Securing AI-Generated Code

Andreas E. Nelson
nel02254@morris.umn.edu
Division of Science and Mathematics
University of Minnesota, Morris
Morris, Minnesota, USA

Abstract

The increasing use of AI for code generation presents significant security challenges, as these tools often lack inherent security awareness and can produce vulnerable code. This paper investigates these security risks, outlining common types of vulnerabilities (such as injection flaws and improper resource handling) found in AI-generated code. It further explores and evaluates mitigation techniques aimed at improving code security, including model fine-tuning and adversarial strategies like Security Verifier Enhanced Neural Steering (SVEN). Findings indicate that while current methods offer promising ways to reduce vulnerabilities, ongoing research and development are crucial for the secure and responsible deployment of AI in software development.

Keywords: Large Language Models, AI Code Generation, Security Risks, Secure Coding Practices, Adversarial Testing.

1 Introduction

The landscape of software development is rapidly evolving with the advent of powerful Artificial Intelligence (AI) tools capable of generating functional code. Driven by advancements in deep learning, particularly Large Language Models (LLMs), tools like GitHub Copilot, Amazon CodeWhisperer, and Codeium have gained significant popularity [1, 2]. These AI pair programmers promise substantial benefits, offering potential increases in developer productivity by automating repetitive coding tasks, suggesting code completions, and even generating entire code blocks based on natural language descriptions [1, 4]. However, this technological advancement presents a significant challenge: the security risk of the generated code. This paper focuses primarily on vulnerabilities detectable by automated scanning tools, while acknowledging that other types of security issues, potentially requiring deeper semantic understanding, might also arise in AI-generated code and represent an area for future investigation.

LLMs are typically trained on vast datasets comprising billions of lines of code scraped from public repositories like GitHub [4]. While this allows them to learn diverse coding patterns, it also means they inadvertently learn and replicate security vulnerabilities present in their training data [2, 4]. Consequently, the code generated by these AI tools may contain common weaknesses, potentially introducing critical security flaws into software projects at scale. Empirical studies confirm this risk. For instance, Fu et al. analyzed code snippets identified as potentially AI-generated within GitHub projects (identification methodology involved searching for specific keywords and patterns associated with AI tool usage, though review status by human programmers was not definitively determined), finding a notable percentage (approximately 27-30%) contained security weaknesses [2]. The widespread adoption of these tools raises concerns about the potential for propagating vulnerabilities across the software ecosystem, making the security assessment of AI-generated code a critical area of research [2].

This paper provides a roadmap for understanding and addressing these security challenges. We begin by providing essential background on the underlying technologies, namely Neural Networks and the Transformer architecture powering modern LLMs (Section 2). We then explore the specific security risks and common vulnerability types observed in code generated by these models, drawing on recent empirical studies (Section 3). Subsequently, we examine current strategies aimed at mitigating these risks. These include fine-tuning pre-trained models on security-focused datasets using real-world vulnerability fixes (Section 4). This technique has been shown to improve secure code generation, with studies demonstrating notable improvements for C, although results varied for C++ potentially due to language-specific factors impacting vulnerability patterns and fine-tuning effectiveness [6]. We also examine employing specialized security hardening techniques such as adversarial training and output steering mechanisms like Security Verifier Enhanced Neural Steering (SVEN) (Section 5). SVEN not only enabled effective adversarial testing but also significantly increased secure code generation rates (e.g., from 59.1% to 92.3% in some settings) [4]. Finally, we conclude by summarizing the key challenges and future directions for ensuring the safe and responsible use of AI in code generation (Section 6). Understanding both the capabilities and the security pitfalls of these tools is crucial for navigating their integration into modern software development practices effectively.

2 Background: Neural Networks and Large Language Models

To understand the security risks of AI-generated code, it's essential to grasp the fundamentals of the underlying technologies: *Neural Networks (NNs)* and, more specifically, the

Transformer architecture which is often used to build *Large Language Models (LLMs)* [8].

2.1 Neural Network Fundamentals

Neural Networks (NNs) are computational models inspired by the structure and function of biological brains, representing a model for information processing [5]. They consist of interconnected nodes, or "neurons," organized in layers: an input layer designed to receive information, one or more layers of hidden units, and an output layer that signals the network's response [5]. Each connection between neurons carries a real-valued weight, determining the strength and influence (excitatory or inhibitory) of the signal passed between them [5]. Information flows through the network, starting at the input layer, passing through the hidden layers where computations involving weighted sums and activation functions occur, and finally reaching the output layer to produce the result (e.g., a prediction, classification, or generated code) [5]. Common activation functions include the *sigmoid function*, often used to introduce non-linearity [5].

The core of NN learning involves adjusting these connection weights so the network can efficiently perform a task, often by learning from examples or input-output relations [5]. This adjustment is typically achieved through minimizing errors using a process called *backpropagation* [7]. During training, the network makes a prediction based on its current weights. This prediction is compared to the correct target output, and an error value is calculated using a *loss function*, which measures how incorrect the prediction was [7]. Backpropagation then efficiently calculates the *gradient* (the direction and magnitude of the error's sensitivity to each weight) of the loss function with respect to each weight using the chain rule [5, 7]. These gradients guide the update of the weights, typically via an optimization algorithm like *gradient descent*, iteratively reducing the error and improving the network's performance on the training data [5, 7].

Deep Learning involves NNs with multiple hidden layers (deep architectures) [7]. These numerous layers enable the networks to learn complex patterns and hierarchies of features from large datasets, making them powerful for tasks requiring sophisticated data representations [7]. Various types of NNs exist, including simple *Feed Forward Neural Networks* where data travels in one direction [5], *Recurrent Neural Networks (RNNs)* designed to handle sequences by incorporating memory of previous steps [3, 5], and *Convolutional Neural Networks* which use convolutional filters and are particularly effective for representing spatial relationships and structures (hierarchies) like those in images [5, 7].

2.2 Transformer Architecture and Large Language Models

Most current state-of-the-art AI code generation tools are based on *Large Language Models (LLMs)*. LLMs are a type of deep learning model characterized by their massive size (often billions of parameters or weights) and their training on vast quantities of text and code data scraped from sources like GitHub [4]. This extensive training allows them to understand and generate human-like text and functional code across various programming languages [3, 4]. These models typically undergo a common training process involving unsupervised pre-training on raw data to learn general representations, subsequently followed by fine-tuning for specific downstream tasks [3].

A key innovation enabling the creation of modern LLMs is the *transformer architecture*, introduced by Vaswani et al. [3, 8]. The transformer architecture processes sequences of input data (e.g., natural language text or code tokens) and produces sequences of output data (e.g., translated text or generated code) [8]. Unlike earlier sequential models like RNNs (which struggled with parallelization and processing very long sequences partly due to *vanishing gradients*, where error signals diminish exponentially as they propagate back through time, making it hard to learn long-range dependencies [3, 7]), transformers process input data in parallel and rely solely on *attention mechanisms* [3]. The attention mechanism allows the model, when generating a specific word or code token, to dynamically weigh the importance of all other tokens in the input sequence [3]. It does this by mapping a *query* (representing the current focus) and a set of *key-value (K-V) pairs* (representing all input tokens and their associated information) to an output [3]. It can "pay attention" to the most relevant parts of the input, regardless of their distance within the sequence [3]. This mechanism is vital for understanding context, handling long-range dependencies in text and code (e.g., matching brackets, tracking variable usage), and ultimately generating coherent and contextually appropriate output [3, 4]. This ability to capture complex, long-range relationships within code structures is what makes transformer-based LLMs particularly adept at code generation tasks [1, 4].

3 Security Risks in AI-Generated Code

While powerful, LLMs trained on vast datasets, including code from public repositories like GitHub, inadvertently learn and replicate security vulnerabilities present in that data [2, 4]. Key code-generation tools include GitHub Copilot (GitHub/Microsoft/OpenAI), Amazon CodeWhisperer, and Codeium [2]. The sheer scale and heterogeneity of this training data make it currently impractical to curate a sufficiently large, vulnerability-free dataset for training general-purpose code-generation models. LLMs often lack inherent security awareness [4]. Empirical studies have confirmed the risks; one evaluation by Pearce et al. discovered that, in various security-relevant scenarios, 40% of Copilot-generated programs contained dangerous vulnerabilities, with other state-of-the-art LLMs found to have similarly concerning security levels [4]. Another study found ChatGPT often generates

code below minimal security standards [4]. A large-scale empirical study by Fu et al. analyzing code generated by Copilot, CodeWhisperer, and Codeium found within GitHub projects provides further evidence [2]. Their methodology involved searching for keywords and patterns associated with AI tool usage to identify potentially generated code, but did not definitively determine if the code was subsequently reviewed or modified by human developers [2]. Analyzing 733 distinct code files identified as likely containing AI-generated code, they found security weaknesses in 29.5% of Python snippets and 24.2% of JavaScript snippets [2]. Overall, 200 of the 733 snippets (27.3%) contained at least one security weakness, highlighting the significant frequency of insecure code integration in real-world open-source development [2].

3.1 Nature and Types of Vulnerabilities

The diverse security issues in AI-generated code reflect the insecure patterns learned from vast training datasets, including extensive public code from repositories like GitHub [2, 4]. Studies have shown that LLMs can produce code susceptible to a range of *Common Weakness Enumeration (CWE)* types – a standardized classification system for software weaknesses [4, 6].

The study by Fu et al. identified a total of 628 distinct security weaknesses spanning 43 different CWE types within the 200 vulnerable code snippets analyzed [2]. This variety underscores the breadth of potential security flaws developers might encounter. Among the most frequently occurring issues identified by Fu et al. were *CWE-330 (Use of Insufficiently Random Values)*, representing 18.15% of identified weaknesses, followed by *CWE-94 (Improper Control of Generation of Code - 'Code Injection')*, *CWE-79 (Cross-site Scripting - XSS)*, and *CWE-78 (OS Command Injection)* [2]. Other notable examples include Path Traversal (CWE-22), Use of Uninitialized Variable (CWE-457), Missing Release of Resource (CWE-772), and SQL Injection (CWE-89) [2]. The high prevalence of these specific CWEs suggests common pitfalls in how LLMs handle randomness, code execution based on input, web interactions, and database queries. Crucially, eight of the 43 CWE types identified by Fu et al. were listed in the 2023 CWE Top-25 Most Dangerous Software Weaknesses list, accounting for 233 (37.1%) of the total security weaknesses found, emphasizing that AI tools are propagating some of the most critical and widespread vulnerabilities [2].

3.2 Factors Influencing Vulnerabilities

Several factors contribute to the generation of insecure code. The sheer complexity of secure coding practices means that even human developers make mistakes, and LLMs trained on human code inherit these tendencies [1]. The vast scale of training data scraped from public repositories inevitably includes insecure patterns [2, 4]. The specific programming language or domain can also influence risk [2, 6]. For instance, Fu et al. observed that Python weaknesses often related to system calls (like OS Command Injection) and data processing, while JavaScript issues more commonly involved dynamic code-generation (Code Injection) and web security flaws (XSS), reflecting the typical usage contexts of these languages [2]. This difference might be linked to language features like dynamic typing, which can make static analysis harder and potentially simplify insecure code-generation if the model doesn't need to reason about the entire program flow [2]. Application domain also matters; Fu et al. found Web Applications contributed the most CWEs in JavaScript, while Utility Tools were the largest source in Python [2]. Furthermore, the prompts given to the LLM can significantly impact the security of the output; vague or poorly defined prompts are more likely to result in insecure code [1, 4]. User studies also suggest task complexity might play a role; Asare et al. found Copilot assistance correlated with more secure solutions for harder problems, potentially by providing established secure patterns, but observed no difference for simpler tasks [1].

3.3 Evaluating Security Risks

Evaluating the security of LLM-generated code is crucial for understanding and mitigating risks. Researchers determine vulnerability rates by employing a combination of methods, often starting with automated scanning and refining results through manual checks. *Static Analysis* tools like CodeQL, a semantic code analysis engine developed by GitHub, are frequently used to automatically scan generated code for patterns matching known vulnerability types (CWEs) across specific, predefined scenarios; these tools analyze the code structure without executing it [2, 6]. Since automated tools may produce false positives or miss certain vulnerability types (especially logical flaws), *Manual Code Review* by security experts is often necessary to confirm findings and assess the severity of potential issues [1, 2, 4], serving as a key step in studies like Fu et al. to filter static analysis results [2]. Additionally, *benchmarking* helps quantify risks and track improvements by comparing vulnerability rates across different models and conditions (e.g., with/without security prompts), or against human-written code [1, 4, 6].

For example, a study by He et al. evaluated an LLM on a scenario designed to elicit a Null Pointer Dereference (CWE-476) [4]. The prompt could ask the model to allocate memory and then use it, like copying data from standard input into a newly allocated buffer [4]. A vulnerable response might use `malloc` but fail to check if the returned pointer (`buf`) is `NULL` before attempting to read from standard input into `buf` using `fgets`, potentially leading to a crash if `malloc` failed [4]. This reflects a common pattern where basic safety checks are overlooked [4]. A static analysis tool like CodeQL could detect this potential dereference of a possibly null pointer [4]. Comparing the frequency of such vulnerable generations

across different models or techniques provides a measure of their relative security [4].

4 Mitigation Strategy: Fine-Tuning for Security

One promising approach to reduce vulnerabilities in AI-generated code, without requiring complex architectural changes, is *fine-tuning* [6]. This involves taking a general-purpose pre-trained LLM and further training it on a smaller, curated dataset specifically designed to teach secure coding practices [6]. The goal is to adapt the LLM's learned patterns to favor secure code generation, effectively learning to generate secure code directly from prompts or transform potentially unsafe patterns into safer equivalents [6].

4.1 Fine-Tuning Methodology

The process typically involves several key steps:

1. **Creating a Security-Focused Dataset:** This requires compiling a specialized dataset. A key approach by Li et al. is using real-world vulnerability fixes [6]. They collected 4900 vulnerability-fixed commits across 580 open-source projects, extracting paired examples of insecure code and their corresponding secure versions from established vulnerability datasets [6]. This dataset contained over 14,000 source files (primarily C/C++, distinct from the Python/JavaScript focus in Section 3) and aimed to expose the model to practical examples of fixing vulnerabilities and implementing secure coding patterns [6]. Compiling such datasets focuses on specific vulnerability types (e.g., CWEs) relevant to the target programming languages [6].
2. **Model Selection and Training:** A pre-trained LLM is chosen for fine-tuning. Li et al. selected GPT-J, a smaller model in the GPT family, with comparable code generation performance to the larger Codex model while being more computationally feasible for academic research and fine-tuning [6]. The chosen model is then trained (weights adjusted via backpropagation) on the specialized security dataset to better predict or generate the secure code examples [6].
3. **Evaluation Setup:** Assessing the effectiveness requires a dedicated evaluation framework. This often involves crafting a dataset of code generation scenarios focusing on specific, high-risk CWEs (such as those from the "CWE Top 25 Most Dangerous Software Weaknesses" list [6]). These scenarios typically include a natural language description and an incomplete code snippet for the model to complete, targeting languages like C and C++ [6].
4. **Vulnerability Detection:** Static analysis tools are frequently used to evaluate the generated code. Tools like CodeQL, a popular semantic code analysis engine, automatically detect potential vulnerabilities in the model's output for the test scenarios [2, 6]. The rates of vulnerable vs. non-vulnerable code generated by the fine-tuned model are then compared to the original pre-trained model [6].

4.2 Fine-Tuning Results and Implications

Studies by Li et al. [6] and He et al. [4] have shown that fine-tuning can significantly reduce the generation of insecure code for specific vulnerability types targeted during training. The experiments by Li et al. [6] on GPT-J fine-tuned with vulnerability fixes yielded a notable 10% absolute increase (from 60% to 70.4%) in the generation of non-vulnerable code for C scenarios. A slight increase (from 63.9% to 64.5%) was observed for C++ [6]. Manual examination suggested the fine-tuned model learned specific secure behaviors present in the fix data; for example, in scenarios related to out-of-bounds array access (CWE-125), the fine-tuned model consistently implemented proper boundary checks before accessing array elements, mirroring patterns observed in the vulnerability fixes used for training [6]. Related work also suggests models fine-tuned on datasets emphasizing secure handling of web inputs showed a marked decrease in generating code vulnerable to XSS [4].

However, fine-tuning presents challenges and limitations. It is often specific; improving security for one CWE might not generalize well to others, as evidenced by differing improvement rates across languages (C vs C++ in the Li et al. study) [6]. Careful dataset creation is crucial and labor-intensive, and the process requires significant computational resources, although choices like using GPT-J can make it more feasible compared to training larger models from scratch [6]. Furthermore, Li et al. [6] observed a higher rate of syntactically invalid code generation in C++ scenarios for the fine-tuned model, suggesting potential trade-offs between enforcing security patterns and maintaining code correctness or fluency (the study primarily focused on syntax, but functional correctness could also be impacted) [6]. While effective for targeted improvements, fine-tuning alone may not eliminate all risks, and its success depends heavily on the quality, coverage, and representativeness of the fine-tuning dataset [6].

5 Mitigation Strategy: Security Hardening Techniques

Beyond fine-tuning the model's weights, other techniques aim to harden large language models (LLMs) against generating insecure code, often by influencing the generation process directly or by using adversarial methods to improve robustness and evaluation [4]. These methods are particularly relevant given the significant cost associated with retraining or fully fine-tuning massive foundation models [4].

5.1 Adversarial Methods for Security Evaluation and Hardening

Adversarial methods leverage the concept of an "adversary," which can be another model or a crafted process, attempting to expose weaknesses in the target LLM or, conversely, to improve its defenses through targeted training [4]. Two key directions exist. Firstly, *Adversarial Testing* focuses on evaluating an LLM's security posture from an adversarial standpoint. This involves crafting specific prompts or inputs designed to intentionally induce the LLM into generating insecure code [4]. The goal is often to deliberately degrade the LLM's security performance to understand its vulnerabilities under attack scenarios and identify weaknesses missed by standard benchmarks or functional testing [4]. This provides a stress test of the model's security awareness [4]. Secondly, *adversarial training* aims to improve model robustness by incorporating adversarial examples directly into the training process [4]. Inputs known or likely to cause the target LLM to produce insecure code (identified perhaps by an auxiliary adversarial model or specific heuristics designed to probe weaknesses) are included alongside standard training data [4]. The target LLM learns to recognize and resist generating insecure code even when presented with these challenging or potentially malicious inputs [4]. This process can involve iterative refinement, where the adversary gets better at finding weaknesses and the target model gets better at defending against them, minimizing susceptibility [4]. Studies show adversarial training can improve robustness against specific types of adversarial prompts [4].

5.2 Steering LLMs with Verifiers and Prefixes (SVEN)

A distinct and novel approach, exemplified by Security Verifier Enhanced Neural Steering (SVEN), aims to guide the output of an existing, unmodified LLM towards desired properties like security without retraining or fine-tuning its core parameters (i.e., without changing the weights of the large, pre-trained model) [4]. This directly addresses the challenge of *modularity* – the prohibitive expense and difficulty of modifying massive, pre-trained foundation LLMs [4].

SVEN formulates security hardening and adversarial testing as a *controlled code generation* task: the LLM receives not just the standard prompt but also an additional binary property (e.g., "generate secure code" or "generate unsafe code") that guides the generation process towards the specified property, while aiming to preserve the LLM's ability to generate functionally correct code [4].

SVEN achieves this control using learnable "prefixes" – small, *property-specific continuous vectors* (sequences of numbers optimized during training) that are prepended to the LLM's internal state before processing the user prompt [4]. These prefixes effectively encode the desired property (like 'security' or 'vulnerability'). They are optimized using specialized loss functions applied to different code regions (neutral vs. security-sensitive) within a high-quality, manually curated dataset derived from sources like CrossVul, BigVul, and VUDENC (large datasets containing vulnerability information and associated code fixes from open-source projects) after careful filtering to remove non-security fixes and project-specific code [4]. The loss function rewards the model for producing outputs similar to the original model in non-security-sensitive regions, helping preserve functionality [4]. Crucially, during this prefix training phase, the weights of the base LLM remain frozen; only the small prefix vectors are updated [4]. This makes the approach highly data-efficient compared to full fine-tuning, as it leverages the existing capabilities of the large LLM [4].

The operational process involves several steps. First is **Prefix Training**, where separate prefixes are trained using a security-labeled dataset to represent characteristics like 'secure' (`SVEN_sec`) and 'vulnerable' (`SVEN_vul`) [4]. Specialized loss functions, including a contrastive loss (to differentiate secure vs. insecure generation) and a Kullback-Leibler (KL) divergence loss, are used to optimize only the prefix vectors while the main LLM's weights are kept frozen [4]. The KL divergence loss term encourages the model's output distribution (when using a prefix) to remain similar to the original LLM's distribution in non-security-sensitive code regions, thereby helping to preserve functional correctness [4]. Second is **Guided Generation**: during code generation (inference), the user selects a desired prefix (e.g., `SVEN_sec` for security hardening or `SVEN_vul` for adversarial testing). This prefix vector is simply prepended to the input representation before being fed into the *original, unchanged* LLM [4]. The prefix vector influences the LLM's internal computations, particularly the attention mechanism (how the model weighs the importance of different input tokens when generating output). This effectively steers the generation process towards producing code that aligns with the characteristic encoded in the prefix (e.g., more secure code or intentionally vulnerable code) without having altered the base LLM [4]. Extensive evaluations showed SVEN was highly effective [4].

As shown in Figure 1, applying `SVEN_sec` to harden a 2.7B parameter CodeGen LLM significantly boosted its secure code generation rate on a set of test scenarios from a baseline of 59.1% to 92.3% [4]. Conversely, using `SVEN_vul` for adversarial testing degraded the security rate to 36.8% [4]. Importantly, as illustrated in Figure 2, these strong security controls were achieved while closely matching the original LLM's functional correctness on benchmarks like HumanEval [4]. Further experiments showed SVEN's robustness to prompt perturbations, applicability across different LLMs (CodeGen, InCoder, SantaCoder), and even some generalization capability to CWEs not seen during prefix training [4]. SVEN thus offers a flexible, modular, and data-efficient

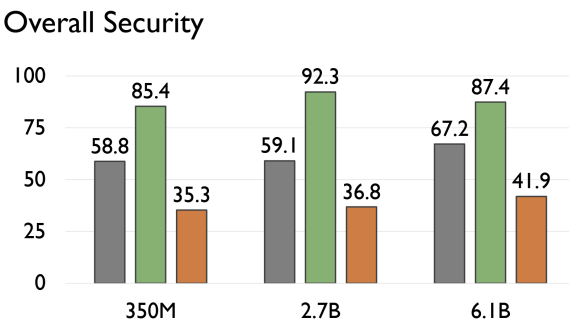

Figure 1. Security results of SVEN hardening (`SVEN_sec`, green) and adversarial testing (`SVEN_vul`, orange) compared to the original CodeGen LM (grey) across different model sizes (parameters). Security rate is the percentage of secure programs generated for main CWE test scenarios [4].

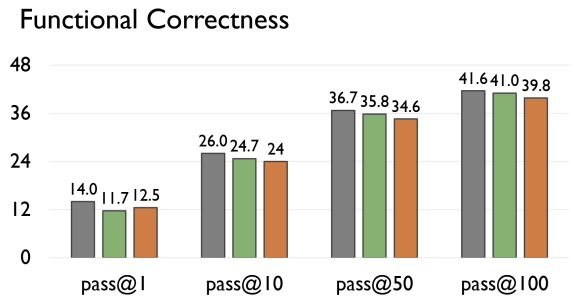

Figure 2. Functional correctness results (pass@k on HumanEval benchmark) for SVEN hardening (`SVEN_sec`, green) and adversarial testing (`SVEN_vul`, orange) compared to the original CodeGen LM (grey), likely across different model sizes [4]. Pass@k measures the percentage of problems for which at least one of k generated solutions passes unit tests.

method to control output security without costly retraining or fine-tuning [4].

6 Conclusion

AI code generation tools present a significant advancement for software development, offering notable productivity benefits but also introducing substantial security risks [1, 4]. Large Language Models, trained on vast public datasets, can inadvertently learn and propagate vulnerabilities like injection flaws and insecure handling of randomness or resources, making mitigation essential [2, 4]. Empirical studies confirm that AI-generated code frequently contains common and severe weaknesses found in real-world projects [2, 4]. Current mitigation strategies, including security-focused fine-tuning and adversarial techniques like SVEN, demonstrate potential for reducing specific vulnerabilities [4, 6]. Fine-tuning can teach models secure patterns from vulnerability fixes [6], while methods like SVEN allow for guiding existing models towards security without costly retraining [4]. However, these approaches often face limitations in generalization across different vulnerability types or languages, require high-quality curated data, and may have trade-offs with code correctness (primarily syntactic, though potentially functional as well) [4, 6]. Presently, no single technique offers a complete solution [4].

Moving forward, ensuring the safe deployment of AI in software development necessitates a multi-faceted approach. Key challenges include improving the generalization of security enhancements and overcoming the difficulty of obtaining comprehensive training and evaluation datasets [4, 6]. Future research should prioritize developing inherently secure model architectures, creating more realistic evaluation benchmarks, exploring combinations of mitigation techniques (like fine-tuning plus inference-time steering or robust post-generation analysis), enhancing automated repair tools (including LLM-based ones like Copilot Chat which have shown some success [2]), and addressing the data bottleneck through automated curation or careful crowdsourcing [1, 2, 4, 6]. Care must be taken in crowdsourcing efforts, however, to mitigate the risk of potentially malicious contributions. Effective human-AI collaboration remains paramount. Developers must stay vigilant, utilizing tools and knowledge to critically evaluate AI-generated code [1, 2]. Integrating automated security analysis into AI-assisted workflows and promoting developer education on secure coding in the context of AI are vital steps [2]. Ultimately, harnessing AI's benefits while managing its security risks demands a concerted effort from researchers, developers, and practitioners. A commitment to rigorous evaluation and balancing innovation with security will be crucial as these powerful tools become increasingly integrated into software creation [1, 2, 4].

Acknowledgments

I would like to thank Dr. Elena Machkasova for guiding me through the writing process. I would also like to thank Dr. Wenkai Guan for his advice and guidance of the senior seminar course. Finally, I would like to extend thanks to Ollie Willete for his valuable feedback.

References

[1] Owura Asare, Meiyappan Nagappan, and N. Asokan. 2024. A User-centered Security Evaluation of Copilot. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering* (Lisbon, Portugal) *(ICSE '24)*. Association for Computing Machinery, New York, NY, USA, Article 158, 11 pages. https://doi.org/10.1145/3597503.3639154

[2] Yujia Fu, Peng Liang, Amjed Tahir, Zengyang Li, Mojtaba Shahin, Jiaxin Yu, and Jinfu Chen. 2025. Security Weaknesses of Copilot-Generated Code in GitHub Projects: An Empirical Study. *ACM Trans. Softw. Eng. Methodol.* (Feb. 2025). https://doi.org/10.1145/3716848 Just Accepted.

[3] Anthony Gillioz, Jacky Casas, Elena Mugellini, and Omar Abou Khaled. 2020. Overview of the Transformer-based Models for NLP Tasks. In *2020 15th Conference on Computer Science and Information Systems (FedCSIS).*

179–183. https://doi.org/10.15439/2020F20

[4] Jingxuan He and Martin Vechev. 2023. Large Language Models for Code: Security Hardening and Adversarial Testing. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security* (Copenhagen, Denmark) *(CCS '23)*. Association for Computing Machinery, New York, NY, USA, 1865–1879. https://doi.org/10.1145/3576915.3623175

[5] Mohaiminul Islam, Guorong Chen, and Shangzhu Jin. 2019. An Overview of Neural Network. *American Journal of Neural Networks and Applications* 5, 1 (2019), 7–11. https://doi.org/10.11648/j.ajnna.20190501.12

[6] Junjie Li, Aseem Sangalay, Cheng Cheng, Yuan Tian, and Jinqiu Yang. 2024. Fine Tuning Large Language Model for Secure Code Generation. In *Proceedings of the 2024 IEEE/ACM First International Conference on AI Foundation Models and Software Engineering* (Lisbon, Portugal) *(FORGE '24)*. Association for Computing Machinery, New York, NY, USA, 86–90. https://doi.org/10.1145/3650105.3652299

[7] Jürgen Schmidhuber. 2015. Deep learning in neural networks: An overview. *Neural Networks* 61 (2015), 85–117. https://doi.org/10.1016/j.neunet.2014.09.003

[8] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In *Proceedings of the 31st International Conference on Neural Information Processing Systems* (Long Beach, California, USA) *(NIPS'17)*. Curran Associates Inc., Red Hook, NY, USA, 6000–6010.