

# Case Study: Fine-tuning Small Language Models for Accurate and Private CWE Detection in Python Code

Md. Azizul Hakim Bappy<sup>\*1</sup>, Hossen A Mustafa<sup>†1</sup>, Prottoy Saha<sup>‡1</sup>, and Rajinus Salehat<sup>§2</sup>

<sup>1</sup>Institute of Information and Communication Technology, Bangladesh University of Engineering Technology, Dhaka, Bangladesh

<sup>2</sup>Hajee Mohammad Danesh Science and Technology University, Dinajpur, Bangladesh

## Abstract

Large Language Models (LLMs) have demonstrated significant capabilities in understanding and analyzing code for security vulnerabilities, such as Common Weakness Enumerations (CWEs). However, their reliance on cloud infrastructure and substantial computational requirements pose challenges for analyzing sensitive or proprietary codebases due to privacy concerns and inference costs. This work explores the potential of Small Language Models (SLMs) as a viable alternative for accurate, on-premise vulnerability detection. We investigated whether a 350-million parameter pre-trained code model (codegen-mono) could be effectively fine-tuned to detect the MITRE Top 25 CWEs specifically within Python code. To facilitate this, we developed a targeted dataset of 500 examples using a semi-supervised approach involving LLM-driven synthetic data generation coupled with meticulous human review. Initial tests confirmed that the base codegen-mono model completely failed to identify CWEs in our samples. However, after applying instruction-following fine-tuning, the specialized SLM achieved remarkable performance on our test set, yielding approximately 99% accuracy, 98.08% precision, 100% recall, and a 99.04% F1-score. These results strongly suggest that fine-tuned SLMs can serve as highly accurate and efficient tools for CWE detection, offering a practical and privacy-preserving solution for integrating advanced security analysis directly into development workflows.

**Keywords:** Small Language Models (SLMs), Vulnerability Detection, CWE, Fine-tuning, Python Security, Privacy-Preserving Code Analysis.

## 1 Introduction

Software security has become an undeniable cornerstone of our interconnected digital world. The increasing complexity and pervasiveness of software systems have, unfortunately, been paralleled by a rise in software vulnerabilities, making applications prime targets for malicious exploitation [1]. Among the critical classes of software weaknesses, Common Weakness Enumerations (CWEs) stand out as fundamental flaws in code that can lead to a cascade of security issues [2]. The ability to detect and remediate CWEs early in the software development lifecycle is therefore paramount,

---

<sup>\*</sup>0424312039@iict.buet.ac.bd

<sup>†</sup>hossen.mustafa@iict.buet.ac.bd

<sup>‡</sup>prottoysaha@iict.buet.ac.bd

<sup>§</sup>rajinussalehat@gmail.com

offering significant cost savings by preventing costly breaches and reducing the effort required for late-stage fixes [3]. Traditionally, approaches to CWE detection have relied on static analysis tools and manual code reviews [4]. While these methods are valuable, they can be resource-intensive, prone to false positives and negatives, and may struggle with the nuanced understanding required to identify complex weakness patterns. The emergence of Large Language Models (LLMs) has heralded a new era in code analysis and software engineering [5]. These models, with their remarkable ability to understand and generate code, have shown impressive capabilities in various security tasks, including vulnerability detection and CWE identification [6]. However, a significant hurdle remains for organizations handling sensitive or proprietary code, such as in finance, healthcare, or government sectors. These entities often face stringent data governance policies and security protocols that restrict or entirely prohibit the transmission of their codebase to external cloud services for analysis [7]. This creates a critical gap: while LLM-powered security tools offer promising capabilities, their inherent cloud dependency makes them inaccessible for on-premise security analysis in many contexts, leaving organizations with confidential codebases with limited options for leveraging state-of-the-art language model technology for CWE detection. In response to this challenge, this paper explores the potential of Small Language Models (SLMs) as a viable and effective solution for on-premise CWE detection. SLMs, with their reduced parameter count and computational footprint, offer several key advantages. Firstly, and most importantly in this context, they can be deployed directly within an organization’s infrastructure, ensuring that sensitive code remains secure and under local control [8]. Secondly, SLMs are significantly less computationally demanding than their larger counterparts, requiring fewer resources for both deployment and inference, potentially leading to faster analysis and lower operational costs [9]. This makes them particularly attractive for environments with limited computational resources or where rapid analysis is critical. In this work, we hypothesize that a carefully fine-tuned Small Language Model can achieve high accuracy in CWE detection within Python code, providing a practical and privacy-preserving on-premise security solution. To validate this hypothesis, we focus on fine-tuning the codegen-mono model (350M parameters) [24] using a novel, semi-supervised approach. The key contributions of this paper are as follows:

- We demonstrate the successful instruction-following fine-tuning of codegen-mono for enhanced CWE detection in Python code, achieving performance comparable to or exceeding more resource-intensive methods.
- We introduce a semi-supervised dataset generation methodology, leveraging a reasoning-focused LLM (Gemini-2.0-flash-thinking-exp-01-21) and rigorous manual verification, to create a targeted dataset for security-specific fine-tuning, addressing the challenge of data scarcity in this domain.
- Our experiments showcase the capability of a relatively small model to achieve demonstrably high performance in CWE detection, reaching near-perfect accuracy, precision, recall, and F1-score, highlighting the potential of SLMs for resource-constrained environments.
- This research addresses the critical need for on-premise security solutions by providing a practical and effective approach to leveraging language model technology for CWE detection in environments where data confidentiality is paramount.

The remainder of this paper is structured as follows: Section 2 will delve into related work in CWE detection and the application of language models in software security. Section 3 will detail our methodology for dataset creation and model fine-tuning. Section 4 will present and analyze the experimental results, demonstrating the performance of our approach. Section 5 will discuss the

implications, limitations, and potential future directions of this research. Finally, Section 6 will conclude the paper, summarizing our key findings and their contribution to the field of software security.

## 2 Related Works

Our research intersects with several areas within software engineering and artificial intelligence, primarily Static Analysis Security Testing (SAST), the application of Large Language Models (LLMs) to code security, the use of Small Language Models (SLMs) for code tasks, and synthetic data generation for software engineering.

### 2.1 Traditional Tools

Traditional Static Analysis Security Testing (SAST) tools form a baseline for Python code security, utilizing methods like Abstract Syntax Tree (AST) parsing (e.g., Bandit [11]), flexible pattern matching (e.g., Semgrep [12]), and data/control flow analysis [13]. Beyond static analysis, other techniques contribute, including dynamic analysis frameworks like DynaPyt for runtime checks [14] and specialized machine learning models like BiLSTMs trained for vulnerability detection [15]. However, traditional SAST approaches, in particular, often face challenges with high false positive/negative rates [16] and require continuous, expert-driven maintenance of explicit rules to keep pace with evolving threats. Our work contrasts with these methods by leveraging the emergent pattern-recognition capabilities of a language model fine-tuned on specific vulnerability examples, aiming for high accuracy without reliance on predefined, manually curated rules.

### 2.2 Large Language Models for Code Security

The impressive performance of LLMs, such as OpenAI’s Codex [17], Google’s PaLM variants [18], and GPT-4 [19], has spurred significant interest in their application to software security. Research has shown their potential in tasks like automatically detecting vulnerabilities from code descriptions or raw snippets [20, 21], suggesting fixes for identified issues [22], and even generating security test cases [23]. These studies often highlight the models’ ability to understand code context and semantics better than traditional methods. However, as noted earlier, these powerful models are typically large (billions or trillions of parameters), computationally expensive, and often accessed via APIs, posing practical barriers related to cost, latency, and data privacy for security scanning of proprietary code. Our work specifically addresses these limitations by exploring the capabilities of significantly smaller models.

### 2.3 Small Language Models for Code Tasks

While LLMs grab headlines, there is a growing body of work focusing on Small Language Models (SLMs) models typically under 1 billion parameters tailored for code. Models like CodeGen [24], CodeT5 [25], and smaller variants of StarCoder [26] have demonstrated competence in tasks such as code completion, code summarization, and code translation. These models offer advantages in terms of deployment feasibility and reduced computational cost. However, their application to fine-grained security vulnerability detection, particularly through targeted fine-tuning for specific CWEs, has been less explored compared to their larger counterparts. Our research directly investigates this gap, assessing the extent to which a pre-trained SLM can be specialized for high-accuracy vulnerability detection post-fine-tuning.

## 2.4 Synthetic Data Generation for Code and Security

The performance of data-driven models heavily relies on the quality and quantity of training data. In specialized domains like software security, obtaining large, labelled datasets of real-world vulnerabilities can be challenging. Consequently, researchers have explored using generative models, including LLMs, to create synthetic data. Efforts exist in generating code for general software engineering tasks [27], augmenting existing datasets [28], and specifically generating examples for security training or testing [29]. Our approach aligns with this trend by using an LLM (Gemini-Flash-Exp) to generate paired vulnerable and fixed code snippets. We contribute a specific methodology focused on generating data for targeted CWEs, emphasizing the role of a reasoning-focused generator model and subsequent human validation to ensure data quality for fine-tuning a security-focused SLM.

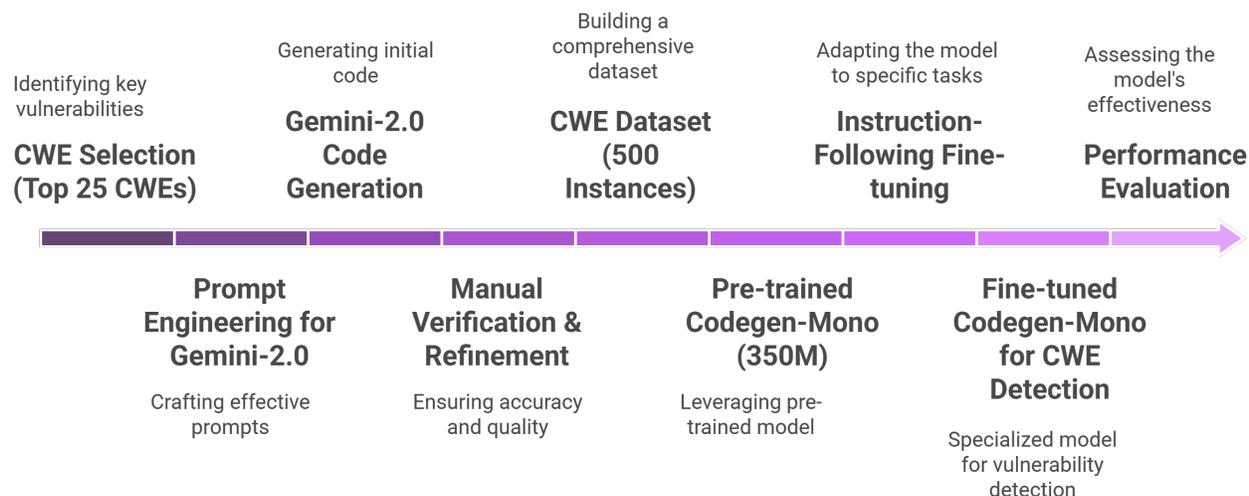


Figure 1: Semi-Supervised Dataset Creation and Fine-Tuning Pipeline

In summary, while extensive research exists on traditional static analysis, machine learning, and large language models for CWE detection, our work uniquely explores fine-tuning small language models for this task. This approach offers a novel, privacy-preserving, and computationally efficient solution, addressing a critical gap in the literature, particularly concerning on-premise deployment for confidential codebases and achieving remarkable performance with a significantly smaller model.

## 3 Methodology

Our approach focuses on fine-tuning a Small Language Model (SLM) to specialize in detecting specific Common Weakness Enumerations (CWEs) within Python code snippets. The methodology encompasses target selection, dataset creation, model selection, and the fine-tuning strategy. The whole process can be visualized in figure 1.

### 3.1 Target Vulnerabilities (CWE Selection)

To ensure practical relevance and focus on high-impact issues, we selected the MITRE Top 25 Most Dangerous Software Weaknesses list [30] as the target for our detection model. This list represents common and critical vulnerabilities encountered in real-world systems, making it a suitable starting point for evaluating the feasibility of SLM-based detection.

### 3.2 Dataset Curation: Semi-Supervised Synthetic Data Generation

Creating a sufficiently large and accurately labelled dataset for vulnerability detection is often a bottleneck. To address this, we employed a semi-supervised approach combining automated generation with human oversight:

- **Generation Tool:** We utilized Google’s gemini-2.0-flash-thinking-exp-01-21 model via its API . This model was chosen for its reported reasoning capabilities, which we deemed beneficial for generating plausible code exhibiting specific logical flaws corresponding to CWEs.
- **Generation Process:** For each of the 25 selected CWEs, we iteratively refined prompts to instruct the Gemini model to generate: (a) Five distinct Python code snippets, each realistically demonstrating the specific CWE vulnerability. (b) For each vulnerable snippet, a corresponding "counter-example" snippet where the underlying issue causing the CWE was addressed and fixed.
- **Prompt Engineering:** Significant effort was invested in prompt engineering to guide the generator model towards producing code that reflects plausible real-world programming patterns, rather than trivial or overly simplistic examples.
- **Human Review:** Crucially, all generated code snippets (both vulnerable and secure) were manually reviewed by the authors to verify: (a) The correctness of the vulnerability classification (i.e., the vulnerable code actually exhibits the intended CWE). (b) The validity of the fix (i.e., the counter-example correctly addresses the vulnerability without introducing others). (c) The overall realism and relevance of the code examples. This human validation step was essential to ensure the quality and reliability of the training data.
- **Final Dataset:** This process resulted in a dataset of 500 labelled instances (25 CWEs  $\times$  10 examples  $\times$  [1 vulnerable + 1 fixed]). Each instance was formatted according to a standard instruction-following structure: Instruction, Input, Output.

Instruction	Input	Output
Analyze the given code and check if it contains any known security flaws.	<pre>import flask app = flask.Flask(__name__) @app.route('/set_discount', methods=['POST']) def set_discount():     discount = flask.request.form.get('discount') try:     discount_percent = int(discount)     return f'Discount set to {discount_percent}%' except ValueError:     return 'Invalid discount value' if __name__ == '__main__':     app.run(debug=True)</pre>	CWE-ID352:Cross-Site Request Forgery (CSRF)

Figure 2: Dataset Example

### 3.3 Model Selection

We selected the codegen-mono 350M model [24] as our base SLM. This model was chosen because: (a) It is a publicly available, pre-trained model specifically designed for code-related tasks. (b) Its

Table 1: Summary of Prior Works on Vulnerability Detection (ML/LLM Focus)

Reference	Technique Used	Reported Performance	Remarks
Farasat & Posegga [15]	ML (BiLSTM)	Python: Avg Acc=98.6%, F1=94.7%, Prec=96.2%, Rec=93.3%, ROC=99.3%.	High performance on specific Python vulnerability detection task using ML.
Bagheri et al. [31]	Hybrid ML (Self-attention + CNN - Conformer)	F1 score of 99% reported	High F1 reported, potentially domain-specific.
Singh et al. [32]	ML (Logistic Regression)	CWE Prediction: Acc=0.66, Prec=0.65, Rec=0.66, F1=0.64.	Moderate performance using simpler ML for CWE prediction.
Mechri et al. [33]	LLM Prompting (Chain-of-Thought)	Qualitative increase in F1 mentioned for CoT prompts on real vulns; no specific Python metrics.	Highlights prompt engineering benefits, lacks quantitative Python data.
Dozono et al. [6]	LLM Prompting (Various strategies)	Python F1: GPT-4o=0.80, GPT-4T=0.76, Gemini 1.5 Pro=0.75, CodeLlama-7b=0.72, GPT-3.5T=0.70, CodeLlama-13b=0.35.	Compares various LLMs for Python, shows GPT-4 variants leading.
Steenhoek et al. [34]	LLM Prompting (zero/n-shot, CoT variants)	Evaluated on C/C++: Low Balanced Accuracy (54.5%); Python N/A.	Concludes current LLMs perform poorly; common prompting strategies ineffective (on C/C++).
Shestov et al. [35]	LLM Fine Tuning for JAVA CWE detection	Best F1 @ .86 for binary classification using WizardCoder	N/A
Li et al. [36]	LoRa and IA3 Fine tuning approach for LLMs	5-6% improvement over base model	9 CWE was tested using a 2.7b model, codegen.
Jiang et al. [37]	LLM Fine Tuning using LoRa approach	Best F1 achieved using Llama 2-7b model @ 87%	N/A
<i>This Work</i>	Instruction Following Fine Tuning, 350m parameters model	Accuracy: 99%, Precision: 98.08%, Recall: 100%, F1 score: <b>99.04%</b>	High performance on Python CWEs with SLM fine-tuning.

350 million parameter size places it firmly in the "small" language model category, making it suitable for exploring feasibility for on-premise deployment and efficient fine-tuning. (c) Its focus on code generation/understanding provides a relevant foundation for the downstream task of vulnerability detection.

### 3.4 Fine-tuning Strategy: Instruction Following

We employed an Instruction-Following Fine-tuning approach to adapt the pre-trained codegen-mono model to our specific CWE detection task.

- **Data Format:** The dataset was structured with three fields per instance: *Instruction:* A directive telling the model what task to perform. *Input:* The Python code snippet to be analyzed. *Output:* The expected label (e.g., "Vulnerable - CWE-XXX" or "Secure"). An example of data instance is presented in figure 2.
- **Consistent Instruction:** Through experimentation with various instruction phrasings and strategies (including varying instructions per row), we found that using a single, consistent instruction across all training examples yielded the best performance. This same instruction was subsequently used during the evaluation/inference phase. In our case we kept the instruction depicted in figure 2 for all the rows as well as during interface.
- **Training Process:** The model was fine-tuned using this structured dataset. We iterated through different hyperparameters (learning rate, batch size, epochs) and instruction formats to optimize performance, leading to the final configuration reported in Section 4. The objective was to train the model to accurately predict the Output label given the Instruction and Input code snippet.

## 4 Results and Discussion

### 4.1 Baseline Performance: Un-tuned Codegen-Mono

Prior to fine-tuning, we assessed the zero-shot performance of the base codegen-mono model on a randomly selected subset of 100 examples from our CWE dataset. Strikingly, in this baseline evaluation, the un-tuned codegen-mono model failed to detect a single CWE within any of the code snippets presented. This indicates that while pre-trained on a large corpus of code, the base model lacks the specific knowledge and instruction-following capabilities necessary for accurate CWE identification, at least in a zero-shot setting and without task-specific fine-tuning. This starkly underscores the need for targeted fine-tuning to adapt such models for specialized security tasks like CWE detection.

Table 2: Performance Metrics of Fine-tuned Model

Metric	Value
Accuracy	99%
Precision	≈ 98.08%
Recall	100%
F1-Score	≈ 99.04%

## 4.2 Performance of Fine-tuned Codegen-Mono

After instruction-following fine-tuning on our CWE dataset, the performance of codegen-mono underwent a dramatic transformation. We evaluated the fine-tuned model on a held-out test set comprising 100 instances, and the results demonstrate a remarkable level of accuracy in CWE detection. Table 2 summarizes the key performance metrics achieved by the fine-tuned model. As evident from Table 2, the fine-tuned codegen-mono model achieved near-perfect accuracy of 99% on the CWE detection task. This high accuracy is further reinforced by a precision of approximately 98.08%, indicating that out of all instances identified as containing CWEs, the model was correct in the vast majority of cases. Furthermore, the model achieved a perfect recall of 100%, signifying that it successfully detected all instances of CWEs present in the test set. The resulting F1-score of approximately 99.04%, which harmonically balances precision and recall, confirms the overall exceptional performance of the fine-tuned model.

## 4.3 Hardware performance metrics

Performance metrics were evaluated on a desktop system equipped with a 13th Gen Intel(R) Core(TM) i5-13500 CPU (2.5 GHz, 14 Cores, 20 Threads) and 15.6 GB RAM, without GPU acceleration. Key inference timing results for the fine-tuned 350M model are summarized in Table 3. The model performed reasonably well for real-time application on a moderate hardware. These results represent a substantial and statistically significant improvement compared to the baseline performance of the un-tuned model. The fine-tuned codegen-mono has demonstrably acquired a strong capability for accurately identifying CWEs in Python code through instruction-following fine-tuning using our specifically curated dataset. This highlights the effectiveness of our approach and the potential of even small language models, when appropriately fine-tuned, to deliver high-performance solutions for specialized security tasks like on-premise CWE detection. Table 1 summarizes the current trends among the researchers for CWE detection.

Table 3: CPU Inference Timing Metrics (codegen-mono 350M)

Metric	Value
Time to First Token (TTFT)	0.253 seconds
Tokens per Second (TPS)	6.01 tokens/sec
Median Latency (P50)	0.165 seconds
P95 Latency	0.182 seconds
P99 Latency	0.239 seconds

## 5 Limitations and Future Work

Our study, while demonstrating the potential of fine-tuned SLMs for CWE detection, has limitations. Its scope was confined to the MITRE Top 25 CWEs in Python, using a modest-sized dataset of synthetic snippets. This restricts known applicability and raises questions about generalization to real-world code complexity, other vulnerabilities, different languages, or vulnerabilities spanning multiple files. The model’s performance on independent benchmarks and its inherent explainability also remain open questions.

Future work should directly address these limitations. Key priorities include expanding the scope to more CWEs and languages, enriching datasets with real-world examples, and improving model

functionality towards localization and fix suggestions. Exploring alternative SLM architectures, advanced fine-tuning methods, and conducting rigorous comparative benchmarks against SAST tools and LLMs are also crucial next steps.

Furthermore, investigating model explainability techniques and piloting the integration into real-world development environments will be vital for assessing practical utility and fostering adoption. These efforts will help determine the true extent to which fine-tuned SLMs can serve as robust, trustworthy components in the software security toolkit.

## 6 Conclusion

In this paper, we have demonstrated the successful application of instruction-following fine-tuning to adapt a small language model, codegen-mono, for high-accuracy Common Weakness Enumeration (CWE) detection in Python code in a reasonable hardware. Our results show a remarkable transformation from a baseline model incapable of detecting CWEs to a fine-tuned model achieving near-perfect performance metrics. This research underscores the significant potential of fine-tuned Small Language Models as a practical, resource-efficient, and privacy-preserving solution for on-premise code security analysis, particularly for organizations handling confidential codebases. By providing a viable alternative to cloud-dependent LLM security tools, our work paves the way for broader adoption of advanced language model technology in security-sensitive environments, contributing to more secure software development practices and reduced vulnerability risks in critical applications. We encourage future research to build upon these findings by exploring larger and more diverse datasets, extending the approach to other programming languages, and developing robust, real-world CWE detection tools based on fine-tuned Small Language Models.

## Data Availability

The dataset can be found at [https://huggingface.co/datasets/floxihunter/synthetic\\_python\\_cwe](https://huggingface.co/datasets/floxihunter/synthetic_python_cwe) The tuned model can be found at <https://huggingface.co/floxihunter/codegen-mono-CWEdetect>

## References

- [1] H. Mittal. Software security: Threats, solutions and challenges. *Comput. Softw. Media Appl.*, 6(1):3769, Feb. 2024. doi: 10.24294/csma.v6i1.3769.
- [2] R. A. Martin and S. Barnum. Common weakness enumeration (cwe) status update. *ACM SIGAda Ada Lett.*, XXVIII(1):88–91, Apr. 2008. doi: 10.1145/1387830.1387835.
- [3] A. Bagnato. Security in model-driven architecture. In *Proceedings on European Workshop on Security in Model Driven Architecture*, 2009.
- [4] D. Nikolic, D. Stefanovic, D. Dakic, S. Sladojevic, and S. Ristic. Analysis of the tools for static code analysis. In *2021 20th International Symposium INFOTEH-JAHORINA (INFOTEH)*, pages 1–6, East Sarajevo, Bosnia and Herzegovina, Mar. 2021. IEEE. doi: 10.1109/INFOTEH51037.2021.9400688.
- [5] M. Vieira. Engineering trustworthy software: A mission for llms. *arXiv preprint arXiv:2411.17981*, 2024. doi: 10.48550/ARXIV.2411.17981.

- [6] K. Dozono, T. E. Gasiba, and A. Stocco. Large language models for secure code assessment: A multi-language empirical study. *arXiv preprint arXiv:2408.06428*, 2024. doi: 10.48550/ARXIV.2408.06428.
- [7] M. Kazim and S. Ying. A survey on top security threats in cloud computing. *Int. J. Adv. Comput. Sci. Appl.*, 6(3), 2015. doi: 10.14569/IJACSA.2015.060316.
- [8] N. Sehrawat, S. Vashisht, and N. Kaur. Edge-computing paradigm: Survey and analysis on security threads. In *2021 International Conference on Computing Sciences (ICCS)*, pages 254–259, Phagwara, India, Dec. 2021. IEEE. doi: 10.1109/ICCS54944.2021.00057.
- [9] P. G. Recasens et al. Towards pareto optimal throughput in small language model serving. In *Proceedings of the 4th Workshop on Machine Learning and Systems*, pages 144–152, Athens Greece, Apr. 2024. ACM. doi: 10.1145/3642970.3655832.
- [10] E. Nijkamp et al. Codegen: An open large language model for code with multi-turn program synthesis. *arXiv preprint arXiv:2203.13474*, Feb. 2023. doi: 10.48550/arXiv.2203.13474.
- [11] J. White. *Bandit algorithms for website optimization*. O’Reilly Media, Inc., 2013.
- [12] G. Bennett, T. Hall, E. Winter, and S. Counsell. Semgrep\*: Improving the limited performance of static application security testing (sast) tools. In *Proceedings of the 28th International Conference on Evaluation and Assessment in Software Engineering*, pages 614–623, 2024.
- [13] A. H. Jerónimo, P. M. Moreno, J. A. V. Camacho, and G. C. Vega. Techniques of sast tools in the early stages of secure software development: A systematic literature review. In *2024 IEEE International Conference on Engineering Veracruz (ICEV)*, pages 1–8. IEEE, 2024.
- [14] A. Eghbali and M. Pradel. Dynapyt: a dynamic analysis framework for python. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2022.
- [15] T. Farasat and J. Posegga. Machine learning techniques for python source code vulnerability detection. In *Proceedings of the Fourteenth ACM Conference on Data and Application Security and Privacy (CODASPY ’24)*, pages 151–153, Jun. 2024. ACM.
- [16] W. Charoenwet, P. Thongtanunam, V.-T. Pham, and C. Treude. An empirical study of static analysis tools for secure code review. *arXiv preprint arXiv:2407.12241 [cs.SE]*, 2024.
- [17] A. Kumar and P. Sharma. Open ai codex: An inevitable future? *International Journal for Research in Applied Science and Engineering Technology*, 11:539–543, 2023.
- [18] A. Maddy. Integrating ai services into semantic kernel: A case study on enhancing functionality with google palm and large language models. *Transactions on Open Source Software Projects*, 1(1), 2024.
- [19] J. Achiam et al. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774*, 2023.
- [20] A. A. Mahyari. Harnessing the power of llms in source code vulnerability detection. In *MILCOM 2024 - 2024 IEEE Military Communications Conference (MILCOM)*, pages 251–256, Oct. 2024. IEEE.
- [21] X. Du et al. Vul-rag: Enhancing llm-based vulnerability detection via knowledge-level rag. *arXiv preprint arXiv:2406.13749*, 2024.

- [22] N. T. Islam et al. Llm-powered code vulnerability repair with reinforcement learning and semantic reward. *arXiv preprint arXiv:2401.03374*, 2024.
- [23] Y. Zhang, W. Song, Z. Ji, N. Meng, et al. How well does llm generate security tests? *arXiv preprint arXiv:2310.00710*, 2023.
- [24] E. Nijkamp et al. Codegen: An open large language model for code with multi-turn program synthesis. *arXiv preprint arXiv:2203.13474*, 2022.
- [25] Y. Wang, W. Wang, S. Joty, and S. C. Hoi. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. *arXiv preprint arXiv:2109.00859*, 2021.
- [26] R. Li et al. Starcoder: may the source be with you! *arXiv preprint arXiv:2305.06161*, 2023.
- [27] W. Murphy et al. Combining llm code generation with formal specifications and reactive program synthesis. *arXiv preprint arXiv:2410.19736*, 2024.
- [28] S. Ugare, T. Suresh, H. Kang, S. Misailovic, and G. Singh. Syncode: Llm generation with grammar augmentation. *arXiv preprint arXiv:2403.01632*, 2024.
- [29] J. Leinonen, P. Denny, O. Kiljunen, S. MacNeil, S. Sarsa, and A. Hellas. Llm-itation is the sincerest form of data: Generating synthetic buggy code submissions for computing education. *arXiv preprint arXiv:2404.01898*, 2024.
- [30] MITRE CWE. CWE - CWE Top 25 Most Dangerous Software Weaknesses. <https://cwe.mitre.org/top25/>. Accessed: 15-Apr-2025.
- [31] A. Bagheri and P. Hegedűs. Towards a block-level conformer-based python vulnerability detection. *Software*, 3(3):310–327, 2024. MDPI.
- [32] S. Singh, S. Jeevan, N. Reddy, M. Moharir, et al. Temporal analysis and common weakness enumeration (cwe) code prediction for software vulnerabilities using machine learning. In *2024 8th International Conference on Computational System and Information Technology for Sustainable Solutions (CSITSS)*, pages 1–6. IEEE, 2024.
- [33] A. Mechri, M. A. Ferrag, and M. Debbah. Secureqwen: Leveraging llms for vulnerability detection in python codebases. *Computers & Security*, 148:104151, 2025. Elsevier.
- [34] B. Steenhoek et al. To err is machine: Vulnerability detection challenges llm reasoning. *arXiv preprint arXiv:2403.17218*, 2024.
- [35] A. Shestov et al. Finetuning large language models for vulnerability detection. *IEEE Access*, 2025. IEEE.
- [36] J. Li, F. Rabbi, C. Cheng, A. Sangalay, Y. Tian, and J. Yang. An exploratory study on finetuning large language models for secure code generation. *arXiv preprint arXiv:2408.09078*, 2024.
- [37] X. Jiang et al. Investigating large language models for code vulnerability detection: An experimental study. *arXiv preprint arXiv:2412.18260*, 2024.