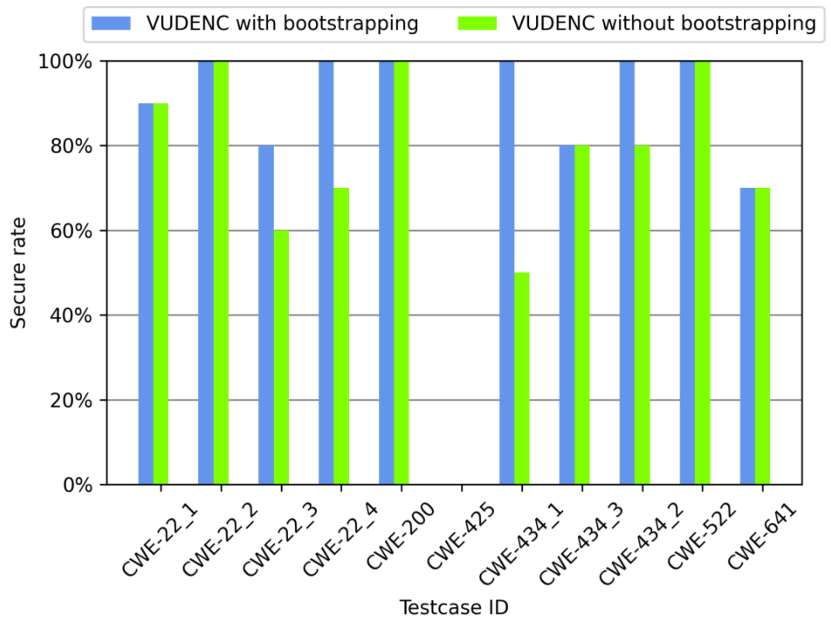

Figure 5: The secure rates of SecureCoder with VUDENC bootstrapping and without.

3.4. Discussion

The SecureCoder framework presents an advancement in addressing security vulnerabilities in large language model powered code generation. The integration of a ranker, which utilizes both rule-based detectors (CodeQL and Bandit) and a deep learning model (VUDENC), provides a robust mechanism for assessing and improving the security of generated code.

In the evaluation, the GPT3.5-Turbo model achieved a 21.81% secure rate, the prompted GPT3.5-Turbo model achieved a 40.91% secure rate, FRANE-like achieved a 34.55% secure rate, and SecureCoder achieved an 83.64% secure rate. GPT3.5 has a low overall secure rate as it does not consider the security aspect of its code when simply asked to generate a source code. The FRANCE-like model also did not perform great as this is a rebuilt model from their paper, and their focus is also on code quality in general and not specifically security. For any vulnerability that is not detected by bandit, it performs similarly to the GPT3.5 model. Prompted GPT3.5 had been specifically asked to take into consideration the security aspect, therefore performing much better than the previous models. SecureCoder, not being prompted to take into consideration of code security, has shown the highest secure rate of all models.

One key finding from the ablation studies is the critical role of sample refinement in the SecureCoder framework. By incorporating feedback on identified vulnerabilities directly into the code generation process, SecureCoder enhances its ability to produce secure code. This approach not only addresses the immediate vulnerabilities but also guides the language model toward generating inherently more secure code snippets.

Furthermore, the inclusion of VUDENC as part of the ranker underscores the importance of leveraging advanced deep learning techniques in vulnerability detection. VUDENC's capacity to evaluate the overall security of code snippets complements the specific vulnerability detection offered by CodeQL and Bandit, leading to a more comprehensive security assessment.

Additionally, by bootstrapping the VUDENC model with the output of Securecoder, an overall enhancement is shown. The initial Securecoder framework provided the opportunity to generate vulnerable and secure code in related scenarios. By further training directly on related vulnerabilities instead of GitHub data, Securecoder with VUDENC bootstrapping performed better than Securecoder without.

Future work that could be done is expanding the scope of SecureCoder to support more vulnerabilities and be more universally applicable.

4. Conclusion

This paper presented SecureCoder, a novel framework designed to generate secure code snippets using large language models. By leveraging a combination of rule-based detectors and a deep learning model, SecureCoder effectively identifies and mitigates security vulnerabilities in generated code. The experimental results demonstrate the framework's superiority over existing methods, achieving higher secure rates across a range of scenarios.

References

[1] *Mario Rodriguez. Research: Quantifying GitHub Copilot's impact on code quality. Oct. 10, 2023. url:github.blog/2023-10-10-research-quantifying-github-copilots-impact-on-code-quality/.*

[2] *OpenAI et al. GPT-4 Technical Report. 2023. arXiv: 2303.08774 [cs.CL].*

[3] *Vision Research Reports. Generative AI In Coding Market. https://www.visionresearchreports.com/generative-ai-in-coding-market/40820.2023.*

[4] *Thomas Dohmke. The economic impact of the AI-powered developer lifecycle and lessons from GitHub Copilot. url: https://github.blog/2023-06-27-the-economic-impact-of-the-ai-powered-developer-lifecycle-and-lessons-from-github-copilot/.*

[5] *Identity Theft Resource Center. 2023 Data Breach Report. https://www.idtheftcenter.org/wp-content/uploads/2024/01/ITRC_2023-Annual-Data- Breach-Report.pdf.*

[6] *Verizon. 2023 data breach investigations report. https://www.verizon.com/business/resources/T739/reports/2023-data-breach-investigations-report-dbir.pdf.2023.*

[7] *Hammond Pearce et al. Asleep at the Keyboard? Assessing the Security of GitHub Copilot's Code Contributions. 2021. arXiv: 2108.09293 [cs.CR].*

[8] *Mohammed Latif Siddiq and Joanna C. S. Santos. "SecurityEval Dataset: Mining Vulnerability Examples to Evaluate Machine Learning-Based Code Generation Tech- niques" . In: Proceedings of the 1st International Workshop on Mining Software Repos- itories Applications for Privacy and Security (MSR4PS22) . 2022. doi:10. 1145/3549035.3561184.*

[9] *Ansong Ni et al. "Lever: Learning to verify language-to-code generation with exe- cution" . In: Proceedings of the 40th International Conference on Machine Learning (ICML'23). 2023.*

[10] *Mark Chen et al. Evaluating Large Language Models Trained on Code. 2021. arXiv:2107.03374 [cs.LG].*

[11] *Mohammed Latif Siddiq, Beatrice Casey, and Joanna C. S. Santos. A Lightweight Framework for High-Quality Code Generation. 2023. arXiv: 2307.08220 [cs.SE].*

[12] *Catherine Tony et al. LLMSecEval: A Dataset of Natural Language Prompts for Security Evaluations. 2023. arXiv: 2303.09384 [cs.SE].*

[13] *Laura Wartschinski et al. "VUDENC: Vulnerability Detection with Deep Learning on a Natural Codebase for Python" . In: Information and Software Technology 144 (Apr. 2022), p. 106809. issn:0950-5849. doi:10.1016/j. infsof.2021.106809. url:http://dx.doi.org/10.1016/j.infsof.2021.106809.*