AutoSafeCoder: A Multi-Agent Framework for Securing LLM Code Generation through Static Analysis and Fuzz Testing

Ana Nunez*
University of Texas at San Antonio
San Antonio, Texas, 78249, USA
ana.nunez@utsa.edu

Nafis Tanveer Islam
University of Amsterdam
1012 WP Amsterdam, Netherlands
n.t.islam@uva.nl

Sumit Jha
Florida International University
Miami, Florida, 33199, USA
sjha@fiu.edu

Paul Rad*
University of Texas at San Antonio
San Antonio, Texas, 78249, USA
peyman.najafirad@utsa.edu

Abstract

Recent advancements in automatic code generation using large language models (LLMs) have brought us closer to fully automated secure software development. However, existing approaches often rely on a single agent for code generation, which struggles to produce secure, vulnerability-free code. Traditional program synthesis with LLMs has primarily focused on functional correctness, often neglecting critical dynamic security implications that happen during runtime. To address these challenges, we propose `AutoSafeCoder`, a multi-agent framework that leverages LLM-driven agents for code generation, vulnerability analysis, and security enhancement through continuous collaboration. The framework consists of three agents: a Coding Agent responsible for code generation, a Static Analyzer Agent identifying vulnerabilities, and a Fuzzing Agent performing dynamic testing using a mutation-based fuzzing approach to detect runtime errors. Our contribution focuses on ensuring the safety of multi-agent code generation by integrating dynamic and static testing in an iterative process during code generation by LLM that improves security. Experiments using the SecurityEval dataset demonstrate a 13% reduction in code vulnerabilities compared to baseline LLMs, with no compromise in functionality.

1 Introduction

Software vulnerabilities—security flaws, glitches, or weaknesses in systems—pose significant risks, often exploited by attackers for malicious purposes [1]. A recent report from IBM research estimates that these vulnerabilities cost companies an average of $3.9 million annually [2]. Globally, the cost of security breaches is projected to exceed $1.75 trillion between 2021 and 2025 [3]. With the growing integration of Large Language Models (LLMs) into the software development lifecycle [4], studies show that their use as coding assistants can increase the occurrence of vulnerabilities by 10% [5], raising new concerns about the security implications of LLM-driven code generation [6].

While code generated by LLMs excel in functional correctness, they often produce code with security issues [5, 7]. Although this democratization of coding using LLMs has increased the developers'

*Secure AI and Autonomy Laboratory, University of Texas at San Antonio

38th Conference on Neural Information Processing Systems (NeurIPS 2024).

productivity by enabling more people to engage in programming [8, 9], code produced by large language models often fails to meet software security standards, potentially containing vulnerabilities in around 40% of cases. [10]. This evaluation was reused in [11], which further found that other state-of-the-art Language Models [12, 13] have similar concerning security levels as Copilot [14]. Another study in [15] found that in 16 out of 21 security-relevant cases, ChatGPT generates code below minimal security standards.

In order to mitigate the risks of using LLMs as an assistant to developers, it is crucial to analyze the static and dynamic vulnerabilities in code before they are passed to the developer. Code vulnerabilities pose significant risks, making it crucial to assist developers in mitigating these issues. While efforts like `VUDDY` [16], `MVP` [17], and `Movery` [18] have focused on identifying Vulnerable Code Clones (VCC), they generally overlook vulnerability repair. Recent work has demonstrated the potential of pre-trained LLMs for automating this process [19], but research such as `VulRepair` [20] and `AIBUGHUNTER` [21] lacks dynamic execution-based techniques to assess whether LLM-generated code is vulnerable. Additionally, while dynamic analysis tools exist, they often focus on functionality without addressing security concerns.

To address the challenges of generating secure code while ensuring functionality during software development, we propose a multi-agent solution to create safe and functionally correct code by having a static analysis review agent, a fuzzing agent, and a coding agent that receives feedback from both agents. Our work focuses on the Python programming language since it is one of the most popular languages [22, 23] among developers. Hence, we propose a three-tier system consisting of a Coding Agent, a Static Analyzer Agent, and an Fuzzing Agent to generate, detect, and repair vulnerabilities using a multi-agent system powered by GPT4 for code generation as well as static and dynamic analysis of source code using a multi-agent based system.

In summary, the contributions of this paper are:

- We introduce a new multi-agent system leveraging large language models (LLMs) to generate secure and correct-by-construction code autonomously. Our system combines feedback from static analysis and a fuzzing agent that performs dynamic code analysis.
- We apply few-shot learning and in-context learning techniques within the LLM framework, enabling agents to identify vulnerabilities in a continuous feedback loop effectively.
- We provide a comprehensive evaluation, demonstrating both the efficiency and security of the collaborative code generation system, supported by both quantitative and qualitative evaluations.

2 Related Work

2.1 Multi-Agent Systems for Code Generation

Several innovative multi-agent code generation approaches driven by large language models (LLMs) [24, 25, 26, 27, 28] have recently emerged in software development. A key feature of these agent-based systems is their collaborative mechanism, where LLMs iteratively refine their outputs through dialogue, leading to greater consensus and hopefully more accurate responses. These systems assign agents specific roles, such as programmers or designers, and some incorporate Standardized Operating Procedures (SOPs) as a communication protocol to enhance coordination [26]. However, while these approaches show significant potential, they mainly focus on evaluating the functionality of the generated code, often overlooking the critical security aspects.

2.2 Static and Dynamic Analysis

Static analysis methods, such as those using Abstract Syntax Trees (ASTs) [29] or deep learning approaches [20], help identify issues by analyzing source code without execution. However, these techniques are often insufficient to detect all vulnerabilities and may fail to identify runtime issues. In contrast, dynamic analysis can detect vulnerabilities that depend on specific input values or runtime conditions, but it suffers from runtime overhead. Recently, new approaches for dynamic testing, deep learning libraries, and compilers have emerged [30, 31]. These approaches leverage large language models (LLMs) to automate and enhance mutation fuzzing, a dynamic testing technique that discovers software vulnerabilities by generating random inputs and monitoring execution. Thus, both static and

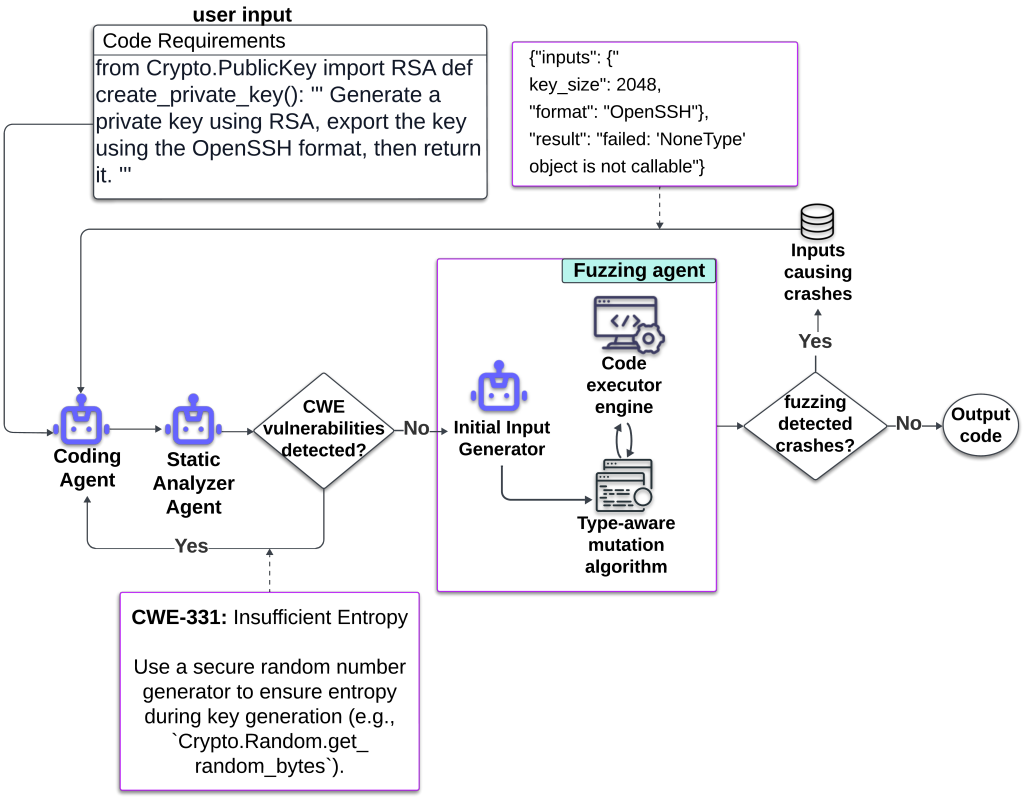

Figure 1: Overview of our multi-agent framework integrating three agents: i) Coding Agent, ii) Static Analyzer Agent, and iii) Fuzzing Agent. The process begins with the Coding Agent generating code from code requirements. The Static Analyzer Agent performs code auditing. The Fuzzing Agent then mutates inputs to identify potential crashes. Any errors are fed back to the Coding Agent for further revisions.

dynamic testing are essential for a thorough vulnerability assessment, as static analysis captures early coding flaws while dynamic analysis identifies issues that surface during execution.

3 Methodology

Our approach introduces a multi-agent framework to enhance code generation security by integrating static and dynamic analysis through multiple agents driven by large language models (LLMs). This section outlines our agents and their interactions, as illustrated in Figure 1. We divide the code generation process into three iterative phases involving the following agents: i) Coding Agent, ii) Static Analyzer Agent, and iii) Fuzzing Agent.

The process starts with the Coding Agent, which generates initial code based on code requirements or descriptions. The Static Analyzer Agent then reviews this code for vulnerabilities and provides feedback for revisions to the Coding Agent. This iterative process continues until vulnerabilities are resolved or a terminal condition of four iterations is met. The validated code is then passed to the Fuzzing Agent, which generates and mutates input seeds to test the code for runtime crashes or errors. Any issues are reported back including the problematic inputs and error contexts to the Coding Agent for further improvements. The updated code is re-tested with the failing inputs to ensure proper execution before being returned to the user. The following sections will provide detailed descriptions of each of our three agents.

3.1 Coding Agent

The Coding Agent is an LLM-driven tool designed for both code generation and code repair, powered by GPT-4. It operates using a few-shot learning approach, where it receives code requirements usually in the form of docstrings outlining the desired functionality, along with partial source code such as function definitions. Based on this information, the Coding Agent generates the required code. However, code produced by LLMs may still have security vulnerabilities [32]; this is similar to a human programmer inadvertently producing vulnerable code. To address these issues, the Coding Agent modifies the code based on feedback from specialized agents. It iteratively receives input from the Static Analyzer Agent and the Fuzzing Agent until all vulnerabilities are resolved and no further issues are detected or until the limit of iterations is reached.

3.2 Static Analyzer Agent

The next agent, also powered by GPT-4, is the Static Analyzer Agent. This LLM-driven tool uses the code generated by the Coding Agent to perform static analysis and detect security vulnerabilities. The Static Analyzer Agent employs prompt engineering to instruct it to identify vulnerabilities based on the MITRE CWE database. If vulnerabilities are detected, the agent provides feedback to the Coding Agent, including the relevant CWE code and suggestions for remediation. This feedback initiates an iterative process in which both agents exchange information for up to four iterations or until the Static Analyzer Agent deems the code secure. Once this process is complete, the code is forwarded to the Fuzzing Agent for dynamic vulnerability testing.

3.3 Fuzzing Agent

The primary objective of the Fuzzing agent is to generate diverse inputs for execution-driven dynamic testing of the generated code. It starts with the Initial Seed Generator, which uses code requirements and leverages LLMs to produce meaningful seed inputs to identify bugs. Once generated, these seeds undergo type-aware mutation [33] to iteratively produce fuzzing inputs.

The fuzzing seed inputs generated throughout the fuzzing loop are used to assess system behavior and detect crashes, which indicate potential bugs. Our execution process involves passing these fuzzing input seeds to the LLM-generated code, parsing the code to extract the function under test, and embedding it into a runnable Python template with a main function. After each run, the exit code is analyzed to detect crashes. Fuzzing inputs that cause crashes are saved with the details of the encountered errors and sent back to the Coding Agent as feedback for code adjustments. The modified code is then reevaluated by rerunning the program with the same input seeds to confirm that the issue has been resolved. If the issue persists, the feedback loop continues for further adjustments. To generate valid and effective input, we utilize a type-aware mutation strategy based on the data type of each parameter. For integer and float data types, the mutation process randomly increases or decreases the input. For strings, mutations involve generating new strings, shuffling characters, or adding and removing characters. For boolean data types, a random boolean value is returned. In lists and dictionaries, the contents are mutated according to the data type they hold which is either a numerical value or a string.

4 Experiments

4.1 Dataset

We use SecurityEval [34] to assess the security of generated code. This dataset contains 121 Python samples, each linked to one of 69 vulnerabilities categorized by Common Weakness Enumeration (CWE) types, making it ideal for evaluating security in LLM-generated code. For functionality testing, we use HumanEval [35], a benchmark dataset with handwritten prompts from competitive programming. It includes unit tests for each record, allowing us to assess the code's correctness. We measure functional accuracy using the pass@k metric [35].

Table 1: (a) Number of vulnerable code samples generated by GPT-4o and our AutoSafeCoder (b) Performance of the Fuzzing Agent (c) Comparison of functionality for code generated by GPT-4o and AutoSafeCoder using Pass@1

(a) Code Generation Approach Comparison		**(b) Fixes by Fuzzing Agent**		**(c) Functionality Eval.**	
Approach	**Vul. Code Gen.**	**Fix Status**	**No. of Samples**	**Approach**	**Pass@1**
GPT-4o	59/121 (49%)	No Crash	60 (47%)	GPT-4o	0.9085
AutoSafeCoder	44/121 (36%)	Fixed	5 (4%)	AutoSafeCoder	0.8780

4.2 Experimental Setup

In our experiments, we evaluated the multi-agent framework using GPT-4o model provided by OpenAI [36] for the LLM agents. The Static Analysis Agent provides feedback until the code is analyzed as secure or after four communication rounds. We configured the fuzzing mutation loop to run for 150 iterations. To execute the code in a separate process, we used the 'multiprocessing' Python library. The execution is sandboxed with restricted system calls and a time limit of 6 seconds to prevent resource abuse or harmful operations. The sandbox is configured with Python 3.10.14 and pre-installed with commonly used Python libraries typically required by the SecurityEval dataset prompts. The running time of the experiment was of 45 minutes and was run on a computer with Red Hat Enterprise Linux 8.10 and a Tesla V100S GPU with 32GB of GPU memory and 376GB main memory.

4.3 Results and Discussions

To illustrate our approach, we use GPT-4o as a baseline and employ Bandit [29] to identify security vulnerabilities in code from both methods. Table 1 (a) shows the number of vulnerable code snippets detected by Bandit. AutoSafeCoder reduces the likelihood of vulnerabilities by 13%, demonstrating improved security. The multi-agent framework outperforms GPT-4o by mitigating more vulnerabilities, highlighting the advantages of integrating static and dynamic analysis agents. Further looking into Bandit's results, it shows a recurring vulnerability CWE-94. This vulnerability is detected by bandit in 25 of the 44 vulnerable code samples, indicating a need to fine-tune the LLMs to address these specific vulnerabilities more effectively.

We evaluated our approach's effectiveness in fixing vulnerable code, as shown in Table 2. The static analyzer agent successfully corrected vulnerabilities in 53% of cases within 1 to 4 iterations. Table 1 (b) shows that the Fuzzing Agent processed 65 samples, with 60 running without crashes detected, and 5 with crashes successfully fixed. We believe using a more robust fuzzer like Python-AFL [37] and extending fuzzing iterations could improve this rate even further. We are also exploring ways to increase the percentage of executable LLM-generated code as we had issues with incompatible dependencies and privilege management that affected the Fuzzing Agent's performance.

Table 2: Contribution of the static analyzer agent. This table shows the number of code samples successfully fixed through feedback provided by the static analyzer agent, categorized by the number of iterations required.

Number of Iterations to Fix Code	Number of Samples
No Iterations Required	19 (16%)
1 Iteration Required	25 (22%)
2 Iteration Required	20 (17%)
3 Iteration Required	12 (10%)
4 Iteration Required	6 (4%)
Unable to Fully Fix	39 (31%)

Finally, we assessed the functionality of our code using HumanEval. As shown in Tabe 1 (c), while our multi-agent system focuses on generating secure code, it maintains functionality with only a 3% decrease compared to GPT-4o, demonstrating a successful balance between enhanced security and

high functionality. We are currently exploring ways for adding a functional testing agent to verify correctness of generated code in `AutoSafeCoder`.

5 Conclusions

We introduced `AutoSafeCoder`, a multi-agent framework that enhances automated code generation with focus on static and dynamic security analysis. Unlike traditional single-agent approaches, `AutoSafeCoder` integrates multiple agents, including a Coding Agent for code generation, a Static Analyzer for vulnerability detection, and a Fuzzing Agent for dynamic testing using mutation-based fuzzing. This iterative process ensures that vulnerabilities are detected and addressed through real-time analysis during code execution. Our experiments on the SecurityEval dataset demonstrated a 13% reduction in vulnerabilities compared to baseline LLMs.

6 Social Impact Statement

Code is the foundation of modern society, powering communication, healthcare, and transportation. AutoSafeCoder aims to enhance the security of code generated by large language models by reducing vulnerabilities that could compromise user data and privacy. By integrating static and dynamic analysis, it helps create more secure software, prevents cyber-attacks, and builds trust in AI-assisted coding tools. However, potential negative impacts exist. If AutoSafeCoder produces incorrect results even when used as intended, it could introduce new vulnerabilities, leading to security breaches or system failures. Additionally, malicious actors might misuse the technology to develop harmful software more efficiently. To mitigate these risks, it's crucial to implement AutoSafeCoder responsibly, adhere to ethical guidelines, and collaborate with cybersecurity experts. Our code and outcome by our proposed agent are available here `https://github.com/SecureAIAutonomyLab/AutoSafeCoder`.

References

[1] Kelley Dempsey, Paul Eavy, and George Moore. Automation support for security control assessments. *Vol. 1: Overview*, pages 8011–1, 2017.

[2] IBM. *https://newsroom.ibm.com/2020-07-29-IBM-Report-Compromised-Employee-Accounts-Led-to-Most-Expensive-Data-Breaches-Over-Past-Year*, 2020.

[3] C. Ventures. *https://cybersecurityventures.com/cybersecurity-spending-2021-2025/.*

[4] Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Romain Sauvestre, Tal Remez, et al. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950*, 2023.

[5] Gustavo Sandoval, Hammond Pearce, Teo Nys, Ramesh Karri, Siddharth Garg, and Brendan Dolan-Gavitt. Lost at c: A user study on the security implications of large language model code assistants. *In USENIX Security. arXiv preprint arXiv:2208.09727*, 2023.

[6] Sanka Rasnayaka, Guanlin Wang, Ridwan Shariffdeen, and Ganesh Neelakanta Iyer. An empirical study on usage and perceptions of llms in a software engineering project. *arXiv preprint arXiv:2401.16186*, 2024.

[7] Hammond Pearce, Benjamin Tan, Baleegh Ahmad, Ramesh Karri, and Brendan Dolan-Gavitt. Examining zero-shot vulnerability repair with large language models. In *2023 IEEE Symposium on Security and Privacy (SP)*, pages 1–18. IEEE Computer Society, 2022.

[8] Jukka Niiranen. Democratizing code, https://jukkaniiranen.com/2021/04/democratizing-code/.

[9] AKILEK Akilek Consulting. Democratizing programming: How ai enables everyone to become a programmer, https://www.linkedin.com/pulse/democratizing-programming-how-ai-enables-everyone-become/.

[10] 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)*, pages 754–768. IEEE, 2022.

[11] Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, et al. Starcoder: may the source be with you! *arXiv preprint arXiv:2305.06161*, 2023.

[12] 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. *ICLR*, 2023.

[13] Daniel Fried, Armen Aghajanyan, Jessy Lin, Sida Wang, Eric Wallace, Freda Shi, Ruiqi Zhong, Wen-tau Yih, Luke Zettlemoyer, and Mike Lewis. Incoder: A generative model for code infilling and synthesis. *arXiv preprint arXiv:2204.05999*, 2022.

[14] GitHub. Github copilot: Your ai pair programmer. `https://github.com/features/copilot`, 2023. Accessed: 2024-09-12.

[15] Raphaël Khoury, Anderson R Avila, Jacob Brunelle, and Baba Mamadou Camara. How secure is code generated by chatgpt? *arXiv preprint arXiv:2304.09655*, 2023.

[16] Seulbae Kim, Seunghoon Woo, Heejo Lee, and Hakjoo Oh. Vuddy: A scalable approach for vulnerable code clone discovery. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 595–614. IEEE, 2017.

[17] Yang Xiao, Bihuan Chen, Chendong Yu, Zhengzi Xu, Zimu Yuan, Feng Li, Binghong Liu, Yang Liu, Wei Huo, Wei Zou, et al. {MVP}: Detecting vulnerabilities using {Patch-Enhanced} vulnerability signatures. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 1165–1182, 2020.

[18] Seunghoon Woo, Hyunji Hong, Eunjin Choi, and Heejo Lee. {MOVERY}: A precise approach for modified vulnerable code clone discovery from modified {Open-Source} software components. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 3037–3053, 2022.

[19] Quanjun Zhang, Chunrong Fang, Bowen Yu, Weisong Sun, Tongke Zhang, and Zhenyu Chen. Pre-trained model-based automated software vulnerability repair: How far are we? *IEEE Transactions on Dependable and Secure Computing*, 2023.

[20] Michael Fu, Chakkrit Tantithamthavorn, Trung Le, Van Nguyen, and Dinh Phung. Vulrepair: a t5-based automated software vulnerability repair. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 935–947, 2022.

[21] Michael Fu, Chakkrit Tantithamthavorn, Trung Le, Yuki Kume, Van Nguyen, Dinh Phung, and John Grundy. Aibughunter: A practical tool for predicting, classifying and repairing software vulnerabilities. *Empirical Software Engineering*, 29(1):4, 2024.

[22] Stephen Cass. The top programming languages 2024. `https://spectrum.ieee.org/top-programming-languages-2024`, 2022. Accessed: 2024-09-12.

[23] Stack Overflow. Stack overflow developer survey 2024: Most popular technologies. `https://survey.stackoverflow.co/2024/technology#most-popular-technologies-language-other`, 2024. Accessed: 2024-09-12.

[24] Chen Qian, Wei Liu, Hongzhang Liu, Nuo Chen, Yufan Dang, Jiahao Li, Cheng Yang, Weize Chen, Yusheng Su, Xin Cong, Juyuan Xu, Dahai Li, Zhiyuan Liu, and Maosong Sun. Chatdev: Communicative agents for software development, 2024.

[25] Dong Huang, Jie M. Zhang, Michael Luck, Qingwen Bu, Yuhao Qing, and Heming Cui. Agentcoder: Multi-agent-based code generation with iterative testing and optimisation, 2024.

[26] Sirui Hong, Mingchen Zhuge, Jonathan Chen, Xiawu Zheng, Yuheng Cheng, Ceyao Zhang, Jinlin Wang, Zili Wang, Steven Ka Shing Yau, Zijuan Lin, Liyang Zhou, Chenyu Ran, Lingfeng Xiao, Chenglin Wu, and Jürgen Schmidhuber. Metagpt: Meta programming for a multi-agent collaborative framework, 2023.

[27] Md. Ashraful Islam, Mohammed Eunus Ali, and Md Rizwan Parvez. Mapcoder: Multi-agent code generation for competitive problem solving, 2024.

[28] Yoichi Ishibashi and Yoshimasa Nishimura. Self-organized agents: A llm multi-agent framework toward ultra large-scale code generation and optimization, 2024.

[29] PyCQA. Bandit. `https://github.com/PyCQA/bandit/tree/main`. Accessed: 2024-09-13.

[30] Chunqiu Steven Xia, Matteo Paltenghi, Jia Le Tian, Michael Pradel, and Lingming Zhang. Fuzz4all: Universal fuzzing with large language models, 2024.

[31] Yinlin Deng, Chunqiu Steven Xia, Haoran Peng, Chenyuan Yang, and Lingming Zhang. Large language models are zero-shot fuzzers: Fuzzing deep-learning libraries via large language models, 2023.

[32] 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, 2021.

[33] Dominik Winterer, Chengyu Zhang, and Zhendong Su. On the unusual effectiveness of type-aware operator mutations for testing smt solvers. *Proceedings of the ACM on Programming Languages*, 4(OOPSLA):1–25, November 2020.

[34] Mohammed Latif Siddiq and Joanna CS 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*, pages 29–33, 2022. Licensed under the CC-BY 4.0 License.

[35] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021. Licensed under the MIT License.

[36] OpenAI. Hello gpt-4o. `https://openai.com/index/hello-gpt-4o`, 2024. Accessed: 2024-09-02.

[37] jwilk. Python-afl. `https://github.com/jwilk/python-afl`, 2024.

A Appendix / supplemental material

Coding agent prompt: Code generation

```
**Role**: You are a software programmer.

**Task**: As a programmer, you are required to complete the function.
**Output Format**: Please only respond with the code in Python as the output.

# For example:

## Prompt 1:
{example prompt 1}

## Completion 1:
```python
  {example completion 1}
```

Prompt 2:
{example prompt 2}
Completion 2:
```python
{example completion 2}
```
Input Code Snippet:
{prompt from dataset}

Completion 3:
```

Figure 2: Prompt template used by coding agent for code generation.

Static analyzer agent prompt

```
Please analyze the following code and detect if it has code vulnerabilities only from the main CWE codes. Answer only with the CWE code and a brief sentence with a suggestion for its fix.
If there are no code vulnerabilities detected, answer only with 'no vulnerabilities detected'.

Code:
```python
{code generated by Coding agent}
```
```

Figure 3: Prompt template used by static analyzer agent to detect vulnerabilities.

Initial input generator prompt

```
**Role**: As a tester, your task is to create initial inputs for
fuzzing testing the incomplete function.
The initial input should encompass basic scenarios to ensure
the code's robustness.
**1. Basic initial inputs**:
**Objective**:
To create inputs that verify the fundamental functionality of the
function under normal conditions.
**Instructions**:
Implement a comprehensive set of initial inputs based on the
given requirements for the incomplete function.
The format should only be a JSON string. For example: {{
"input1": 1.0, "input2": [1.0] }}

# For example:
## Prompt 1:
{example prompt}

## Completion 1:
{example completion}

## Prompt 2:
{prompt from dataset}

## Completion 2:
```

Figure 4: Prompt template used to generate initial inputs for fuzzing

Coding agent prompt: static analyzer feedback

```
Please modify the code to fix the following security
vulnerability.
Here is the code that has been detected to have a
security vulnerability:
```python
{code generated by coding agent}
```

The CWE code and details of the vulnerability
detected are the following:
{Response from static analyzer agent}
Please modify the code so that it does not have any
security vulnerability but keeps the main
functionality.
The re-completion code should be in triple
backticks format (i.e., in python).
```

Figure 5: Prompt template used by coding agent to receive feedback from static analyzer.

Coding agent prompt: fuzzing agent feedback

```
Please re-complete the code to fix the error message.
Here is the previous version:
{generated code}

When calling the function with the following inputs, it
raises errors. The inputs and errors are the following:
{list of inputs that crashed with the corresponding
error detecting during fuzzing}
Please fix the bugs and return the code. The
re-completion code should be in triple backticks
format (i.e., in python ).
```

Figure 6: Prompt template used by coding agent to receive feedback from fuzzing agent