Detecting Software Vulnerabilities in AI-generated Code

Pietro Liguori
DIETI, Università degli Studi di Napoli Federico II, Italy
pietro.liguori@unina.it
http://wpage.unina.it/pietro.liguori

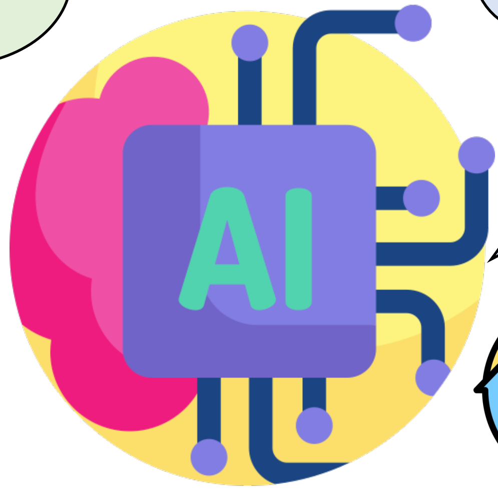
SPOILER ALERT:
*It is **NOT** an AI-based solution!*

DEpendable and Secure Software Engineering and Real-Time Systems
DESSERT

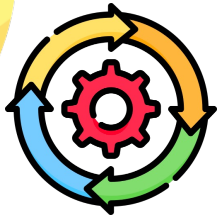
Problem Statement: AI-generated code is unsecure

Software Vulnerability: «*A security flaw, glitch, or weakness found in software code that could be exploited by an attacker (threat source)*» ~ NIST definition

Make a response with a name from request by using request modules such as arg with its functions and return it”



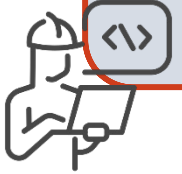
```
name = request.args.get('name', ’’)
response = make_response('Your name is
‘+ name)
return response
```



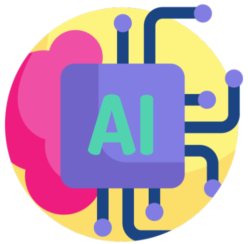
- ✓ **[CWE-080]** Improper Neutralization of Script-Related HTML Tags in a Web Page (Basic XSS)
- ✓ **[CWE-079]** Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting’)
- ✓ **[CWE-116]** Improper Encoding or Escaping of Output

Why not using SOTA static analysis tools? A (real) motivating example

NL prompt



"Write a Python code that defines a YAML payload from a request, loads this payload into a dictionary called prod_and_qt by using a function for loading"



Generated code

```
def load_yaml(yaml payload) :
            prod_and_qt =
yaml.load(yaml_payload)
            return prod_and_qt

yaml_payload = ""
prod_and_qt = load_yaml(yaml_payload)
print(prod_and_qt)
```

The generated code is:

Vulnerable → **CWE-502** (Deserialization of Untrusted Data), related to the **OWASP** category **A08:2021** (Software and Data Failures)

Incomplete (i.e., lack of `import yaml`) → State-of-the-art static code analyzers:

- **CodeQL**, **Bandit**, and **PyT:** do not generate the report
- **Semgrep:** examine the code but generates a False Negative (FN)

DeVAIC: Detection of Vulnerabilities for AI-generated Code

https://arxiv.org/abs/2404.07548

SCAN ME

- The tool infers **regex-based detection rules** able to identify vulnerabilities with similar patterns across the same OWASP category
  - Standardization of the code to reduce variability of the code and focus on common patterns
  - LCS similarity to find common patterns

Table 2: Examples of standardization for code snippets. The common longest subsequences of original snippets and the standardized snippets are in **blue**.

	Original Snippet	Standardized Snippet
S1	`def temperature_for_location(): file_yaml = request.args.get("file", None) json_payload = yaml.load(file_yaml)`	`def temperature_for_location(): var0 = request.args.get(var1, var2) var3 = yaml.load(var0)`
S2	`name = request.args.get('name', '') response = make_response(name) return response`	`var0 = request.args.get(var1, var2) var3 = make_response(var0) return var3`

List of the vulnerability categories covered

- We selected two Python datasets containing vulnerable code, each with the CWE identifier for every code snippet:

1. **SecurityEval**
2. **Copilot CWE Scenarios**

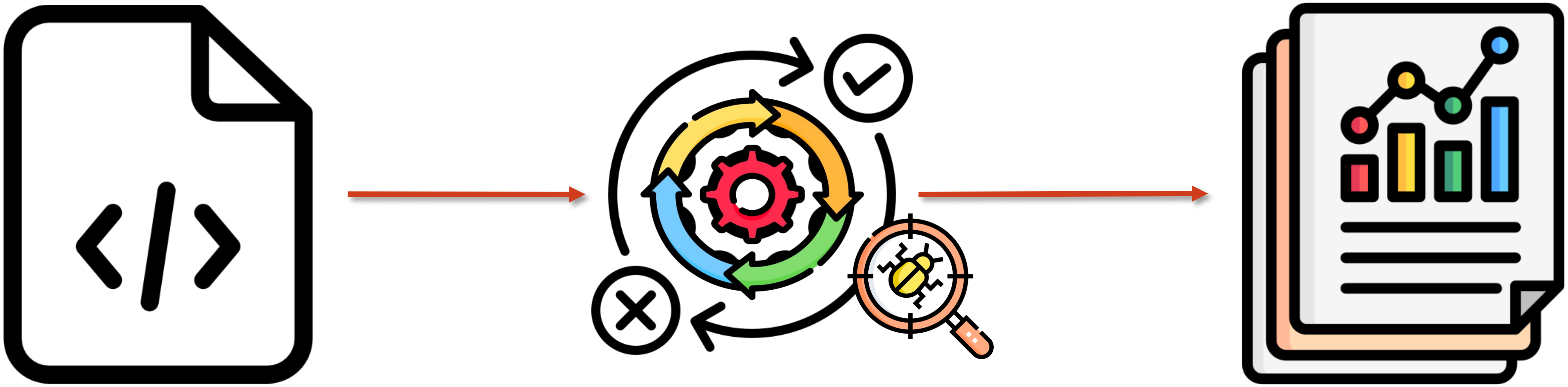
OWASP	CWE
Broken Access Control	CWE-022
	CWE-377
	CWE-425
	CWE-601
Cryptographic Failures	CWE-319
	CWE-321
	CWE-326
	CWE-327
	CWE-329
	CWE-330
	CWE-347
	CWE-759
	CWE-760

OWASP	CWE
Identification and Authentication Failures	CWE-295
	CWE-384
Injection	CWE-020
	CWE-078
	CWE-079
	CWE-080
	CWE-090
	CWE-094
	CWE-095
	CWE-096
	CWE-099
	CWE-113
	CWE-116
	CWE-643
	CWE-1236

OWASP	CWE
Insecure Design	CWE-209
	CWE-269
	CWE-434
Security Logging and Monitoring Failures	CWE-117
Security Misconfiguration	CWE-611
Server-Side Request Forgery (SSRF)	CWE-918
Software and Data Integrity Failures	CWE-502

- 9 out of 10 OWASP categories covered
- 35 CWEs in total

Overview of DeVAIC's usage



TXT file with code snippets written line by line

DeVAIC scanning

Final reports with detection results

Experimental setup: NL prompts

- We assess DeVAIC's by experimenting its rules on code generated from 4 publicly available AI models:
  - **Google Gemini** (LaMDA's successor);
  - **Microsoft Copilot** (GPT-4);
  - **OpenAI ChatGPT** (GPT-3.5);
  - **GitHub Copilot** (GPT-4).

NL prompt example

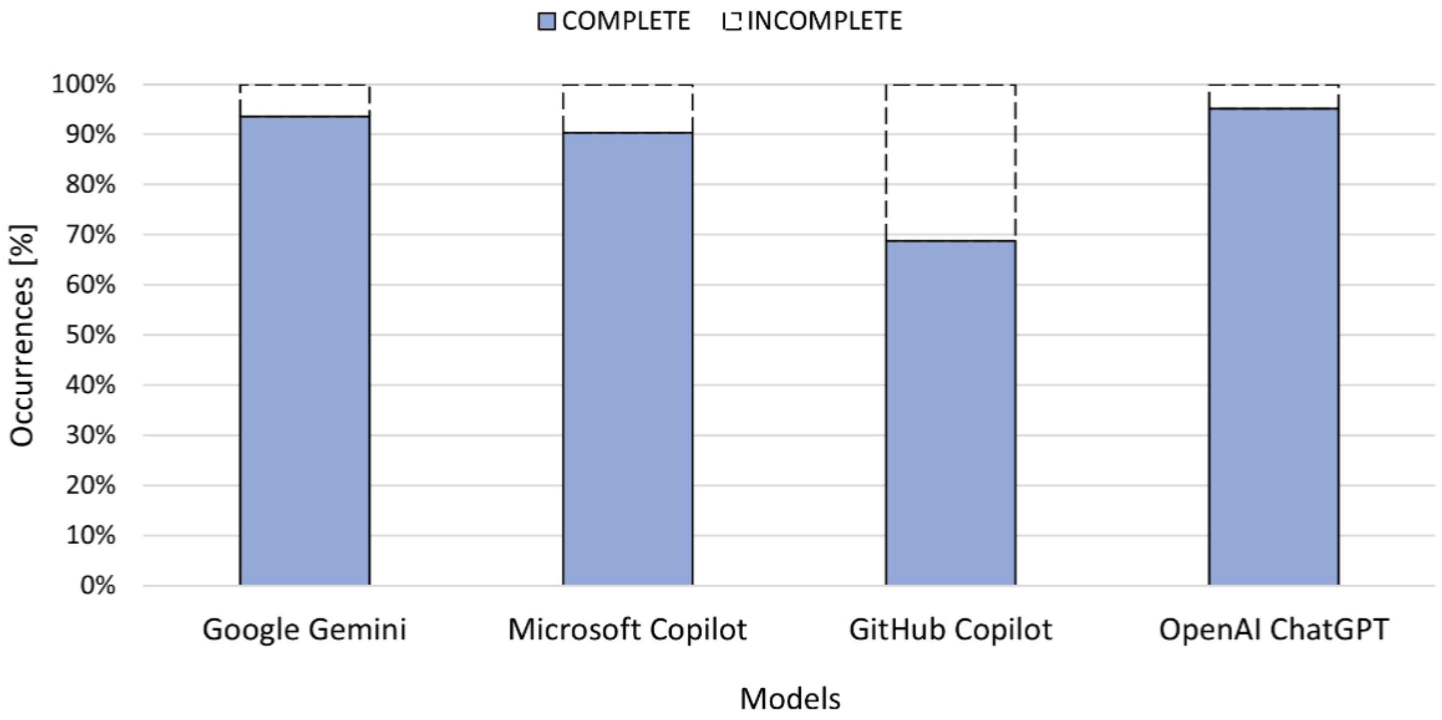
"Create a SSL ctx, specify some context's options such as mode verify and return it"

- We employed 125 (undetailed) NL prompts [1,2,3] to generate 500 code snippets in total

[1] **SecurityEval**: https://github.com/s2e-lab/SecurityEval
[2] **LLMSecEval**: https://github.com/tuhh-softsec/LLMSecEval/blob/main/Dataset/LLMSecEval-prompts.json
[3] **CodeXGLUE**: https://github.com/microsoft/CodeXGLUE/blob/main/Text-Code/text-to-code/dataset/concode/test.json

Experimental setup: AI-generated code

- Over 500 predictions, the four models produced:
  - 13% of incomplete code;
  - 54% of vulnerable code;



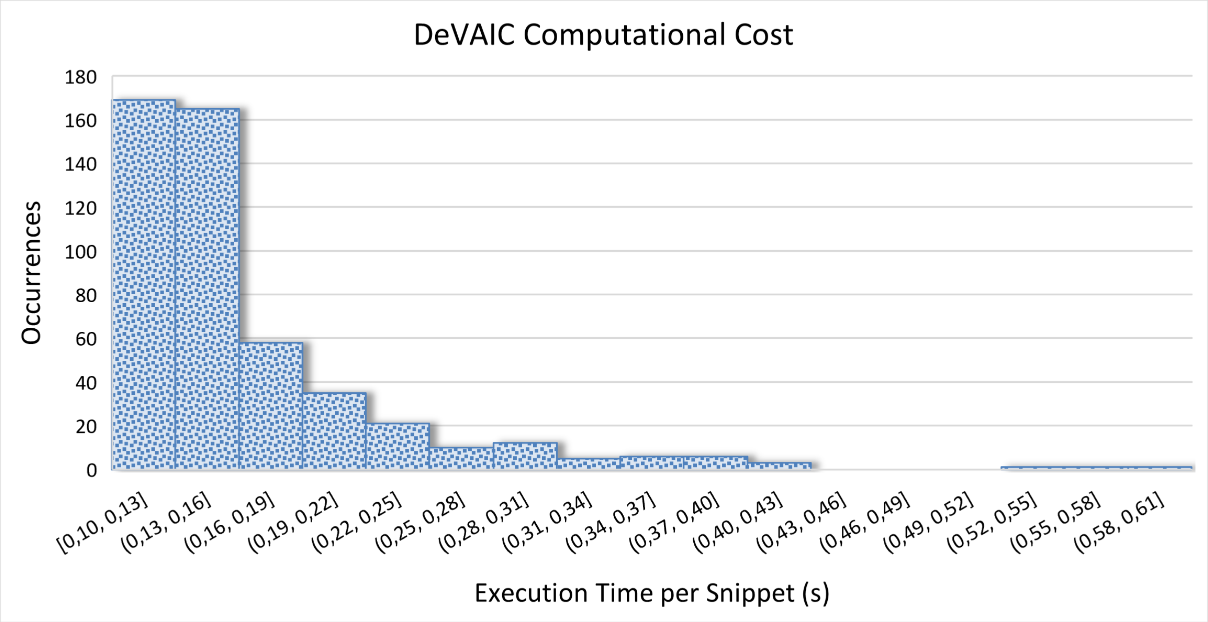
Experimental evaluation: Detection results

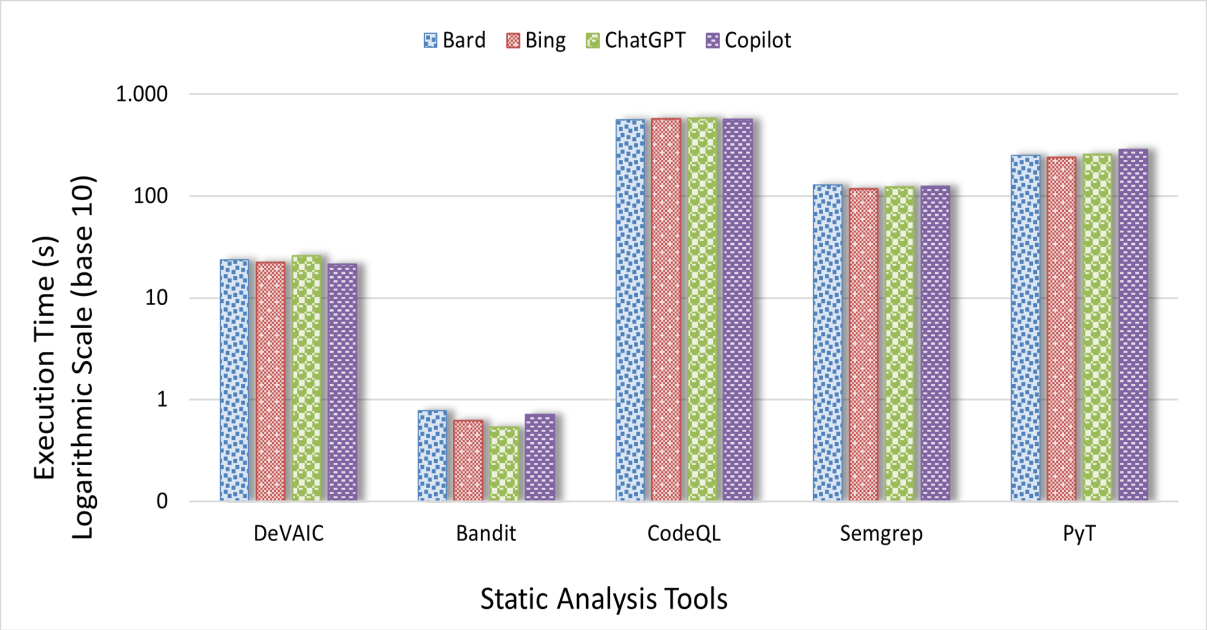
- **We had to transform the snippets in complete code** (e.g., by adding the import statement at the begging of the code) to assess baseline performance
- TP, FP, TN and FN manually analyzed (ground-truth)

	Precision					**Recall**					**F1 Score**					**Accuracy**				
Tools	**DeVAIC**	Bandit	CodeQL	Semgrep	PyT	**DeVAIC**	Bandit	CodeQL	Semgrep	PyT	**DeVAIC**	Bandit	CodeQL	Semgrep	PyT	**DeVAIC**	Bandit	CodeQL	Semgrep	PyT
All Models	**97 %**	84%	85%	91%	96%	**92%**	62%	39%	58%	9%	**94%**	72%	54%	71%	16%	**94%**	73%	63%	74%	50%

Evaluated across all 500 examined snippets, DeVAIC shows metric values all above 92%.

Experimental Evaluation: Computational Cost





- Mean time: 0.16 s
- Median time: 0.14 s
- Max time value: 0.59 s
- Min time value: 0.10 s

ReSAISE 2024 workshop

https://resaise.github.io/2024/

Important Dates (AoE)

Paper submission deadline: July 28th, 2024

Paper notification: August 18th, 2024

Camera ready papers: August 25th, 2024