Evaluating and Improving the Security of AI-Based Code Assistants

CS 329M Project Proposal

Timothy Gu
Stanford University
Stanford, CA, USA
timothygu@stanford.edu

Abstract

Since OpenAI released the Codex system in 2021 [1], there has been a significant increase in interest into using large machine learning (ML) models as programming aids. The concurrent introduction of commercial products based on these models, such as GitHub Copilot and Amazon CodeWhisperer, further made these tools available to ordinary software developers. However, since these models are often trained on a large volume of publicly available source code of inconsistent quality, the security of inferred code is hardly a guarantee. We seek to concretely improve the security and usability of AI-driven code recommendation platforms. We will first quantify the security and quality of a few of these systems over a range of potential inputs. We then propose the use of input preprocessing, output scanning, and rules-based output rewriting as techniques in an overall outcome improvement system.

1 Background

1.1 Assistant Services

The study will focus on *AI-driven programming assistants.* Examples for such services include line autocomplete services like Kite[1], GitHub Copilot[2] and/or its underlying model OpenAI Codex, and Amazon CodeWhisperer[3].

1.2 Security Issues

The security industry has a variety of categorization schemes for vulnerabilities. Out of the 2021 edition of OWASP Top 10 [2] – categories that represent "the most critical security risks to web applications" according to industry practioners – many of them represent failures in the software design or system configuration, which are typically undertaken by human developers and are not the most applicable to this study. As such, we will narrow our focus on the following items:

2. **Cryptographic Failures:** Recommendations include old or broken ciphers or those that fell out of the general recommendations.
3. **Injection:** Recommendations include injection vulnerabilities, like SQL injection in server code and cross-site scripting (XSS) in HTML generators.

[1]https://www.kite.com/
[2]https://github.com/features/copilot
[3]https://aws.amazon.com/codewhisperer/

1.3 Prior Work

Some previous work has looked into security of large language model code assistants. Sandoval et al. [4] ran a medium-scale user study comparing the coding performance of assisted and unassisted student programmers, finding no significant impact on security. A Stanford lab conducted similar research, though thus far unpublished [3]. However, as far as I am aware, there is no work that look into systematically improving the security of code assistants.

2 Methodology

2.1 Quantifying the Problem

We will first attempt to quantify how (in)secure code generated by assistant services are. We will collect prompts from existing open-source web application projects (both code and comments), datasets (e.g., OpenAI's HumanEval evaluation set[4]). Then, we will pass prompts to each service, and run several code vulnerability scanners on the generated output. We may also incorporate human review for logic bugs or misused cryptographic primitives.

Notably, Amazon CodeWhisperer already includes a code security scanner. However, in my experience it does not catch all vulnerabilities, including an obvious SQL injection attack vector in Python. So we will only use it in conjunction with traditional rules-based scanners.

2.2 Improving Outcome

In order to improve the outcome of programming assistants, we propose investigating a few interlocking methods. These include both MP-specific strategies like Input Preprocessing as well as more traditional methods that also apply to human-written code. See Figure 1 for a summary of how these techniques work together.

[4]https://github.com/openai/human-eval

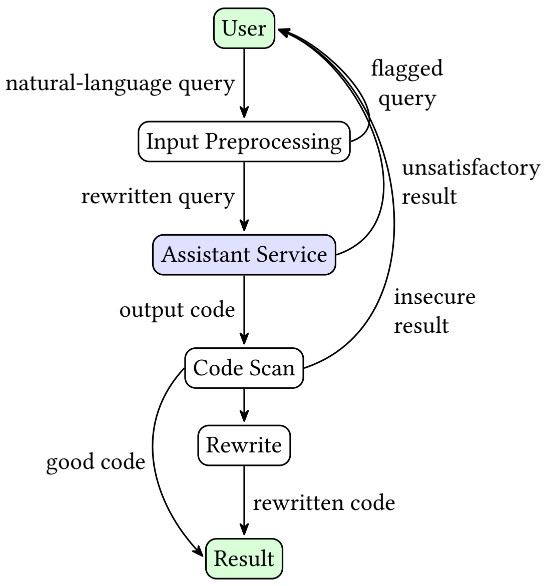
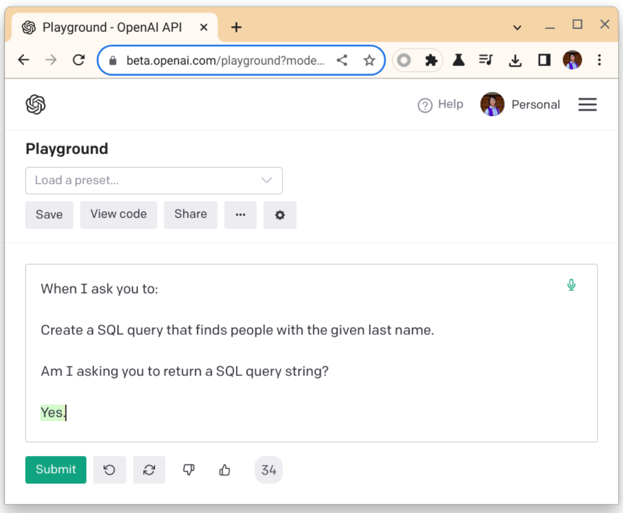


Figure 1. Overview of the outcome improvement engine.



When I ask you to:

Create a SQL query that finds people with the given last name.

Am I asking you to return a SQL query string?

Yes

Figure 2. Inquiring GPT-3 whether the given task requires returning a SQL query string

2.2.1 Input Preprocessing. AI-driven programming assistant models depend on a natural language prompt to generate relevant code. If the prompt describes an insecure software design, an assistant will likely generate insecure code. As an example, the prompt *"Create a SQL query to find people with the given last name"* invites SQL injection in the output code, since the best way to defeat SQL injection is by placing placeholders in the query and separating it from concrete values. We could flag badly designed prompts using natural language understanding, potentially using large language models themselves (see Figure 2).

Second, we could even investigate automatically rewriting the natural language input. One evaluation of Codex [5] noticed that by default, Codex generated code with SQL injection vulnerability in response to *"Insert $name into the people table."* However, after appending *"safely"* to the prompt, Codex generated a SQL-based query free from injections. Similarly, while the prompt *"Generate user profile HTML for name"* elicited insecure code vulnerable to XSS, appending *"using Jinja2"* made it safer by referencing a popular Python HTML template library.

2.2.2 Output Scanning. Despite our best attempts at flagging queries, it is likely impossible to completely avoid bad code from being generated. We can integrate industry-standard code scanners with IDE extensions to further flag suspicious output as a backstop mechanism. Code scanners such as Semgrep and Snyk as well as static analyzers like Coverity are widely adopted in industry. Emerging ML-based solutions such as Merly Mentor can also be integrated.

Though such tools often have high false positive rates, one may argue that model-written code requires a higher level of scrutiny compared to human-written code. Developers may also have higher tolerance for false positives, as the machine programmed code is new to them as well.

2.2.3 Rules-based Output Rewriting. A number of code scanning tools support automatic code rewriting for security (e.g., Snyk) and code style (code formatters and linters). We will attempt to create our own rewriter that detects injection vulnerabilities and ensure correct escaping is done.

3 Timeline

We propose the following timeline:

Time	Task
Week 3	Submit project proposal
Week 5	Security evaluation
Week 8	Outcome improvement
Week 9	Project report due

References

[1] Mark Chen, Jerry Tworek, Heewoo Jun, et al. 2021. Evaluating Large Language Models Trained on Code. arXiv:2107.03374 [cs.LG]

[2] OWASP. 2021. OWASP Top 10:2021. https://owasp.org/Top10/

[3] Neil Perry, Megha Srivastava, and Dan Boneh. 2022. Evaluating security of AI-based programming assistants (unpublished study). (2022).

[4] Gustavo Sandoval, Hammond Pearce, Teo Nys, Ramesh Karri, Brendan Dolan-Gavitt, and Siddharth Garg. 2022. Security Implications of Large Language Model Code Assistants: A User Study. arXiv:2208.09727 [cs.CR]

[5] Smitty. 2021. How good is Codex? https://smitop.com/post/codex/