Demo: SGCode: A Flexible Prompt-Optimizing System for Secure Generation of Code

Khiem Ton*
Nhi Nguyen*
kt477@njit.edu
yennhi1908hcm@gmail.com
New Jersey Institute of Technology
Newark, New Jersey, USA

Mahmoud Nazzal
Abdallah Khreishah
mn69@njit.edu
abdallah@njit.edu
New Jersey Institute of Technology
Newark, New Jersey, USA

Cristian Borcea
NhatHai Phan*
borcea@njit.edu
phan@njit.edu
New Jersey Institute of Technology
Newark, New Jersey, USA

Ruoming Jin
Kent State University
Kent, Ohio, USA
rjin1@kent.edu

Issa Khalil
Qatar Computing Research Institute
Doha, Qatar
ikhalil@hbku.edu.qa

Yelong Shen
Microsoft Azure AI
Redmond, Washington, USA
yelong.shen@microsoft.com

Abstract

This paper introduces SGCode, a flexible prompt-optimizing system to generate secure code with large language models (LLMs). SGCode integrates recent prompt-optimization approaches with LLMs in a unified system accessible through front-end and back-end APIs, enabling users to 1) generate secure code, which is free of vulnerabilities, 2) review and share security analysis, and 3) easily switch from one prompt optimization approach to another, while providing insights on model and system performance. We populated SGCode on an AWS server with PromSec, an approach that optimizes prompts by combining an LLM and security tools with a lightweight generative adversarial graph neural network to detect and fix security vulnerabilities in the generated code. Extensive experiments show that SGCode is practical as a public tool to gain insights into the trade-offs between model utility, secure code generation, and system cost. SGCode has only a marginal cost compared with prompting LLMs. SGCode is available at: **SGCode**.

CCS Concepts

• **Security and privacy** → **Software security engineering**; *Vulnerability management*; • **Computing methodologies** → **Natural language generation**.

Keywords

Demonstration system, Prompt optimization, Secure code, LLMs

*Corresponding author; Khiem Ton and Nhi Nguyen are co-first authors.

1 Introduction

Ground-breaking developments of LLM-powered code generation commercial tools, such as Microsoft GitHub Copilot and Amazon CodeWhisperer, have significantly improved software developer productivity [3]. However; recent studies show that these tools frequently inherit security flaws from their open-source training data, leading to vulnerabilities such as common weakness enumerations (CWEs) in the generated code [1, 7, 9]. In real-world systems, attackers can exploit such code vulnerabilities, potentially leading to cyberattacks, data breaches, and degraded performance. Therefore, there is an urgent need for reliable approaches and systems to identify and fix vulnerabilities in LLMs-powered code generation.

Recent approaches, including PromSec [6] and SafeCoder [4], have shown great potential in optimizing prompts toward generating secure, free of vulnerabilities code. Yet, the lack of a flexible, deployable system limits our understanding of the trade-offs between secure prompt optimization, code utility, security analysis, and system performance. To bridge this gap, we propose SGCode, a flexible system for deploying and evaluating various prompt optimization methods. This tool enables users to gain insights into the trade-offs between code utility, security analysis, and system performance, ultimately aiding in the development of more secure approaches while minimizing system costs. For example, analyzing different CWEs and their associated costs in computation, communication, and prompting LLMs allows us to refine our model to better address these vulnerabilities and reduce overall costs.

Contributions. We develop SGCode with two integrated components: **(1) Back-end Services**, which seamlessly integrate **Code Security Analysis Tools**, such as Bandit [2] and CodeQL [5], and commercial LLMs, such as GPT-4o, to generate optimal prompts for secure code generation and send the results, including the code, security analysis, and system performance report, to the user front-end; **(2) User front-end**, a web-based interface that enables users to query prompt-optimizing systems, display code, view and share security analysis and performance reports. SGCode is deployed on

an AWS lightweight server, and extensive experiments show minimal system cost (CPU, memory, and latency) compared to the high cost of LLM code generation. In addition, SGCode allows users to easily switch prompt-optimizing approaches [4, 6]. SGCode offers a portable and lightweight solution to gain insights into the trade-offs between code utility, secure code generation, and system performance cost.

2 Prompt-Optimizing Secure Code Generation

Two recently developed secure code generation approaches are PromSec [6] and SafeCoder [4]. The key idea of PromSec is leveraging a graph generative adversarial neural network (gGAN) to mitigate code vulnerabilities by altering the code's graph representations. These altered representations are translated into prompt adjustments using LLMs. PromSec trains the gGAN model using a differentiable contrastive objective, which seamlessly integrates code security analysis tools, LLMs, and the gGAN's generator together, ensuring secure code generation.

An alternative solution is SafeCoder [4], an instruction tuning approach to resolve code vulnerabilities in two steps: (1) Curating a dataset consisting of standard instructions and a code vulnerability dataset consisting of GitHub commits for fixing code vulnerabilities; and (2) Instruction tuning that minimizes the language modeling loss $\mathcal{L}^{\text{std}}$ if the data point is in the instruction dataset; otherwise, the loss is $\mathcal{L}^{\text{sec}} + \mathcal{L}^{\text{vul}}$ if the data point is in the vulnerability dataset. $\mathcal{L}^{\text{sec}}$ is the likelihood loss that aligns the LLM model toward the secured tokens in o^{sec}, whereas $\mathcal{L}^{\text{vul}}$ is the unlikelihood loss that helps to reduce the probability of generating vulnerable tokens in o^{vul}. The fine-tuned LLM is used to generate secure code.

3 SGCode System

We develop SGCode (Figure 1a) as an API-accessible service and a web application for users to access our system on the web or integrate our API into their applications. The system design focuses on simplicity to optimize system costs, extensibility to customize the API, and maintainability with our modular architecture.

SGCode's back-end is powered by FastAPI [8]. It receives instructions to generate code from the users and then utilizes a pipeline consisting of three main components: (1) an interchangeable security analysis component, (2) a gGAN model from PromSec, and (3) an easily modifiable LLM. This design allows users to generate secure code with customization in mechanisms such as PromSec or SafeCoder. Users can use gGAN with an LLM for PromSec, or they can simply change the LLM to the SafeCoder-tuned LLM as in [4] to enable SafeCoder standalone or even combine the two approaches. Our system employs a NoSQL database to store the generated code for shareable security reports.

Before performing security analysis, the back-end processes the request from the front-end. If the payload instructs code generation, it is fed to the LLM component to produce the code. The generated code is then analyzed for vulnerabilities using Bandit or CodeQL.

If no vulnerability is found in the code or if the SafeCoder standalone is selected, the system returns the code to the user since the code is secure. Otherwise, the vulnerable code is sent to the gGAN module from PromSec for further processing and then passed to the LLM component.

The LLM plays a pivotal role in generating code that aligns with users' instructions and performs prompt optimization in PromSec. Our design allows users to modify the LLM by supporting any API servers that are OpenAI API-compatible. As a result, users can use our OpenAI model connections, connect via their personal OpenAI account, or implement self-hosted LLMs. This approach also helps facilitate the integration of SafeCoder into our system.

A FastAPI server manages all three modules, ensuring seamless operation of the pipeline. In addition to managing the modules, the server performs auxiliary tasks, such as saving the programs with their corresponding security analysis to a NoSQL (MongoDB) database and creating shareable reports. These tasks are background processes, thus do not impact the pipeline's performance.

3.1 Front-End Interface

The interface is built using Streamlit, an open-source Python framework for creating interactive user interfaces (UI). The interface is organized into two main screens: (1) AI Prompt UI, which facilitates user input (Figure 1b); and (2) Security Analysis Report screen, which displays detailed results and insights (Figure 1c).

AI Prompt UI. Users configure preferences for security analysis tools, code securing mechanisms, and LLMs. Once preferences are set, users can initiate their work by entering a prompt into the input box at the bottom of the screen. In addition, users can upload code directly using the code upload button. The submitted data is then sent to the back-end for processing. For every prompt for which the back-end service returns a response, the generated code from the LLM is displayed, and a button "Security Analysis" is provided to generate the code's security analysis report.

Security Analysis Report. The report page presents vulnerabilities in the code and provides data visualization. The right side of the screen features a container divided into tabs that distinguish between vulnerabilities in the original code (user input prompt) and those in the secured code (generated by SGCode). Each tab displays a detailed list of issue information. This comparison facilitates the identification of modifications made to address vulnerabilities. On the left side, the page includes a comprehensive visualization of issue confidence counts in the original code, which reflects the severity of vulnerabilities. A summary tab provides an overview of detected vulnerabilities, detailing the number of identified, fixed, and remaining issues, thus offering a rapid assessment of the code's security status. The page also features a line-by-line comparison of the original and secured code to illustrate the changes implemented and the effectiveness of the system's fixes. Users can share the security analysis as a PDF report or HTML link.

3.2 Deployment and Evaluation

We deploy SGCode on an AWS c7g.large virtual machine consisting of 2 vCPUs of AWS Graviton3 ARM processor and 4 GiB memory. We run the FastAPI with 4 workers and Streamlit on the same machine. Our back-end connects to a MongoDB instance hosted via MongoDB Atlas. We conduct three experiments using the test data in [6]: **(1)** Evaluating SGCode's resource usage with and without PromSec; **(2)** Measuring SGCode's latency given the number of CWEs, CWE IDs, and the prompt length; and **(3)** Inspecting the code security and code functionality using our Security Report.

First, we measure the resource usage by using the Python libraries psutil and sys. We select the back-end and front-end process
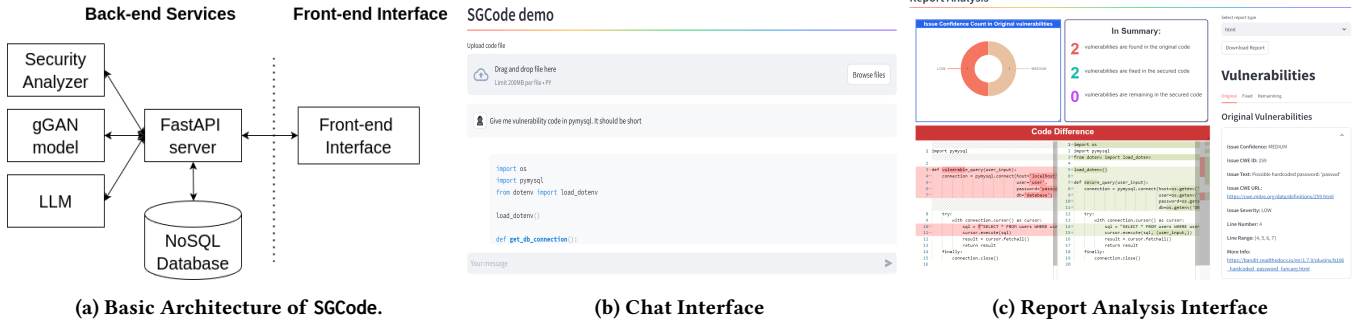
(a) Basic Architecture of SGCode.

(b) Chat Interface

(c) Report Analysis Interface

Figure 1: Front-end User Interface.

#Vulnerabilities	gGAN time	Security Analysis time	LLM time	Communication time	Total time	#Prompt Tokens	#Output Tokens	Time per Code line
2	0.01	0.97	40.28	0.62	41.89	426	355	0.66
3	0.007	0.78	23.17	0.59	24.56	310	264	0.53
4	0.004	0.52	11.19	0.56	12.28	318	217	0.34

Table 1: Performance metrics for different numbers of vulnerabilities (time is in seconds).

CWE ID	gGAN time	Security Analysis time	LLM time	Communication time	Total time	#Prompt Tokens	#Output Tokens	Time per Code line
20	0.015	1.46	55.14	0.60	57.21	365	379	1.04
78	0.007	0.77	22.30	0.58	23.65	310	256	0.52
89	0.008	0.78	25.06	0.59	26.43	357	294	0.52
259	0.008	0.79	25.52	0.59	26.92	339	279	0.54
327	0.005	0.58	13.26	0.58	14.43	313	228	0.37
703	0.002	0.40	6.59	0.59	7.58	236	207	0.22

Table 2: Performance metrics for different CWEs (time is in seconds).

Prompt Length	gGAN time	Security Analysis time	LLM time	Communication time	Total time	#Prompt Tokens	#Output Tokens	Time per Code line
LOW	0.006	0.63	15.58	0.57	16.79	282	231	0.41
HIGH	0.013	1.23	51.25	0.64	53.14	471	400	0.88

Table 3: Comparison of performance metrics for LOW and HIGH prompt length (time is in seconds).

	CPU (Percentage)	Memory (MB)
PromSec	0.06	2,170.75
No PromSec	0.05	2,168.84

Table 4: Resource usage comparison

IDs and their child processes and run the test dataset on the front end via Selenium. We evaluate our system until the test is finished and report the average result in three runs. Table 4 shows that utilizing PromSec has negligible CPU and memory usage.

Second, we measure the system's total latency and the latency of each component. We conduct the latency experiment on the vulnerabilities count, different CWEs, and prompt lengths based on the average number of tokens in the test dataset. In our dataset, we consider "LOW" to be less than and "HIGH" to be larger than the average number of tokens (i.e., 335.5). Tables 1-3 show that the gGAN and the Security Analyzing component are lightweight compared to the enormous latency introduced by the LLM component. Also, longer prompts impose longer latency.

Third, manually inspecting the code functionality reveals that utilizing a standalone gGAN with commercial LLMs, such as GPT3.5, GPT4, and GPT4o, can cause code functionality deviation, that is, users do not get secure code with desirable functionality given the information loss incurred by the iterative process of optimizing prompts with the LLMs. About 98% of the generated code has partially or fully deviated functionality. Flexibly replacing the LLMs with a SafeCoder-tuned LLM can potentially enhance the trade-off between code functionality and security. However, these models have been trained on relatively small datasets. Therefore, a more rigorous study is needed to investigate this trade-off and SGCode offers a practical tool for this purpose.

4 Conclusion and Future Works

We developed SGCode, a timely and flexible system that utilizes prompt-optimizing mechanisms to generate secure code. SGCode performs efficiently, even on a lightweight AWS virtual machine, with negligible overhead from prompt-optimizing mechanisms. Our future work will focus on: (1) Incorporating a feature that allows users to define utility tests and automatically test their code; (2) Further optimizing our system to reduce latency; and (3) Scaling the model to handle larger data and more complex code.

Acknowledgement

References

[1] Yiannis Charalambous, Norbert Tihanyi, Ridhi Jain, Youcheng Sun, Mohamed Amine Ferrag, and Lucas C Cordeiro. 2023. A New Era in Software Security: Towards Self-Healing Software via Large Language Models and Formal Verification. *arXiv preprint arXiv:2305.14752* (2023).
[2] B. Developers. 2022. *Welcome to Bandit — Bandit documentation.* https://bandit.readthedocs.io/en/latest/ [Online; accessed 1. June 2023].
[3] Github. 2023. Research: Quantifying GitHub Copilot's impact on code quality. https://github.com/features/copilot. [Online; accessed 17-June-2024].
[4] Jingxuan He, Mark Vero, Gabriela Krasnopolska, and Martin Vechev. 2024. Instruction Tuning for Secure Code Generation. In *ICML*.
[5] GitHub Inc. 2021. CodeQL Documentation. https://codeql.github.com/docs/ [Online; accessed 4-Dec-2023].
[6] Mahmoud Nazzal, Issa Khalil, Abdallah Khreishah, and NhatHai Phan. 2024. PromSec: Prompt Optimization for Secure Generation of Functional Source Code with Large Language Models (LLMs). In *ACM CCS*.
[7] 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 *IEEE S&P*. 754–768.
[8] Sebastián Ramírez. 2024. *FastAPI.* https://fastapi.tiangolo.com
[9] Mohammed Latif Siddiq, Beatrice Casey, and Joanna Santos. 2023. A Lightweight Framework for High-Quality Code Generation. *arXiv:2307.08220* (2023).