Securing AI-Generated Code

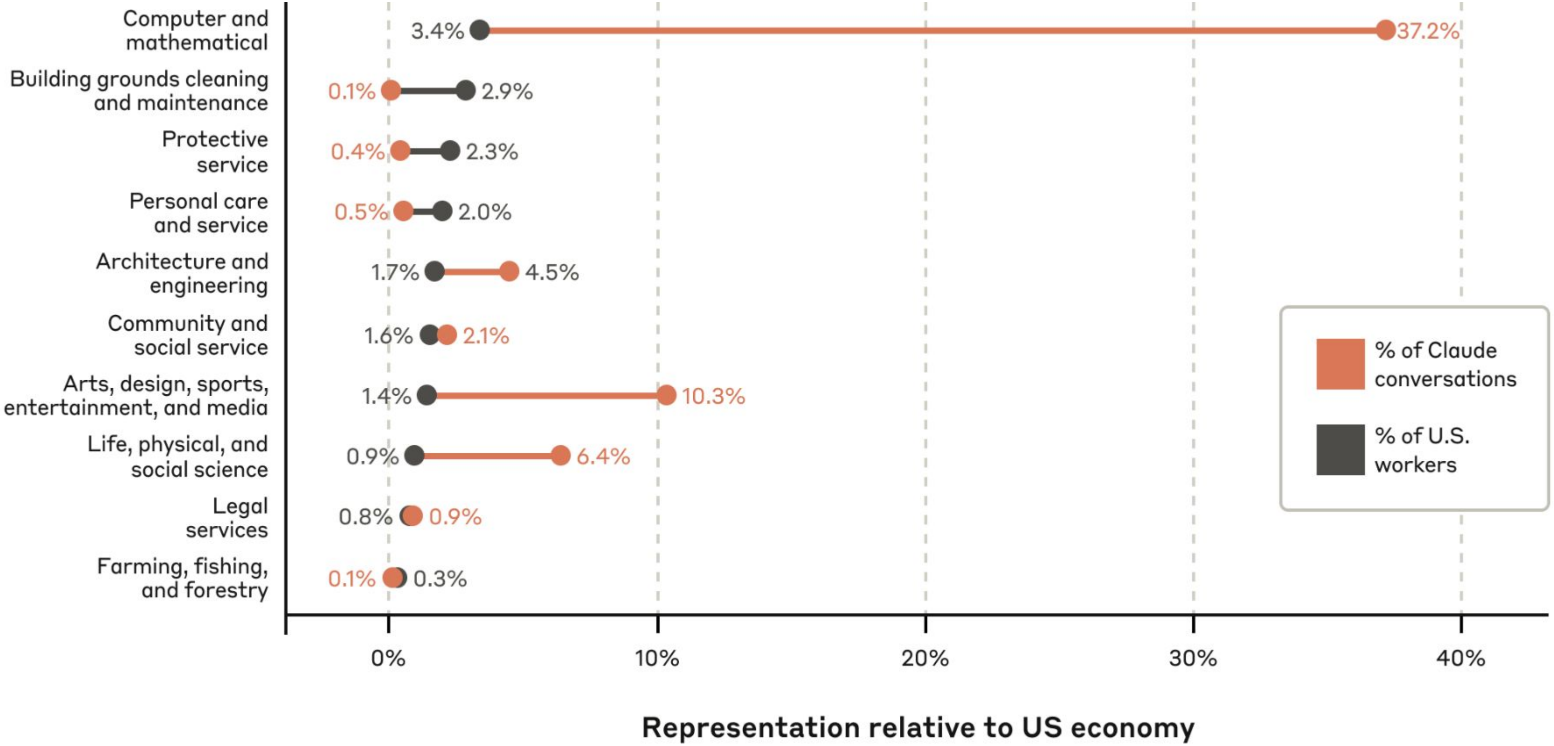
Andreas Nelson

“It is our job to create computing technology such that nobody has to program. And that the programming language is human. Everybody in the world is now a programmer. This is the miracle of artificial intelligence.”

Jensen Huang, CEO of Nvidia

Why should we care?

The majority of AI usage in the US is in software development



Handa et al, 2024

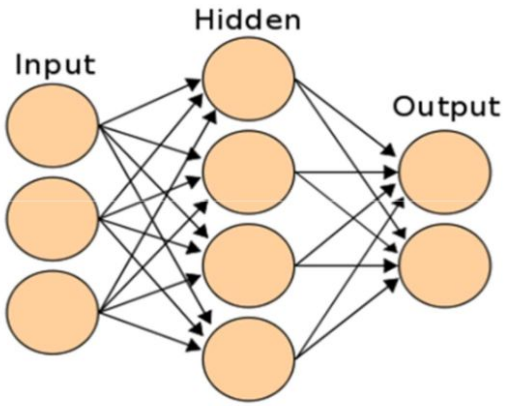
Outline

- Background
- Security Risks
- Methods of Mitigation
- Conclusion
- Sources

Background

Core Technologies - Neural Networks

- Interconnected "neurons" organized in layers
- Connection strength determined by weights
- Computations in hidden layers
- Backpropagation
  - Adjusting weights to minimize errors based on training



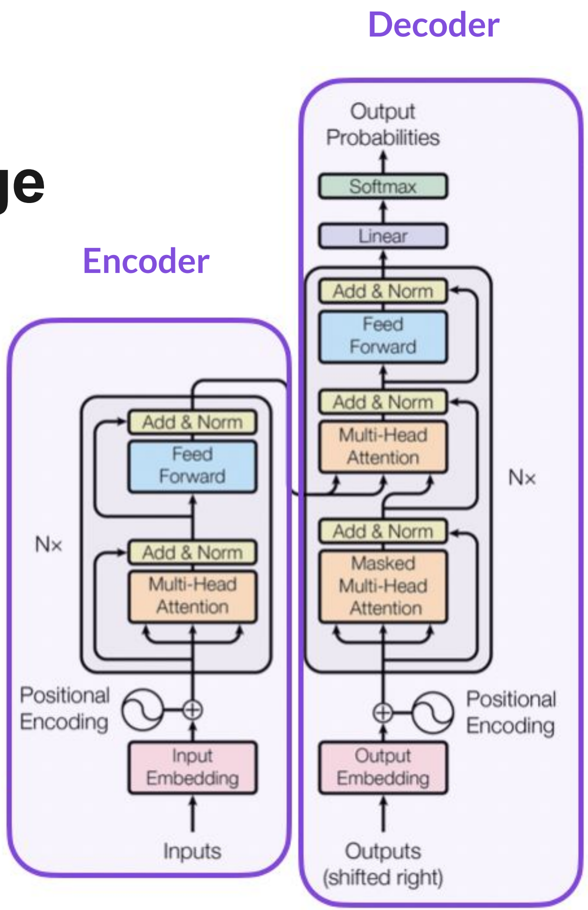
NN's typically have many hidden layers
Islam et al, 2019

Core Technologies - Large Language Models (LLMs)

LLMs are a type of **Deep Neural Network**

Pairs and layers

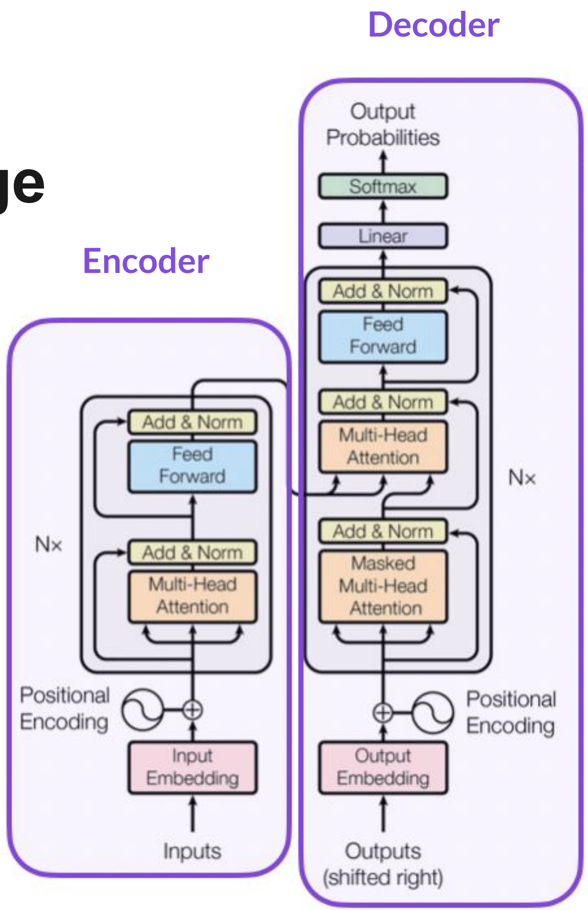
- Training: vast amounts of text and code
- Transformer Architecture
    - NN architecture designed for sequence processing



Vaswani et al, 2017

Core Technologies - Large Language Models (LLMs)

- Input Processing
  - Input text/code ➡ Numbers + Word Order
- Encoder: Context weighting and tokenization
- Decoder: Generates output using input context and previous output



Vaswani et al, 2017

Key Platforms- Github & Visual Studio Code

Github

- Free platform to collaborate and share code

VSC

- Microsoft's Integrated Development Environment (IDE)

Code Generation Tools

Interface between VSC and AI models

- Gemini
- Claude
- ChatGPT

Allows for the generation of code

GitHub Copilot

```python
def CalculateDaysBetweenDates(date1, date2):
    # date1 and date2 are in the format of 'YYYY-MM-DD'
    year1, month1, day1 = map(int, date1.split('-'))
    year2, month2, day2 = map(int, date2.split('-'))
    days = 0
    while year1 < year2 or month1 < month2 or day1 < day2:
        days += 1
        day1 += 1
        if day1 > 30:
            day1 = 1
            month1 += 1
        if month1 > 12:
            month1 = 1
            year1 += 1
    return days
```

Fu et al, 2025

Vulnerabilities - Common Weakness Enumerators

CWE’s stem from issues in,

- Design
- Implementation
- Operation

Designated with different levels of severity

Code Evaluation Tools

CodeQL

- Developed by Github
- Checks code security
    - Runs queries against a database representation code to identify patterns associated CWEs

HumanEval

- Checks Functional Correctness
- Benchmark dataset
    - Compares derived answer
    - Correct tries per attempts

Security Risks

The Fundamental Problem

- Inherited vulnerabilities in training data
  - Stack Overflow
  - Github
- Malicious intent and human error

How Common Are These Risks?

Language	# Snippets	# Security weaknesses	# Snippets containing security weaknesses	# containing more than one security weakness
Python	419	387	124 (29.5%)	65 (15.5%)
JavaScript	314	241	76 (24.2%)	37 (11.8%)
Total	**733**	**628**	**200 (27.3%)**	**102 (13.9%)**

27.3% chance of containing one or more security weakness

Fu et al, 2025

What Kinds of Vulnerabilities?

- 43 different CWE's identified across 733 Copilot code snippets
- most commonly found,
  - CWE-330 (Insufficient Randomization)
  - CWE-94 (Code Injection)
  - CWE-79 (Cross Site Scripting)

Distribution of CWEs in code snippets

CWE-ID	Name	Frequency	Percentage
CWE-330	Use of Insufficiently Random Values Weakness	114	18.15%
CWE-94	Improper Control of Generation of Code ('Code Injection')	62	9.87%
CWE-79	Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')	60	9.55%
CWE-78	Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection')	39	6.21%
CWE-427	Uncontrolled Search Path Element	35	5.57%
CWE-457	Use of Uninitialized Variable	30	4.78%
CWE-22	Improper Limitation of a Pathname to a Restricted Directory ('Path Traversal')	29	4.62%
CWE-772	Missing Release of Resource after Effective Lifetime	29	4.62%

Fu et al, 2025

Severity of Vulnerabilities

- 8 being in the most severe vulnerabilities of 2023
- 2 of the 3 most common being in the top 25 (2024)
  - CWE-94 (**#11**: Code Injection)
  - CWE-79 (**#1**: Cross Site Scripting)
- These 8 critical types accounted for 233 out of 628 (**37.1%**) of all weaknesses found in the study

cwe.mitre.org

1 Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')
CWE-79 | CVEs in KEV: 3 | Rank Last Year: 2 (up 1) ▲

2 Out-of-bounds Write
CWE-787 | CVEs in KEV: 18 | Rank Last Year: 1 (down 1) ▼

3 Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')
CWE-89 | CVEs in KEV: 4 | Rank Last Year: 3

4 Cross-Site Request Forgery (CSRF)
CWE-352 | CVEs in KEV: 0 | Rank Last Year: 9 (up 5) ▲

5 Improper Limitation of a Pathname to a Restricted Directory ('Path Traversal')
CWE-22 | CVEs in KEV: 4 | Rank Last Year: 8 (up 3) ▲

6 Out-of-bounds Read
CWE-125 | CVEs in KEV: 3 | Rank Last Year: 7 (up 1) ▲

7 Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection')
CWE-78 | CVEs in KEV: 5 | Rank Last Year: 5 (down 2) ▼

8 Use After Free
CWE-416 | CVEs in KEV: 5 | Rank Last Year: 4 (down 4) ▼

9 Missing Authorization
CWE-862 | CVEs in KEV: 0 | Rank Last Year: 11 (up 2) ▲

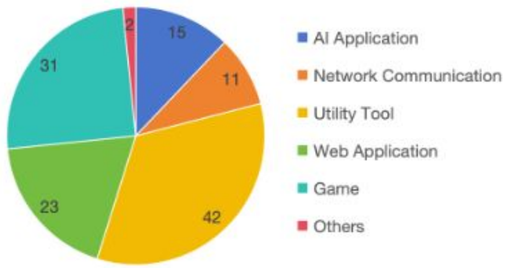
10 Unrestricted Upload of File with Dangerous Type
CWE-434 | CVEs in KEV: 0 | Rank Last Year: 10

11 Improper Control of Generation of Code ('Code Injection')
CWE-94 | CVEs in KEV: 7 | Rank Last Year: 23 (up 12) ▲

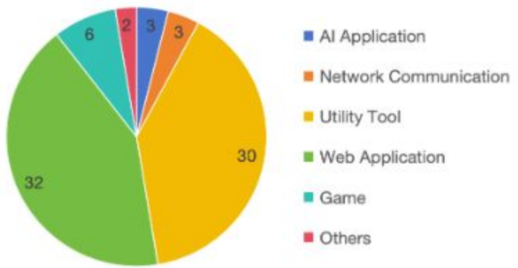
Factors Influencing Risk

Vulnerabilities differ by language

- Python
  - System interaction
    - CWE-78 (OS Command Injection)
  - Utility Tool domain projects
- JavaScript
  - Dynamic code generation
    - CWE-79 (XSS)
    - CWE-94 (Code Injection)
  - Web App projects



(a) application domains of Python code



(b) application domains of JavaScript code

Fu et al, 2025

Mitigation Strategy 1: Fine-Tuning Training Data

Concept

Improve the quality of training material → Improve the quality of the generated code

Methodology

- Creating security focused training data
  - Use real-world vulnerability fixes for CWEs
  - extracting pairs of insecure code and their corresponding secure versions
  - Li et al. collected 4900 commits
- CodeQL to evaluate output before and after

Results

Notable increase in generating non-vulnerable code.

- C: from 60.6% to 70.4%
- C++: 63.9% to 64.5%

Total (C)	278	427	135	194	462	185
Total (C++)	206	365	269	131	238	471
Non-vulnerable ratio (C)		60.6%			70.4%	
Non-vulnerable ratio (C++)		63.9%			64.5%	
Invalid ratio (C)		16.1%			22.0%	
Invalid ratio (C++)		32.0%			56.1%	

Li et al, 2024

Limitations

Training data creation labor-intensive

Computationally resource heavy

Success depends on quality and coverage of the dataset

CWE generalizations (Other languages/CWEs)

Increasing security, decreasing functional correctness

Mitigation Strategy 2: Security Hardening

Concept

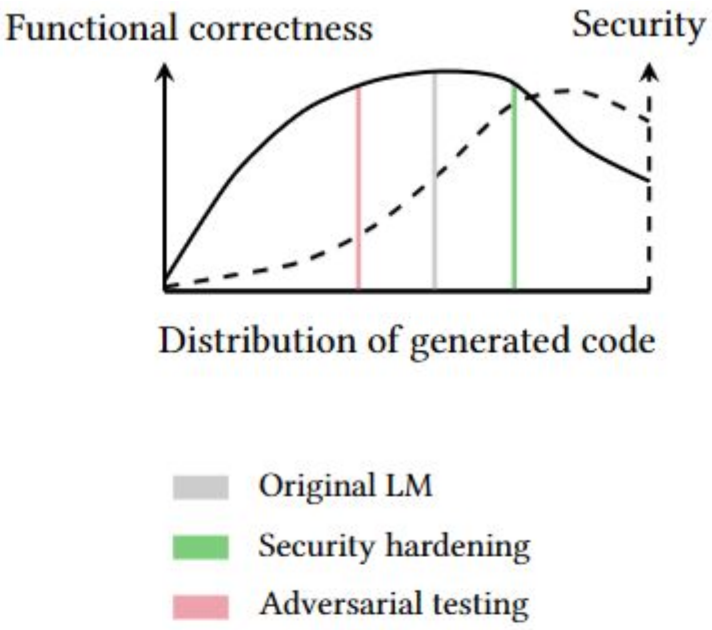
- Fine-tuning models is expensive
- Enhancing security without full model retraining
- influencing the generation process directly at inference time

Steering

Security Verifier Enhanced

Neural-Steering (SVEN)

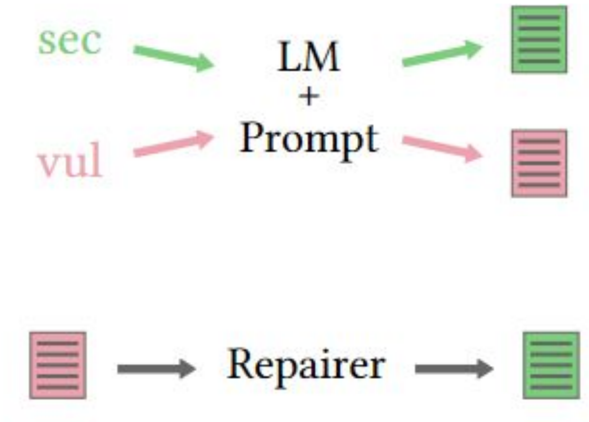
- Balancing functional correctness with security
  - Security Hardening
  - Adversarial Testing



He et al, 2023

Steering Process

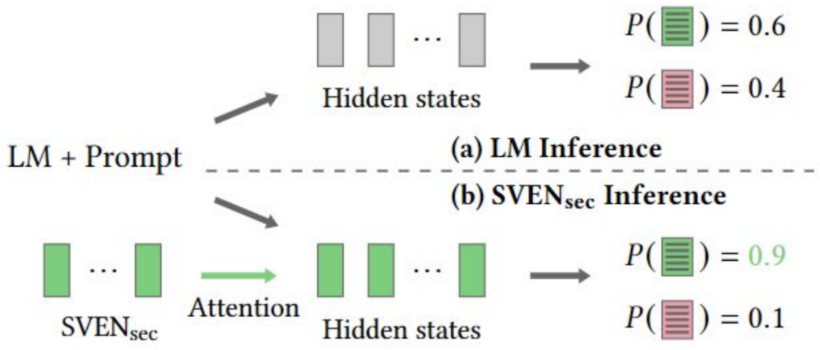
- Adversarial Testing
  - generate insecure code
- Security Hardening
  - Including adversarial examples in hardening process
  - Recognize and resist generating insecure code



He et al, 2023

Methodology

- How SVEN Works
  - Adversarial Training finds areas of insecurity
  - Areas mapped to prefix vectors
  - Vectors guide new prompts away from negative weights
  - Only affects areas of security
- Evaluation
  - CodeQL
  - HumanEval



He et al, 2023

SVEN Effectiveness

Security improvements

≈ 30%

Closely matched

original LLMs in

HumanEval within

1-2%

He et al, 2023

Conclusions

Comparison

Fine Tuning

- Pros: 10%
  - Improved security for targeted CWEs/languages
  - Modifies model's inherent behavior
- Cons:
  - Requires retraining/fine-tuning (compute/data intensive)
  - Improvement may not generalize well
  - Can hurt code validity

Hardening

- Pros: 30%
  - Modular
  - Demonstrated effective security control
  - Preserves functional correctness
- Cons:
  - May not generalize perfectly to all unseen CWEs
  - Effectiveness relies on quality of curated prefix-training data

Future Implications

Humans are safe… for now

- Review of generated code still necessary
- Current need for developer education
- Iterative CodeQL checks

Sources

- 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. Just Accepted (February 2025). https://doi-org.ezproxy.morris.umn.edu/10.1145/3716848
- 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 (ICSE '24). Association for Computing Machinery, New York, NY, USA, Article 158, 1–11. https://doi-org.ezproxy.morris.umn.edu/10.1145/3597503.3639154
- 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 (CCS '23). Association for Computing Machinery, New York, NY, USA, 1865–1879. https://doi-org.ezproxy.morris.umn.edu/10.1145/3576915.3623175
- 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 (FORGE '24). Association for Computing Machinery, New York, NY, USA, 86–90. https://doi-org.ezproxy.morris.umn.edu/10.1145/3650105.3652299

Questions?