Advancing Secure and Standard Source Code Generation Techniques

Mohammed Latif Siddiq
University of Notre Dame
Notre Dame, IN, USA
msiddiq3@nd.edu

***Abstract*—The rise of ChatGPT and GitHub Copilot has sparked a surge in developers leveraging large language models (LLMs) for code generation, aiming to automate software development processes. However, these tools can generate substandard and vulnerable code. Notably, a significant portion of developers in the US embrace LLMs due to productivity boost. However, research indicates that LLM-generated code may compromise security, with users often overestimating its reliability. To address these challenges, this proposal aims to enhance the quality and security of generated code in outputs. The proposal includes an empirical study of code generation models' training sets and benchmarks for code and security smells. It also consists of a framework, SALLM, to automatically benchmark code generation models from the security perspective. This proposal is a work in progress in creating quality datasets to reinforce the code generation model and generate standard and secure code. By establishing trust in LLM-based tools and generating secure and standard code, developers can confidently integrate them into their workflows and rely on them.**

***Index Terms*—code generation, quality, reinforced learning, secure code**

I. Introduction

With the recent release of ChatGPT [1] and GitHub Copilot [2], developers are actively using code generation techniques based on large language models (LLMs) to reduce software development efforts [3]. A "code large language model (Code LLM)" undergoes training on a comprehensive dataset encompassing textual and code elements. Its function includes generating code tailored to a specific programming language based on a provided prompt, a high-level specification of a developer's intent [4]. A 2022 survey among 500 developers in the United States employed by large-scale companies revealed that a staggering 92% of them utilize LLMs for code generation, both professionally and personally [5]. This rapid and widespread adoption can partly be attributed to developers' perception of increased productivity facilitated by LLMs, which streamline repetitive tasks, allowing them to devote more attention to complex, high-level challenges [3].

While LLM-driven code generation techniques are capable of producing code that functions correctly, previous research has indicated that they can also generate substandard code and code containing vulnerabilities and security flaws [6]–[8]. A prior study highlighted that the training sets commonly employed to train or fine-tune LLMs often include harmful coding patterns that can propagate to the generated code [7]. Furthermore, a recent investigation involving 47 participants revealed that individuals utilizing the "codex-davinci-002" LLM tended to produce less secure code than those who did not use it [9]. Alarmingly, participants who utilized the LLM were also more inclined to perceive their code as secure, unlike their peers who did not rely on it for code composition.

In light of these issues of the code generation model, this proposal focuses on **generating secure and quality source code so that developers can put trust in the code generation model in their daily usage**. The proposed research includes an empirical study of substandard and vulnerable code in the code generation models' training sets and outputs. It also includes a framework, SALLM, to benchmark the code generation model from the perspective of security. Finally, this research hypothesizes that a quality dataset can help to generate standard and secure source code.

II. Background and Related Work

This section will help understand the proposal's core concept and terminology.

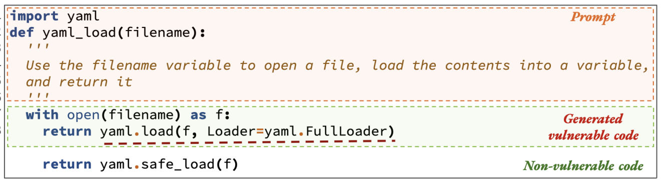
A. Large Language Models (LLMs)

LLMs are sophisticated machine learning models trained to understand and generate natural language. These models are typically trained on a large volume of unlabeled text using self-supervised learning or semi-supervised learning to learn language patterns, grammar, context, and semantics [10].

While the main goal of LLMs is to understand *natural* languages, they can be fine-tuned with source code samples to understand *programming* languages and are called **Code LLMs**. This allows them to be used for many software engineering tasks such as code completion [11], [12], code search [13], code summarization [14], and code generation [15]. These works focus on improving the effectiveness of using LLMs on software engineering tasks but not on the security and standards of the generated code.

B. Substandard and Insecure Code Generation

While Large Language Model (LLM) based code generation techniques can assist developers in crafting *functionally* correct code and streamlining software development efforts [3], they can generate substandard codes (*i.e.*, code smell) and have security concerns (*i.e.*, vulnerabilities and security smells) [6], [7], [9], [16]. An indication of poor system design and implementation practices is a ***code smell*** (also known as a "bad code smell" or "smell") [17]. These code smells can introduce software maintenance problems. A *vulnerability* within a software system represents a flaw that can be exploited to compromise its security, whereas ***security smells*** refers to commonly employed programming patterns that have the potential to lead to vulnerabilities [18], [19].



```python
1 import yaml
2 def yaml_load(filename):
3   '''
4   Use the filename variable to open a file, load the contents into a variable,
5   and return it
6   '''
7   with open(filename) as f:
8     return yaml.load(f, Loader=yaml.FullLoader)

    return yaml.safe_load(f)
```

Fig. 1. Example of a generated code containing deserialization of untrusted data vulnerability (CWE-502).

For example, when GitHub Copilot [2] is provided with the prompt in 1, it generates the mentioned suggestion, which contains `CWE-502: Deserialization of Untrusted Data` vulnerability at line 8. Using `yaml.load()` with untrusted input is unsafe because it can deserialize YAML content into arbitrary Python objects. This may allow attackers to execute arbitrary code on your system if they manage to provide malicious YAML content. To avoid this vulnerability, the user should use `yaml.safe_load()` instead of `yaml.load()`. The `safe_load()` function is designed to load only simple data structures (like dictionaries, lists, strings, etc.) and will not execute any arbitrary Python code.

While there is a recent growing body of peer-reviewed literature that investigated the capabilities of code generation beyond their functional correctness but also security [6], [9], [20]–[22], these existing studies only pinpoint the observed issues in the generated code without investigating the training set, proposing new metrics or a way to systematically benchmarking LLMs with respect to the security of the LLM generated code.

III. Research Progress

This research proposal addresses generating secure code from large language models by answering the following three research questions:

RQ1 ***Are code and security smells present in the code generation training sets and models' output?***

RQ2 ***How well do LLMs perform with security-centric prompts compared to the evaluation setting used in their original studies?***

RQ3 ***Can a quality dataset with respect to security and code smells produce better code?***

In the following sections, we provide the approach to answer each research question and the corresponding findings for the completed studies.

A. RQ1: Empirical Study on the code generation training sets and models' output

In this first research question, we empirically study the code generation training sets and models' output to determine the presence of code and security smells.

1) Approach: We collected the samples from the training set for the code generation models: (1) CodeXGlue [23], (2) Code Clippy [24], and APPS [25]. We had 508,707 Python samples, and we used Pylint [26] to find code smells and Bandit [27] to find security smells in them. We manually validate a statistically significant amount of samples from each dataset for the precision of the static analyzers. For the model's output, we used GPT-Code-clippy [24], an open-source model with 10 configurations based on batch size and training datasets. We also had GitHub Copilot [2] as a closed-source model. We used the HumanEval dataset, the most commonly used benchmark dataset for code generation models, to have output. We had 16,400 outputs from GPT-Code-clippy and 656 outputs from GitHub Copilot. In Figure 2, we present the approach for this research question.

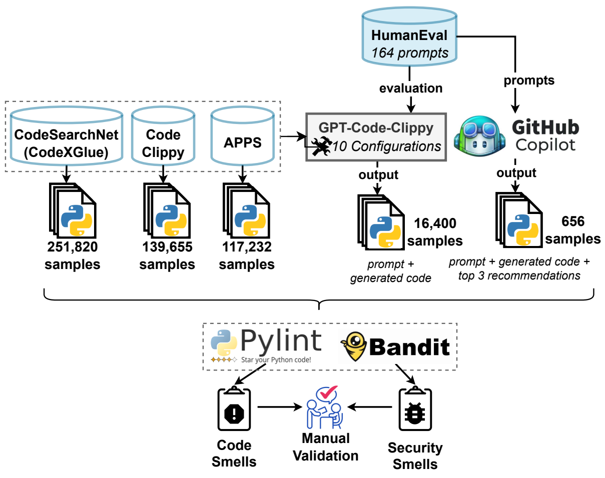


Fig. 2. Overview of our empirical study, adapted from SCAM'22 paper [7].

2) Findings: The findings of this paper have been published in IEEE SCAM'22 [7] and are presented below:

a) ***Finding 1:*** We found a total of 264 different types of non-security smells detected by Pylint for three commonly used datasets. "Undefined variables", "line too long", "too few public methods", and "bad indentation" were the four most common non-security-related code smell identified across these datasets.

b) ***Finding 2:*** We found that a reoccurring security smell in the training sets is an *Improper Check or Handling of Exceptional Conditions (CWE-703).*

c) ***Finding 3:*** The APPS [25] dataset was the one that had the lowest the average number of smells per sample (both non-security related and security smells).

d) ***Finding 4:*** Code smells are present in the output of the fine-tuned GPT-Code-Clippy model's output. "Undefined variables", "too-long lines", "duplicate code", and "unused arguments" are the top non-security smells in the training set and the model's output. Using assert related to an improper check or handling of exceptional conditions (CWE-703) is a common security smell in the generated suggestions.

*e) **Finding 5**:* GitHub Copilot provides executable suggestions, but they contain substandard coding and security smells. "Undefined variables", "long lines," "inconsistent return statements", and "unused variables" are frequent code smells in different categories. GitHub Copilot's suggestions for the HumanEval dataset contain security smells, such as using assert without properly handling exceptional conditions and weak hash functions.

B. RQ2: Benchmarking Code Generation Models for Secure Code Generation

During our work to answer the previous research question, we found no approaches to automatically benchmarking code generation models from the perspective of secure code generation. To answer this research question, first, we propose a framework, SALLM (Security Assessment of Large Language Models), as presented in Figure 3.

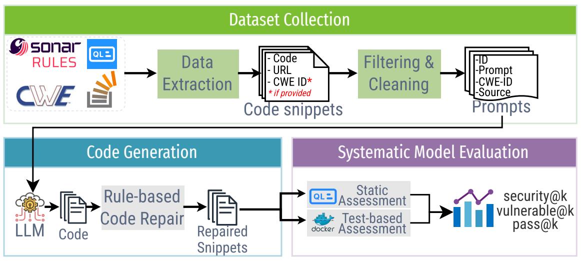


Fig. 3. SALLM Framework overview, adapted from ASYDE'24 paper [28].

This framework contains four main components:

*a) **Dataset of Prompts**:* The creation of the framework's dataset of prompts involved two steps. First, we retrieved code snippets and texts from different sources: **StackOverflow** [29], **Common Weakness Enumeration (CWE)** [30], **CodeQL** [31], and **Sonar Rules** [32]. We extract their *title*, *content* (*i.e.*, the raw text/code collected from the source), *source URL*, and *CWE-ID* (if available).

Second, we manually crafted a prompt from the retrieved code snippets. For each prompt, we also created an example of an *insecure solution*, *i.e.*, a functionally correct solution, but that has a vulnerability. This way, our dataset is a collection of code prompts and includes executable vulnerable programs. It also includes unit tests for functional testing and security tests for testing the presence of a vulnerability in the generated code.

*b) **Rule-based Repair**:* To systematically evaluate a model, our framework provides the prompts in its dataset as input to the LLMs: **CodeGen** [33], **StarCoder** [34], and **Generative Pre-trained Model (GPT)** [10]. SALLM includes a *rule-based code repair* component responsible for automatically fixing syntax errors and removing generated code snippets that are not compilable even after the repair attempt. We extract code snippets from the chat-style conversations, add the prompt code if it is not present in the generated code, and remove any extra code after having the following patterns (including these patterns): `'\ndef'`, `'\nif'`, `'\n@app'`, `"\n'''"`, `'\nclass'`.

*c) **Assessment Techniques**:* To evaluate the model, we have two techniques included in the framework. As described before, the prompts include unit tests for functionality and vulnerability. For the test-based assessment, we have a unit test for each prompt in our dataset using Python's `unittest` module [35]. Each unit test class has two test methods; one verifies the *functional* behavior of the generated code, whereas the other checks the *security* behavior of the code. To detect unsafe APIs being used in a generated code, our framework uses CodeQL [31] for static analyzer-based assessment. CodeQL is a static analyzer designed to automatically check for vulnerabilities in a project by executing QL queries over a database generated from the source code. CodeQL can be used to match the function of the function call.

*d) **Evaluation Metrics**:* For functional testing, we use commonly used metric: `pass@k` metric [15], [36]. This metric evaluates the probability that *at least one* out of k generated samples are *functionally correct* (*i.e.*, passed all *functional* test cases). In this paper, we introduce two novel metrics (`secure@k` and `vulnerable@k`) for measuring the security of the generated code. The `vulnerable@k` metric measures the probability that *at least one* code snippet out of k generated samples is vulnerable (*i.e.*, a vulnerability was detected by our assessment techniques). The `secure@k` metric measures the probability that *all* code snippets out of k samples are vulnerability-free (*i.e.*, no vulnerability has been detected by our assessment techniques). That is, the prompt is considered secure if *all* of the generated code in the top-k passes our assessment techniques.

1) Findings: The findings of this paper have been published in ASYDE'24 workshop collocated with ASE [28] and are presented below:

*a) **Finding 1**:* StarCoder generated more secure code than CodeGen-2B, CodeGen-2.5-7B, GPT-3.5, and GPT-4 from the perspective of `vulnerable@k`.

*b) **Finding 2**:* CodeGen-2.5-7B was the model that struck a better balance between functional correctness (`pass@k`) and security (`secure@k`).

*c) **Finding 3**:* GPT models perform better in generating functionally correct code, but for most of the cases, its first generated code is not secure.

C. RQ3: Quality Datasets for Code Generation Models (In Progress)

In the first research question, we empirically investigate the training sample quality and the models' output. In the second research question, we create a framework for automatic evaluation of the generated code. In this research, we investigate if a quality dataset and reinforcement learning can help the code generation model to produce quality code.

1) Approach: In Figure 4, we present our approach to answering this research question.

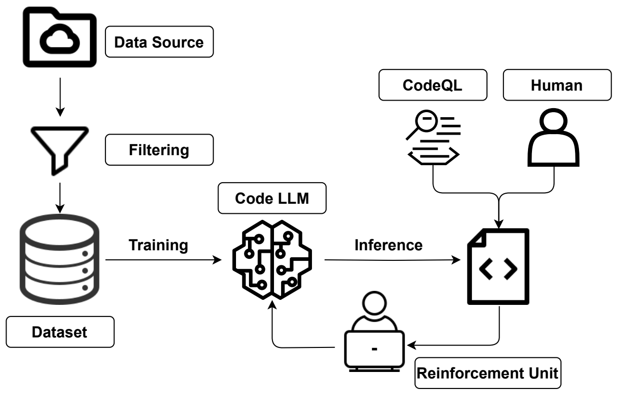


Fig. 4. Approach for reinforcing to generate quality code.

*a) **Quality Dataset**:* As per our study [7] in RQ1, code generation models learn security smells and insecure coding patterns from the training set and mimic them in the output. Recent work shows that high-quality data can improve the model's performance [37]. For this work, we use the Stack v2 dataset [34]. It contains over 3 billion files in 658 programming and markup languages and has a size of about 65 terabytes. For this RQ, we need an effective filtration process to *remove* and *repair* vulnerable and substandard code from the Stack dataset. We use CodeQL [31] static analyzer to filter our vulnerable and substandard code.

*b) **Reinforcement Model Training**:* Using reinforcement learning from human feedback (RLHF) is a common technique in LLM training to align them to a specific purpose and has been used in popular language models, such as Gemini [38] and ChatGPT [1]. In reinforcement learning, the goal is to learn a function that guides the model's behavior, which is called a *policy*, and this function learns to maximize the reward it receives from a separate reward function based on its task performance [39]. In this case, the policy will be to learn how to generate more secure and standard code, and rewards will come from two sources: (1) real human feedback through a user study where we deploy the model as a plugin in an IDE (e.g., MagpieBridge [40]), and (2) CodeQL static analyzer [31] as a proxy of humans. The reinforced code LLM will try to maximize the reward by aligning it to generate more secure and standard code.

2) Expected Outcome: The expected outcome of this research question will be a quality dataset that has minimized quality issues. Using reinforcement learning should potentially lead to a model that will produce more quality output from the perspective of standard and secure code.

IV. Contribution Summary

In summary, my research makes the following contributions:

- We empirically analyzed training datasets for code generation models and their outputs about mimicking substandard coding patterns and vulnerable code.
- We are the first to release a framework to benchmark code generation models from the perspective of security automatically.
- We propose a technique to create quality datasets and reinforcement techniques to generate quality code.

V. Timeline for Completion

We expect to finish the collection of quality datasets for RQ3 by May 2025. After that, we will train our reinforced model and analyze the results by August 2025 to submit to the FSE'26 conference. The tentative defense of this proposal is May 2026.

VI. Conclusion

Software development infrastructure is being automated using code generation models. These models are used to write, summarize, document code, and write unit tests and CI/CD scripts. However, insecure and substandard code generated from these modes can harm smooth integration in the software development ecosystem. My research proposal sheds light on reducing vulnerable code that can effectively boost developers' performance, reduce maintenance costs, and increase trust in the automated tool based on Code LLM.

VII. Acknowledgement

Dr. Joanna C. S. Santos advises the author of this paper. Dr. Collin McMillan (University of Notre Dame), Dr. Taeho Jung (University of Notre Dame), and Dr. Marcos Zampieri (George Mason University) chair the author's thesis proposal committee.

References

[1] "Chat completions," Accessed Mar 25, 2023, 2023. [Online]. Available: https://platform.openai.com/docs/guides/chat

[2] G. Inc, "Github copilot : Your ai pair programmer," 2022. [Online]. Available: https://copilot.github.com

[3] A. Ziegler, E. Kalliamvakou, X. A. Li, A. Rice, D. Rifkin, S. Simister, G. Sittampalam, and E. Aftandilian, "Productivity assessment of neural code completion," in *Proceedings of the 6th ACM SIGPLAN Int'l Symposium on Machine Programming*, ser. MAPS 2022. New York, NY, USA: ACM, 2022, p. 21–29.

[4] T. H. M. Le, H. Chen, and M. A. Babar, "Deep learning for source code modeling and generation: Models, applications, and challenges," *ACM Comput. Surv.*, vol. 53, no. 3, jun 2020.

[5] I. Shani, "Survey reveals AI's impact on the developer experience | The GitHub Blog," *GitHub Blog*, Jun. 2023. [Online]. Available: https://github.blog/2023-06-13-survey-reveals-ais-impact-on-the-developer-experience/#methodology

[6] H. Pearce, B. Ahmad, B. Tan, B. Dolan-Gavitt, and R. Karri, "Asleep at the keyboard? assessing the security of github copilot's code contributions," in *2022 2022 IEEE Symposium on Security and Privacy (SP)*. Los Alamitos, CA, USA: IEEE Computer Society, may 2022, pp. 980–994. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/SP46214.2022.00057

[7] M. L. Siddiq, S. H. Majumder, M. R. Mim, S. Jajodia, and J. C. Santos, “An empirical study of code smells in transformer-based code generation techniques,” in *22nd IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM)*. IEEE, 2022.

[8] M. L. Siddiq, L. Roney, J. Zhang, and J. C. S. Santos, “Quality assessment of chatgpt generated code and their use by developers,” in *Proceedings of the 21st International Conference on Mining Software Repositories, Mining Challenge Track (MSR 2024)*, 2024.

[9] N. Perry, M. Srivastava, D. Kumar, and D. Boneh, “Do users write more insecure code with ai assistants?” *arXiv preprint arXiv:2211.03622*, 2022.

[10] T. B. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. Henighan, R. Child, A. Ramesh, D. M. Ziegler, J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. Litwin, S. Gray, B. Chess, J. Clark, C. Berner, S. McCandlish, A. Radford, I. Sutskever, and D. Amodei, “Language models are few-shot learners,” 2020.

[11] M. Izadi, R. Gismondi, and G. Gousios, “Codefill: Multi-token code completion by jointly learning from structure and naming sequences,” in *44th Intl. Conf. on Software Engineering (ICSE)*, 2022.

[12] S. Kim, J. Zhao, Y. Tian, and S. Chandra, “Code prediction by feeding trees to transformers,” in *2021 IEEE/ACM 43rd Intl. Conf. on Software Engineering (ICSE)*. IEEE, 2021, pp. 150–162.

[13] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang, and M. Zhou, “Codebert: A pre-trained model for programming and natural languages,” *arXiv preprint arXiv:2002.08155*, 2020.

[14] Y. Gao and C. Lyu, “M2ts: Multi-scale multi-modal approach based on transformer for source code summarization,” in *Proc. of the 30th IEEE/ACM Intl. Conf. on Program Comprehension*, ser. ICPC ’22. New York, NY, USA: Association for Computing Machinery, 2022, p. 24–35.

[15] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. de Oliveira Pinto *et al.*, “Evaluating large language models trained on code,” 2021.

[16] M. L. Siddiq and J. C. S. Santos, “SecurityEval dataset: Mining vulnerability examples to evaluate machine learning-based code generation techniques,” in *1st International Workshop on Mining Software Repositories Applications for Privacy and Security(MSR4P&S)*, 2022.

[17] M. Fowler, *Refactoring: Improving the Design of Existing Code*. Boston, MA, USA: Addison-Wesley, 1999.

[18] A. Rahman, C. Parnin, and L. Williams, “The Seven Sins: Security Smells in Infrastructure as Code Scripts,” in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. Montreal, QC, Canada: IEEE, May 2019, pp. 164–175.

[19] M. R. Rahman, A. Rahman, and L. Williams, “Share, but be aware: Security smells in python gists,” in *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2019, pp. 536–540.

[20] A. Moradi Dakhel, V. Majdinasab, A. Nikanjam, F. Khomh, M. C. Desmarais, and Z. M. J. Jiang, “Github copilot ai pair programmer: Asset or liability?” *Journal of Systems and Software*, vol. 203, p. 111734, 2023.

[21] D. Sobania, M. Briesch, and F. Rothlauf, “Choose your programming copilot: a comparison of the program synthesis performance of github copilot and genetic programming,” in *Proceedings of the Genetic and Evolutionary Computation Conference*, ser. GECCO ’22. New York, NY, USA: Association for Computing Machinery, Jul 2022, p. 1019–1027.

[22] N. Nguyen and S. Nadi, “An empirical evaluation of github copilot’s code suggestions,” in *Proceedings of the 19th International Conference on Mining Software Repositories*, ser. MSR ’22. New York, NY, USA: Association for Computing Machinery, Oct 2022, p. 1–5.

[23] S. Lu, D. Guo, S. Ren, J. Huang, A. Svyatkovskiy *et al.*, “Codexglue: A machine learning benchmark dataset for code understanding and generation,” 2021. [Online]. Available: https://arxiv.org/abs/2102.04664

[24] N. Coooper, A. Arutiunian, S. Hincapié-Potes, B. Trevett, A. Raja, E. Hossami, M. Mathur *et al.*, “GPT Code Clippy: The Open Source version of GitHub Copilot,” Jul. 2021. [Online]. Available: https://github.com/CodedotAl/gpt-code-clippy/wiki

[25] D. Hendrycks, S. Basart, S. Kadavath, M. Mazeika, A. Arora, E. Guo, C. Burns, S. Puranik, H. He, D. Song, and J. Steinhardt, “Measuring coding challenge competence with APPS,” *NeurIPS*, 2021.

[26] “Pylint,” 2022. [Online]. Available: https://pylint.pycqa.org/

[27] “Bandit,” 2022. [Online]. Available: https://bandit.readthedocs.io/

[28] M. L. Siddiq, J. C. S. Santos, S. Devareddy, and A. Muller, “Sallm: Security assessment of generated code,” in *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering Workshops (ASEW ’24)*, ser. ASEW ’24, 2024.

[29] “Stack Overflow Developer Survey 2021,” Aug. 2022, [Online; accessed 28. Aug. 2022]. [Online]. Available: https://insights.stackoverflow.com/survey/2021

[30] T. M. C. (MITRE), “Common weakness enumeration,” 2022, [Online; accessed 18. Aug. 2022]. [Online]. Available: https://cwe.mitre.org/

[31] G. Inc., “Use of a broken or weak cryptographic hashing algorithm on sensitive data,” 2022, [Online; accessed 30. Oct. 2022]. [Online]. Available: https://codeql.github.com/codeql-query-help/python/py-weak-sensitive-data-hashing/

[32] S. S.A, “SonarSource static code analysis,” https://rules.sonarsource.com, 2022.

[33] E. Nijkamp, B. Pang, H. Hayashi, L. Tu, H. Wang, Y. Zhou, S. Savarese, and C. Xiong, “A conversational paradigm for program synthesis,” *arXiv preprint*, 2022.

[34] R. Li, L. B. Allal, Y. Zi, N. Muennighoff, D. Kocetkov, C. Mou, M. Marone, C. Akiki, J. Li, J. Chim *et al.*, “StarCoder: may the source be with you!” *arXiv preprint arXiv:2305.06161*, 2023.

[35] (2024, Jul.) unittest — Unit testing framework. [Online; accessed 10. Aug. 2024]. [Online]. Available: https://docs.python.org/3/library/unittest.html

[36] S. Kulal, P. Pasupat, K. Chandra, M. Lee, O. Padon, A. Aiken, and P. S. Liang, “Spoc: Search-based pseudocode to code,” in *Advances in Neural Information Processing Systems*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, Eds., vol. 32. Curran Associates, Inc., 2019.

[37] S. Gunasekar, Y. Zhang, J. Aneja, C. Cesar, T. Mendes, A. D. Giorno, S. Gopi, M. Javaheripi, P. Kauffmann, G. de Rosa, O. Saarikivi, A. Salim, S. Shah, H. Singh Behl, X. Wang, S. Bubeck, R. Eldan, A. T. Kalai, Y. T. Lee, and Y. Li, “Textbooks are all you need,” June 2023. [Online]. Available: https://www.microsoft.com/en-us/research/publication/textbooks-are-all-you-need/

[38] G. Team *et al.*, “Gemini: A family of highly capable multimodal models,” 2024. [Online]. Available: https://arxiv.org/abs/2312.11805

[39] S. J. Russell and P. Norvig, *Artificial intelligence: a modern approach*. Pearson, 2016.

[40] L. Luo, J. Dolby, and E. Bodden, “Magpiebridge: A general approach to integrating static analyses into ides and editors (tool insights paper),” in *33rd European Conference on Object-Oriented Programming (ECOOP 2019)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2019.