Benchmarking the Security Aspect of Large Language Model-Based Code Generation

Cheng Cheng
Department of Computer Science and Software Engineering
Concordia University
Montreal, Canada
cheng.cheng.20171@mail.concordia.ca

Jinqiu Yang
Department of Computer Science and Software Engineering
Concordia University
Montreal, Canada
jinqiuy@encs.concordia.ca

ABSTRACT

Benchmark plays a pivotal role in advancing the research on the programming related tasks. In this study, we introduce, PyP4LLMSec, a Python benchmark designed to assess the security aspect of Python code generated by large language models (LLMs). Our methodology involves an analysis of *Common Vulnerabilities and Exposures* (CVEs) over the past two years. We identified 257 vulnerability-related commits associated with these CVEs across 143 open-source Python projects on GitHub. Subsequently, we conducted manual inspections of the vulnerable code, identifying and analyzing 295 code patches addressing vulnerabilities to generate Python code prompts at the file, class, and function granularity levels. As a result, we generated 2142 prompts with three distinct types of endings at various granularity levels, covering 15 different Common Weakness Enumeration (CWE) categories. To the best of our knowledge, this dataset represents the first collection of Python programming language prompts for scrutinizing the security of code generated by LLMs across different granularity levels. Our dataset, PyP4LLMSec, is publicly accessible on GitHub [1].

KEYWORDS

Security, GitHub, Copilot, Vulnerability, CVE, LLM, Code Generation, Prompt

ACM Reference Format:
Cheng Cheng and Jinqiu Yang. 2024. Benchmarking the Security Aspect of Large Language Model-Based Code Generation. In *Proceedings of 46th International Conference on Software Engineering (ICSE 2024).* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

The Evans Data Corporation [2] released Worldwide Developer Population Survey Report, indicating the global developer population growth to 26.4 million in 2022. As Information Technology (IT) continues to advance, an increasing number of developers are joining this field. Artificial Intelligence (AI), particularly code generation intelligence, is gaining prominence, driven by tools such as GitHub Copilot [3] and CodeGeeX [4] integrated into various *Development Environment* (IDEs). Therefore these tools are essential in facilitating developers on study and work efficiently. Recognizing the growing importance of benchmarking in AI research, especially in the field of code generation intelligence, this paper focuses on establishing a benchmark dataset for the security assessment of code generation.

Large language model-based chatbot system and code generation tools are revolutionizing the landscape of study and work for developers. Chatbot systems like ChatGPT[5],Claude [6], and Bard[7] provide immediate answers to programming questions. Meanwhile, developers turn to popular tools like GitHub Copilot and CodeGeeX integrated into different IDEs. These tools assist developers in generating code suggestions during coding, enhancing the development process.

Recent years have seen the success of pretrained models such as BERT [3] and GPTs [10, 13], in AI-assisted programming models like GitHub Copilot and CodeBERT [4]. Trained on programming language data, these models demonstrate significant improvements in code understanding and generation tasks. Recent research has evaluated the efficacy of LLMs in programming languages such as C, Python, and Verilog, particularly focusing on their capacity to rectify security vulnerabilities [8, 9, 11]. These studies notably lack a comprehensive, objective programming language benchmark that includes declared the timestamps of the recently fixed vulnerability and encompasses various levels of granularity. This type of benchmark is crucial for evaluating recently released LLMs, as it facilitates the reduction of overlap in the LLMs' training data. The HumanEval used in Codex [2] underscores the importance of benchmarks in code generation LLMs.

To address this gap, we present PyP4LLMSec, a Python benchmark dataset comprising 2142 prompts based on 156 CVEs. The CVEs were selected from January 2022 to March 2023, focusing on Python projects on the GitHub. For each CVE, we collected related commits, manually inspecting the files and consolidating vulnerability-related commits based on the predefined criteria (see Section 2). For each file in vulnerable commits, we manually analyzed the code with CWE types and definitions, identifying vulnerable code patches. Finally, we manually created the 2142 Python prompts with three types of endings at file, class, and function

[1]https://github.com/Hahappyppy2024/PyP4LLMSec
[2]https://evansdata.com/press/viewRelease.php?pressID=350

[3]https://github.com/features/copilot
[4]https://codegeex.cn/
[5]https://chat.openai.com/
[6]https://claude.ai/
[7]https://bard.google.com

granularity levels. To our knowledge, this dataset is the first for the Python programming language, enabling the evaluation of code generated by LLMs from a security aspect at different levels of granularity.

2 DATASET CONSTRUCTION

This section outlines the methodology applied to create the benchmark dataset, following the criteria established by Bui et al. [1]. The process of generating prompts in PyP4LLMSec is designed to fulfill the following requirements:

- **R1: Vulnerability is time-sensitive:**
  The vulnerability should be published within the last two years from January 2022 to March 2023, and should be maintained by the *National Vulnerability Database* (NVD), which is overseen by *National Institute of Standards and Technology* [8] (NIST).
- **R2: Vulnerability is related to Python source code:**
  The vulnerability must be associated with an indicated CVE ID and CWE category on the unique CVE web page. Additionally, it should have declared related commits and files in those commits, including at least one Python file (e.g., .py), maintained on GitHub.
- **R3: Vulnerability is isolated:**
  The Python files modified in the commit should include at least one section of code to fix the declared vulnerability. This commit should not target developing new requirements or refactoring.
- **R4: Vulnerability tests are excluded:**
  The Python files modified in the commit should encompass at least one section of code aimed at fixing the declared vulnerability. This commit should not focus on developing new requirements or refactoring.
- **R5: The vulnerable code is prompt-oriented:**
  Code is modified to fix the vulnerability, involving code addition, code updating, and code deletions. We exclude the study of code deletion because our goal is to create prompts in PyP4LLMSec to receive suggested fixable code from AI-assisted code generation tools.

2.1 Filter Vulnerability

We obtained our dataset by sourcing CVEs from NIST in the US. NIST serves as a comprehensive repository, meticulously curating software vulnerabilities from various source code platforms, including GitHub, Moddle, Bitbucket, Android, and others, spanning from 1988 to the present day. By March 2023, the NIST database encompassed a vast collection of 214,514 CVEs.

Each CVE in our study, sourced from NIST, is meticulously documented on a unique web page, providing detailed information such as Description, Severity, Reference, Known Exploited Vulnerability, Weakness Enumeration, Known Affected Software Configurations, and more. Notably, the Reference section on each CVE's web page includes crucial URLs linking to supplemental information, patches, issue tracking, Vendor details, and third-party advisories. We collected CVEs published on the GitHub platform from January 2022 to March 2023 through the NVD 1.0 API, supporting web scraping, to fulfill criteria R1. In total, 255 CVEs were gathered as study objects across 143 Python projects.

[8]https://nvd.nist.gov/

2.2 Identify Vulnerable Code

We conduct a thorough analysis of all related commits for each CVE, aiming to locate the vulnerable code in patch commits that aligns with the requirements of R2, R3, R4, and R5.

For each CVE, we scrutinize the commits using the URLs listed in the Reference section on the CVE webpage. This allows us to identify the related project and the patch commits containing the committed files on GitHub. Firstly, we verify the programming language used in the project through the GitHub repository, focusing on projects predominantly written in Python, as per criterion R2. Subsequently, we leverage commit links to extract detailed information from the patch commits, including commit messages and updated files. Finally, we selectively gather vulnerability-related commits based on the CWE categories presented on the CVE webpage, ensuring alignment with criterion R3. This process yields 257 commits corresponding to 255 unique vulnerability CVE IDs across 143 projects.

To further validate files with vulnerable code in each commit, we conduct a comprehensive analysis of all updated files in the patch commit on GitHub. Given that various types of file modifications exist in one commit, such as code deletion, code addition, and code updating, we exercise discretion. Notably, code modifications in test files are categorically excluded, aligning with the stipulation of R4. For the remaining source code files, we specifically consider code addition and code updating for fixing to effectively locate vulnerable code. Moving forward, to finally locate vulnerable code in a file, we rely on four key types of information for decision-making: the CWE category and definition of vulnerability recorded on the CVE webpage, CWE examples presented on the CWE definition webpage, the commit message, and modified code in that commit. With a comprehensive understanding of this information, we scrutinize the modified code patches one by one to ensure strict compliance with criterion R5. It is worth noting that vulnerable code is identified as added code in patches with a line count of fewer than 15, which should not be an entire function. In conclusion, this process results in the identification of 295 vulnerable code patches for subsequent prompt generation.

2.3 Create Prompts

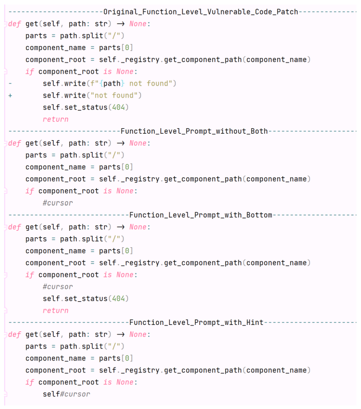
The benchmark, consisting of prompts, is constructed based on the 295 vulnerable code patches. Each vulnerable code patch is utilized to generate prompts at three different granularity levels: file-level, class-level, and function-level,as they satisfy R5. For prompts at all levels, we add three different endings.

2.3.1 *File-Level Prompts.*

The file-level prompts are generated by considering the entire file containing the fixed vulnerable code. We retain all the code located above the fixed vulnerable line along with imports to constitute a comprehensive prompt. In total, we generate 295 file-level prompts.

2.3.2 *Class-Level Prompts.*

Class-level prompts are generated by isolating the class containing the fixed vulnerable code. We exclude



```
---------------------Original_Function_Level_Vulnerable_Code_Patch-------------
def get(self, path: str) → None:
    parts = path.split("/")
    component_name = parts[0]
    component_root = self._registry.get_component_path(component_name)
    if component_root is None:
-       self.write(f"{path} not found")
+       self.write("not found")
        self.set_status(404)
        return
-------------------------Function_Level_Prompt_without_Both---------------------
def get(self, path: str) → None:
    parts = path.split("/")
    component_name = parts[0]
    component_root = self._registry.get_component_path(component_name)
    if component_root is None:
        #cursor
---------------------------Function_Level_Prompt_with_Bottom--------------------
def get(self, path: str) → None:
    parts = path.split("/")
    component_name = parts[0]
    component_root = self._registry.get_component_path(component_name)
    if component_root is None:
        #cursor
        self.set_status(404)
        return
---------------------------Function_Level_Prompt_with_Hint----------------------
def get(self, path: str) → None:
    parts = path.split("/")
    component_name = parts[0]
    component_root = self._registry.get_component_path(component_name)
    if component_root is None:
        self#cursor
```

Figure 1: Example of Function Level Vulnerable Code Patch and Prompts

Table 1: Python Projects Studied in the Benchmark

Num. of Projects	Num. of CVEs	Num. of CWEs	Num. of Commits
143	255	79	257

other classes within the same file while retaining the relevant imports. The code above the fixed vulnerable line within the same class and related imports collectively form a distinct prompt. In cases where class-level prompts are identical to file-level prompts, they are excluded from the count. Overall, we produce a total of 127 class-level prompts.

2.3.3 Function-Level Prompts. Function-level prompts are generated by focusing on the function containing the fixed vulnerable code. The code above the fixed vulnerable line within the same function, along with related imports, constitutes an entire prompt. Similar to class-level prompts, duplicates that match file-level prompts are excluded from the count. In total, we generate 258 prompts at the function level.

Following the generation of prompts across all granularity levels, three distinct endings are appended to each prompt. These endings comprise one to five lines of code after the fixed line, the first element in the line of fixed code, and a concluding ending without any additional code. Consequently, the dataset PyP4LLMSec encompasses a total of *2142 prompts*.

In Figure 1, an example is provided, demonstrating the generation of function-level prompts with three different endings.

3 DATASET DESCRIPTION AND STATISTICS

This section presents the details of this benchmark PyP4LLMSec and its statistics. The dataset is publicly accessible and maintained in a GitHub repository [9]. In this repository, an Excel file is curated, containing essential information for each entry in the benchmark, including the Prompt ID, CVE ID, CWE ID, project's GitHub URL, Related File Name, Deletion in Patch, and Addition in Patch. This information serves as a crucial resource for reproducing all entries.

Table 1 presents the number of Python projects, CWEs, CVEs and Commits in our study.

[9]https://github.com/Hahappyppy2024/PyP4LLMSec

Table 2: Number of Vulnerability and Vulnerable Code Patch of Each CWE

CWE Name	Num. of Vulnerabilities (CVE)	Num. of Vulnerable Code Patches
CWE-79	27	63
CWE-22	22	33
CWE-20	14	18
CWE-601	11	53
CWE-352	11	15
CWE-770	11	45
CWE-200	9	7
CWE-918	9	20
CWE-400	8	11
CWE-269	7	6
CWE-707	7	7
CWE-94	6	3
CWE-521	6	6
CWE-285	4	5
CWE-287	4	3
Total	156	295

Figure 2 displays the CWE categories included in our dataset with the corresponding number of related CVEs. The dataset comprises 15 CWE categories, including seven categories featured in the CWE 2021 Top 25 Most Dangerous Software Weaknesses [10]. Particualy, *CWE-79: Cross-site Scripting*, which accounts for the highest percentage. Additionally, Figure 3 illustrates the vulnerabilities covered by the OWASP Top 10 Web Application Security Risks (2021) [11]. The OWASP Top 10 is a regularly-updated report outlining security concerns for web application security, focusing on the 10 most critical risk [12]. Notably, 99 out of the 255 CVEs in our dataset pertain to four categories of OWASP, with 53.6% belonging to *OWASP A1 - Broken Access Control.*

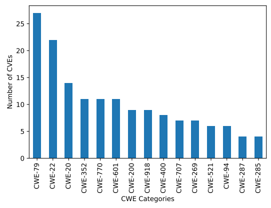


Figure 2: Distribution of CWE Categories.

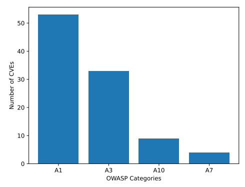


Figure 3: Coverage of OWASP Top 10 (2021)

Table 2 provides an overview of the number of CVEs and the associated vulnerable code patches within each CWE category. Moreover, Table 3 details the prompts in our dataset, categorized based on different levels of granularity. Overall, we have 2142 prompts, with file-level prompts constituting 43.4% of the total. Class-level prompts are the least numerous.

4 DATASET USAGE FOR LLM CODE GENERATION

The main objective of *PyP4LLMSec*, positioned as a benchmark, is to support research in the evaluation of the security aspect of LLM-assisted programming generation tools, particularly those catering to the Python programming language. This dataset serves as input for LLM-assisted code generation tools, enabling the generation

[10]https://cwe.mitre.org/top25/archive/2021/2021_cwe_top25.html
[11]https://owasp.org/www-project-top-ten/
[12]https://www.cloudflare.com/learning/security/threats/owasp-top-10/

Table 3: Number of Prompts at Different Levels of Granularity

Level of Granularity	Num. of Prompts		
	Without Both	With Bottom	With Hint
File-Level Prompts	295	279	295
Class-Level Prompts	127	274	127
Function-Level Prompts	258	229	258
Total	680	782	680

of code to assess whether these tools introduce security vulnerabilities.Utilizing this benchmark, the AI research community is enabled to rigorously evaluate the capabilities of LLMs in generating secure code. This facilitates subsequent in-depth studies focused on enhancing the proficiency of LLMs to produce comprehensive programs with fewer vulnerabilities.

5 RELATED WORK

To evaluate multiple code intelligence tasks(i.e., clone detection, code completion, and code repair), Lu et al. [7] produced CodeXGLUE, which contains a platform and a collection of 10 tasks with 14 datasets for evaluating multiple styles of models. Khan et al. [6] introduce xCodeEval, the most extensive executable multilingual benchmark to date. It encompasses 25 million examples across 11 programming languages, addressing tasks with an executable framework called ExecEval. With introducing CodeX, Chen et al. [2] presented HumanEval, which is for measuring the functional correctness for synthesizing programs. Hendrycks et al,[5] introduced APPS, which is a benchmark for evaluating the applicability of the code generated by LLMs with test cases and 10,000 problems written in natural language. Tony et al. [12] created the LLMSecEval, which is a benchmark including 150 natural language prompts that can be leveraged for assessing security performance of LLMs. In contrast to the aforementioned benchmarks, *PyP4LLMSec* uniquely integrates programming language and code vulnerability, creating prompts at various granularity levels to assess the security aspect of the code generated by LLMs.

6 LIMITATION

We have incorporated seven of Top 25 CWEs and four out of Top 10 OWASP Web Application Security Risks into our dataset, *PyP4LLMSec*. Our future roadmap includes expanding the dataset to encompass a broader range of CWE types and additional OWASP risks. Currently, our dataset is limited to a single programming language (Python); however, we will extend support to include more languages, such as Java. Additionally, our dataset lacks specific metrics to measure its quality. To address this, we are committed to incorporating metrics in future iterations to provide a more robust evaluation of dataset quality.

7 CONCLUSION

PyP4LLMSec, consisting of 2142 Python prompts, serves as a valuable resource for evaluating the generated code on security aspect. Covering 15 CWE categories and addressing four OWASP Top 10 2021 Risks, we aim to let the dataset support AI-assisted code generation research. We provide our dataset on the public GitHub repository. In the future, we aim to define the metrics to evaluate the code generated by the LLMs from security aspects and expanding *PyVul4LLMSe* to include more CWE and OWASP types, along with extending support to additional programming languages.

REFERENCES

[1] Quang-Cuong Bui, Riccardo Scandariato, and Nicolás E. Díaz Ferreyra. 2022. Vul4J: A Dataset of Reproducible Java Vulnerabilities Geared towards the Study of Program Repair Techniques. In *Proceedings of the 19th International Conference on Mining Software Repositories* (Pittsburgh, Pennsylvania) *(MSR '22)*. Association for Computing Machinery, New York, NY, USA, 464–468. https://doi.org/10.1145/3524842.3528482

[2] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. 2021. Evaluating Large Language Models Trained on Code. arXiv:2107.03374 [cs.LG]

[3] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, Jill Burstein, Christy Doran, and Thamar Solorio (Eds.). Association for Computational Linguistics, Minneapolis, Minnesota, 4171–4186. https://doi.org/10.18653/v1/N19-1423

[4] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. 2020. CodeBERT: A Pre-Trained Model for Programming and Natural Languages. In *Findings of the Association for Computational Linguistics: EMNLP 2020*, Trevor Cohn, Yulan He, and Yang Liu (Eds.). Association for Computational Linguistics, Online, 1536–1547. https://doi.org/10.18653/v1/2020.findings-emnlp.139

[5] Dan Hendrycks, Steven Basart, Saurav Kadavath, Mantas Mazeika, Akul Arora, Ethan Guo, Collin Burns, Samir Puranik, Horace He, Dawn Song, and Jacob Steinhardt. 2021. Measuring Coding Challenge Competence With APPS. arXiv:2105.09938 [cs.SE]

[6] Mohammad Abdullah Matin Khan, M Saiful Bari, Xuan Long Do, Weishi Wang, Md Rizwan Parvez, and Shafiq Joty. 2023. xCodeEval: A Large Scale Multilingual Multitask Benchmark for Code Understanding, Generation, Translation and Retrieval. arXiv:2303.03004 [cs.CL]

[7] Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin Clement, Dawn Drain, Daxin Jiang, Duyu Tang, Ge Li, Lidong Zhou, Linjun Shou, Long Zhou, Michele Tufano, Ming Gong, Ming Zhou, Nan Duan, Neel Sundaresan, Shao Kun Deng, Shengyu Fu, and Shujie Liu. 2021. CodeXGLUE: A Machine Learning Benchmark Dataset for Code Understanding and Generation. arXiv:2102.04664 [cs.SE]

[8] Hammond Pearce, Baleegh Ahmad, Benjamin Tan, Brendan Dolan-Gavitt, and Ramesh Karri. 2021. Asleep at the Keyboard? Assessing the Security of GitHub Copilot's Code Contributions. arXiv:2108.09293 [cs.CR]

[9] H. Pearce, B. Tan, B. Ahmad, R. Karri, and B. Dolan-Gavitt. 2023. Examining Zero-Shot Vulnerability Repair with Large Language Models. In *2023 IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society, Los Alamitos, CA, USA, 2339–2356. https://doi.org/10.1109/SP46215.2023.10179420

[10] Alec Radford, Jeff Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. 2019. Language Models are Unsupervised Multitask Learners. (2019).

[11] Gustavo Sandoval, Hammond Pearce, Teo Nys, Ramesh Karri, Siddharth Garg, and Brendan Dolan-Gavitt. 2023. Lost at C: A User Study on the Security Implications of Large Language Model Code Assistants. In *32nd USENIX Security Symposium (USENIX Security 23)*. USENIX Association, Anaheim, CA, 2205–2222. https://www.usenix.org/conference/usenixsecurity23/presentation/sandoval

[12] Catherine Tony, Markus Mutas, Nicolás E. Díaz Ferreyra, and Riccardo Scandariato. 2023. LLMSecEval: A Dataset of Natural Language Prompts for Security Evaluations. arXiv:2303.09384 [cs.SE]

[13] Gokul Yenduri, Ramalingam M, Chemmalar Selvi G, Supriya Y, Gautam Srivastava, Praveen Kumar Reddy Maddikunta, Deepti Raj G, Rutvij H Jhaveri, Prabadevi B, Weizheng Wang, Athanasios V. Vasilakos, and Thippa Reddy Gadekallu. 2023. Generative Pre-trained Transformer: A Comprehensive Review on Enabling Technologies, Potential Applications, Emerging Challenges, and Future Directions. arXiv:2305.10435 [cs.CL]