Leveraging Large Language Models for Command Injection Vulnerability Analysis in Python: An Empirical Study on Popular Open-Source Projects

Yuxuan Wang[1], Jingshu Chen[1], Qingyang Wang[2]

[1]Department of Computer Science, University of Alabama in Huntsville, Huntsville, AL, USA.
[2]Department of Computer Science and Engineering, Louisiana State University, Baton Rouge, LA, USA.

[1]{yw0029, jc1540}@uah.edu; [2]qwang26@lsu.edu;

Abstract

Command injection vulnerabilities are a significant security threat in dynamic languages like Python, particularly in widely used open-source projects where security issues can have extensive impact. With the proven effectiveness of Large Language Models(LLMs) in code-related tasks, such as testing, researchers have explored their potential for vulnerabilities analysis. This study evaluates the potential of large language models (LLMs), such as `GPT-4`, as an alternative approach for automated testing for vulnerability detection. In particular, LLMs have demonstrated advanced contextual understanding and adaptability, making them promising candidates for identifying nuanced security vulnerabilities within code. To evaluate this potential, we applied LLM-based analysis to six high-profile GitHub projects—Django, Flask, TensorFlow, Scikit-learn, PyTorch, and Langchain—each with over 50,000 stars and extensive adoption across software development and academic research. Our analysis assesses both the strengths and limitations of LLMs in detecting command injection vulnerabilities, evaluating factors such as detection accuracy, efficiency, and practical integration into development workflows. In addition, we provide a comparative analysis of different LLM tools to identify those most suitable for security applications. Our findings offer guidance for developers and security researchers on leveraging LLMs as innovative and automated approaches to enhance software security.

Keywords: Large Language Models, Test generations, Software Testing

1 Introduction

Python's rich ecosystem makes it an ideal choice for a wide range of AI tasks, from scripting to developing complex models. Widely adopted libraries, such as TensorFlow [1] and PyTorch [2], offer robust tools for machine learning and deep learning applications. However, as the use of AI techniques and related open-source software continues to grow, addressing security vulnerabilities becomes increasingly essential—particularly in popular libraries and frameworks where security flaws can lead to widespread, systemic risks.

One such security vulnerability, command injection, represents a critical threat in dynamic languages like Python. Command injection vulnerabilities allow attackers to exploit applications by executing unauthorized commands, potentially compromising system integrity and exposing sensitive data. An exploited vulnerability can lead to data breaches, unauthorized access to sensitive resources, and loss of system control. For instance, an attacker could access confidential data or manipulate system files, which can disrupt service and harm user trust [3]. The urgency of addressing these vulnerabilities is underscored by their prominent ranking in security advisories, such as the Common Weakness Enumeration (CWE) [4], and by recent alerts [5] from organizations like CISA and the FBI, which highlight the risks these vulnerabilities pose to common software products.

The Common Vulnerabilities and Exposures (CVE) database has documented multiple instances of command injection vulnerabilities within Python libraries [6]. One such example, shown in Listing 1, involves a vulnerability reported in CVE-2022-29216 [7]. This vulnerability existed in the `preprocess_input_exprs_arg_string` function within the `saved_model_cli.py` file, where an `eval()` method call enabled command injection through unvalidated inputs. By passing malicious commands via the `input_exprs_str` parameter, an attacker could exploit this function due to the lack of an input validation mechanism.

Existing tools, such as Bandit [8], have been instrumental in identifying certain types of command injection vulnerability. However, those static analysis tools often require fully compiled code and may not adapt well to the nuances of large, fragmented codebases or evolving vulnerability patterns. Recent advances in large language models (LLMs), including models like GPT-4, offer promising alternatives in code analysis and generation [9–13]. Unlike traditional tools, LLM-based approaches could analyze both compiled and non-compiled code fragments and generate contextually relevant vulnerability assessments without requiring the complete code structure. In addition, an LLM-based approach can generate security tests that validate vulnerability assessments, providing an additional layer of security assurance.

For instance, as demonstrated in Section 2.1, detecting vulnerabilities in the code snippet in Listing 2 presents motivation for our work in this paper. First, like many similar code examples, the snippet is fragmented and non-compilable, which limits the applicability of traditional vulnerability detection tools that rely on fully compiled code for analysis. Unlike conventional tools, large language models (LLMs) can analyze vulnerabilities directly within code snippets, regardless of their compilability. Furthermore, as demonstrated by the different analysis results of the two methods in

section 2.1.1 and Section 2.1.2, the approach of adopting `GPT-4` are capable of identifying vulnerabilities that may be overlooked by existing detection tools like `Bandit`. Additionally, LLMs can generate security tests for functions to validate their assessments, adding a layer of verification. These advantages motivate our proposed work of applying LLM-based approach to detect command injection vulnerabilities.

```
def preprocess_input_exprs_arg_string(input_exprs_str, safe=True):
 input_dict = {}

 for input_raw in filter(bool, input_exprs_str.split(';')):
   if '=' not in input_exprs_str:
     raise RuntimeError('--input_exprs "%s" format is incorrect. Please follow'
                        '"<input_key>=<python expression>"' % input_exprs_str)
   input_key, expr = input_raw.split('=', 1)
   if safe:
     try:
       input_dict[input_key] = ast.literal_eval(expr)
     except Exception as exc:
       raise RuntimeError(
           f'Expression "{expr}" is not a valid python literal.') from exc
   else:
     input_dict[input_key] = eval(expr)
 return input_dict
```

Listing 1: Command injection vulnerability instance in Tensorflow.

Contributions: The contributions of this paper are as follows:

- We conducted a comprehensive analysis of command injection vulnerabilities across 13,037 Python files from 6 most popular open-source projects, each with over 50K stars on GitHub. Our work explores the effectiveness and completeness of large language model (LLM)-based analysis for detecting command injection vulnerabilities, providing a rigorous evaluation of four different LLMs, including GPT-4 [14], GPT-4o [15], Claude 3.5 Sonnet [16] and DeepSeek-R1[17].
- We examined the characteristics of the command injection vulnerabilities that might be missed by LLM-based approach. Our results show the limitations of LLMs in this area and inspire future research in the domain
- We compared our LLM-based approach to traditional security tools, specifically Bandit, to evaluate improvements in accuracy and completeness when identifying command injection vulnerabilities.
- Our study built a dataset of 13,037 Python files from six high-profile GitHub projects—Django, Flask, TensorFlow, Scikit-learn, PyTorch, and Langchain—each with over 50,000 stars and extensive adoption across software development and academic research. This dataset is available in the GitHub, forming a benchmark resource for further research in vulnerability detection in the domain.

2 Motivation and Background

2.1 Motivating Example

In Listing 2, we present a Python function from the PyTorch [2] project as a motivating example. This function's purpose is to retrieve and return a list of process IDs for all child processes associated with a given process ID (pid). In the second line, it employs the `subprocess.Popen()` method to execute the command `pgrep -P {pid}`. Among

the parameters passed to `subprocess.Popen()`, `pid` is a formal parameter whose value can be controlled externally, and `shell=True` enables the execution of command strings directly in the shell. This combination presents a significant vulnerability to command injection attacks.To investigate its vulnerability, we applied two different analysis methods to this function: the existing detection tool, Bandit, and an LLM-based approach using GPT (e.g., `GPT-4`), and then compared their detection results.

```
def get_child_pids(pid):
    pgrep = subprocess.Popen(args=f"pgrep -P {pid}", shell=True, stdout=
     subprocess.PIPE)
    pgrep.wait()
    out = pgrep.stdout.read().decode("utf-8").rstrip().split("\n")
    pids = []
    for pid in out:
        if pid:
            pids.append(int(pid))
    return pids
```

Listing 2: A candidate function from Pytorch project[2].

2.1.1 Applying `Bandit` to Analyze the Function in Listing 2

First, we used Bandit [8], an existing tool for detecting security issues in Python code, to determine if there was a command injection vulnerability in this function. Figure 1a illustrates the workflow for detecting code security issues with Bandit. We executed Bandit following the instructions shown in the blue block, while the yellow block displays Bandit's detection report. According to the report, Bandit concluded that this function does not contain a command injection vulnerability.

2.1.2 Applying `GPT-4` to Analyze the Function in Listing 2

Figure 1b illustrates our approach in using GPT-4 to analyze the function in Listing 2. The blue square contains the task assigned to GPT-4, while the yellow square displays GPT-4's output. The analysis begins with GPT-4 evaluating the function to identify any command injection vulnerabilities. As shown in Figure 1b, GPT-4's output indicates that it detected a command injection vulnerability in this function. Furthermore, GPT-4 generates a security test to verify the function's safety, adding confidence to its findings.

Figure 1b shows the security test generated by GPT for the get-_child_pids function. It is written using Python's unittest framework. In this code, a test file named test_file.txt is first created for later testing. Next, a malicious command is passed to the get-_child_pids function, and since there is no protection, "pgrep -P {pid}" is interpreted as two commands: first execute pgrep -P, and then execute "rm {self.test_file}", which deletes the test file. After running this security test, the result obtained is True, which indicates that the test file has been deleted, means that the get_child_pids function is vulnerable to command injection attack. This result proves the analysis of the GPT-4 model.

Motivation. Detecting vulnerabilities in the code snippet shown in Listing 2 motivates our work. The snippet, like many similar examples, is fragmented and non-compilable, which limits the effectiveness of traditional detection tools that require

(a) Result of Bandit (b) Result of GPT-4

Fig. 1: The detection results for GPT-4 and Bandit.

fully compiled code. In contrast, large language models (LLMs) can analyze vulnerabilities directly within code snippets. As demonstrated in Sections 2.1.1 and 2.1.2, our approach using `GPT-4` identifies vulnerabilities that tools like `Bandit` may miss. Moreover, LLMs can generate security tests for functions to validate their assessments, adding an extra layer of verification. These advantages motivate our use of LLMs to detect command injection vulnerabilities.

2.2 Large Language Models (LLMs)

Large Language Models (LLMs) [18] are large-scale deep learning models designed to perform a wide range of natural language processing (NLP) tasks, including text recognition, translation, prediction, and generation. LLMs are primarily built upon the Transformer architecture [19], employing self-supervised and semi-supervised learning methods to pre-train on extensive datasets, which allows them to learn language patterns and complex semantic structures.

Due to their foundation on the Transformer, LLMs inherit its encoder-decoder structure and can generally be categorized into three main groups: encoder-only, encoder-decoder, and decoder-only models [20]. Encoder-only models focus on understanding and encoding the input data. Examples include BERT [21] and its variations, such as CodeBERT [22] and ALBERT [23]. Encoder-decoder models utilize both encoding and decoding layers for tasks that require both input processing and output generation, with prominent examples being T5 [24] and CoTexT [25]. Decoder-only models are primarily used for generation tasks; examples include the GPT model

family, such as GPT-3 [26], GPT-3.5 [27], GPT-4 [14], GPT-4o [15] as well as other models like Google's PaLM [28], Meta's LLaMA [29] and DeepSeek-R1[17]. For Claude 3.5 Sonnet [16], a recently popular large language model, it cannot be determined at this time which architecture category it belongs to, as Anthropic has not released information about its internal architecture.

LLMs have demonstrated outstanding performance across various fields, including software engineering [30–34] and healthcare [35, 36], where they have been widely adopted and rigorously evaluated. In this paper, we leverage GPT-4, GPT-4o, Claude 3.5 Sonnet and DeepSeek-R1, four state-of-the-art LLMs, for our experiments, given their advanced capabilities in understanding and generating complex code patterns, which are essential for effective vulnerability analysis.

3 Study Design

3.1 Research Questions

The aim of this study is to answer the following research questions:

- **RQ1:** How effective are large language models (LLMs) like GPT in identifying command injection vulnerabilities in dynamic languages, specifically Python?
- **RQ2:** What types of Python command injection vulnerabilities might our LLM-based approach fail to detect?
- **RQ3:** What is the running cost(in terms of time and finance) of GPT-4?
- **RQ4:** How do different large language models compare in terms of accuracy, consistency, and efficiency in detecting command injection vulnerabilities in Python code?
- **RQ5:** How does the accuracy and efficiency of LLM-based vulnerability detection compare with traditional vulnerability analysis tools, such as Bandit?

3.2 Dataset

We selected six popular open-source Python projects, each with over 50,000 stars on GitHub, for our study. Table 1 lists the project names, versions, the total number of files, and the number of Python files in each project. These projects are widely used in academic research and software development, covering areas such as web framework development—Django [37] and Flask [38]; machine learning modeling—TensorFlow [1], Scikit-learn [39], and PyTorch [2]; and LLM application development—Langchain [40]. We filtered all Python files in these projects because our research focuses on detecting command injection vulnerabilities in Python code

4 Our Approach

4.1 Approach Overview

Figure 2 illustrates the overview of our approach workflow, which consists of five steps: (1) filtering Python files from the projects, (2) identifying candidate functions, (3) using LLMs to detect vulnerabilities, (4) using LLMs to generate security tests, and

Table 1: Studied projects(As Of January 2024)

Project	Version	No. of Files	Stars(K)	No. of Python Files	LOC of Python Files(k)	No. of Candidate Functions
Django	4.2.7	6,740	80.5	2,772	399.8	17
Flask	3.0.0	249	68	82	13.4	2
Langchain	v0.0.330	3,933	94.5	2,290	227.1	13
TensorFlow	2.14.0	31,082	186	3,106	1,014.7	46
Scikit-learn	1.3.2	1,569	60	923	317.5	7
PyTorch	2.1	12,401	83.7	3,864	1,234	105

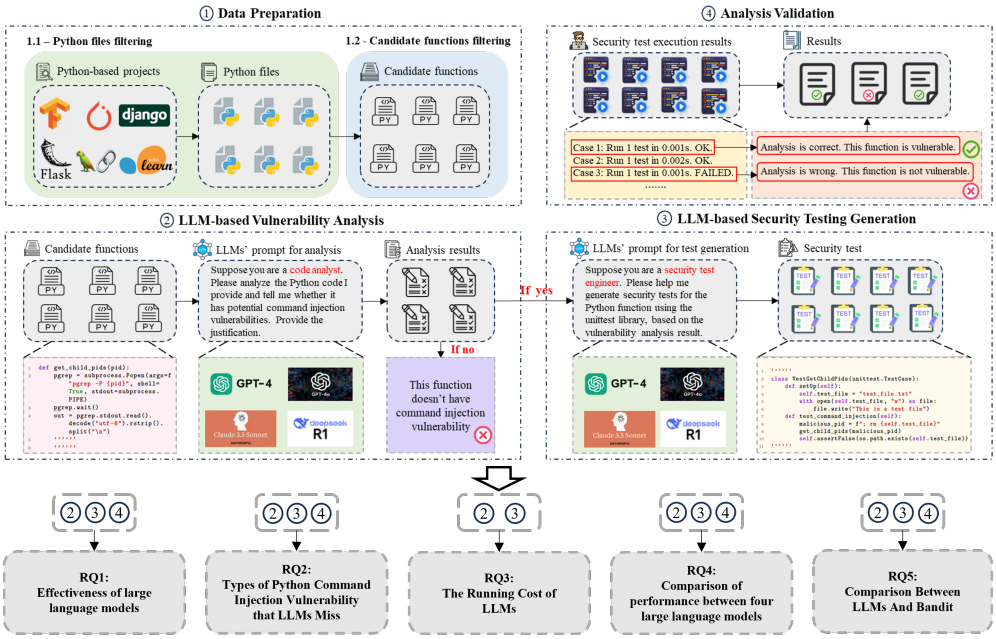


Fig. 2: The overview of the proposed LLM-based approach.

(5) validating the analysis results of LLMs. In the remaining section, we present each step in details.

4.2 Data preparation

In the six target Python projects, in addition to the `.py` source files, there are image resources, README documents, and various other non-Python files. Since our objective is to detect command injection vulnerabilities in Python code, our first step was to extract all Python files from these projects for analysis. We developed a Python script for this purpose, and Table 1 lists the number of Python files we collected. Table 2 lists 26 functions from the Semgrep [41] database that are known to introduce command injection vulnerabilities in Python. We categorized these functions into three

groups: built-in functions, functions from the `subprocess` module, and functions from the `os` module.

Table 2: List of Python methods prone to command injection vulnerabilities.

No.	Types of Python libraries and functions	Methods
1	built-in function	exec() eval()
2	subprocess module	subprocess.call(user_input) subprocess.run(user_input) subprocess.Popen(user_input) subprocess.check_output(user_input)
3	os module	os.popen() os.system() os.spawnl() os.spawnle() os.spawnlp() os.spawnlpe() os.spawnv() os.spawnve() os.spawnvp() os.spawnvpe() os.posix_spawn() os.posix_spawnp() os.execl() os.execle() os.execlp() os.execlpe() os.execv() os.execve() os.execvp() os.execvpe()

To find all the candidate functions, we used a Python script to extract the Python files which contains the above methods from all the Python files in the 6 Python projects. After that, found the functions from the extracted files that contain the above methods and stored each of the found functions in separate files. Finally, we found 190 candidate functions in total and the detailed results are shown in Table 1.

4.3 LLM-based Vulnerability Analysis & Security Testing Generation

In this section, we present two important steps in our approach: using LLMs to analyze functions for potential command injection vulnerabilities and generating security test code for functions identified as potentially vulnerable.

After filtering candidate functions using the method described above, we obtained a set of functions that may contain injection vulnerabilities. However, this set is presenting uncertainty, as the presence of these methods in some functions may not necessarily lead to vulnerabilities, and there may be constraints within the functions that mitigate such risks.

To further refine this analysis, we leverage GPT-4 to generate test case to evaluate whether each candidate function is truly vulnerable to command injection. If GPT-4 determines that a function is vulnerable ("Yes"), it proceeds to generate a security test case. If GPT-4 determines that the function is safe ("No"), no command injection vulnerability is detected.

Figure 3 shows the prompts used for these two steps. Specifically, [INPUT] is the Python function need to test, [JUSTIFICATION] represents the reason provided by LLM for whether a function contains a command injection vulnerability. [CODE] presents the security test generated by LLM. [REQUIREMENTS] represents rules that LLM tool needs to follow in generating security test. Specifically, these rules are:

- Include the source function being tested without modifying its name or content.
- Perform a command injection test, if there are methods in the function that would lead to a command injection attack. Generate an os command as its input to do the test. For example, create a test file and then attempt a command injection to delete it.
- Set the assertion section to verify if the command is executed successfully.
- Only generate the code; do not provide textual descriptions or suggestions.
- Use the unittest library, but avoid using mock modules or other simulation objects.
- Import any necessary libraries to run the code.
- Avoid redefining subprocess.call, subprocess.run, exec, or other methods in the test code.

To ensure the reliability of the GPT output, we adopted the "mimic-in-the-background" prompting method proposed in [42] when designing prompts for the GPT-4 model. As illustrated in Figure 3, the system prompts instruct the LLM to simulate answering the query in the background 10 times and then to select the response it considers most accurate. Additionally, to minimize output randomness, we set the temperature parameter for both the GPT-4 and GPT-4o models to zero.

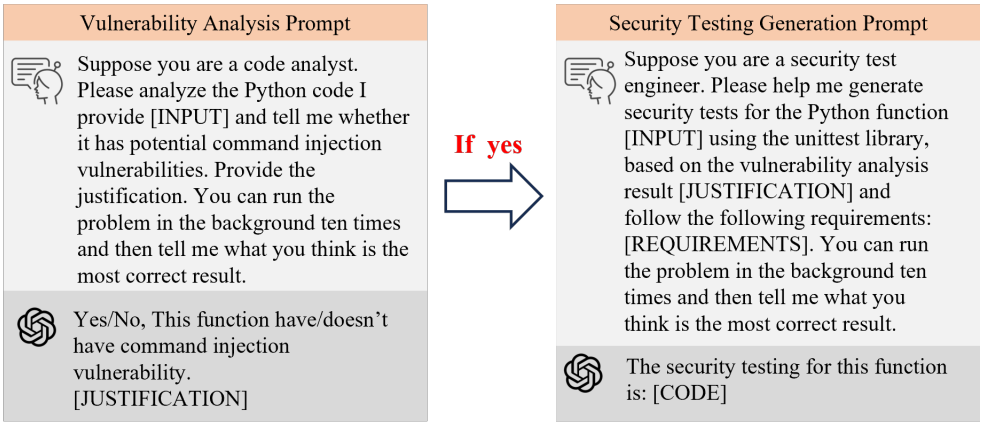


Fig. 3: LLM prompt for vulnerability analysis and security testing generation.

4.4 Validation

Some of the test scripts generated by the GPT model require additional modifications before they can be executed. This is typically due to missing libraries, specific parameter settings needed for execution, or operating system command paths that must be adjusted to the user's environment. To address these issues, we manually modified the test scripts to ensure compatibility with the systems used in our experiments. After making these adjustments, we were able to run the modified test scripts, allowing us to confirm which candidate functions actually contained command injection vulnerabilities. Figure 4 shows a complete example of the flow of detecting command injection vulnerabilities using our approach.

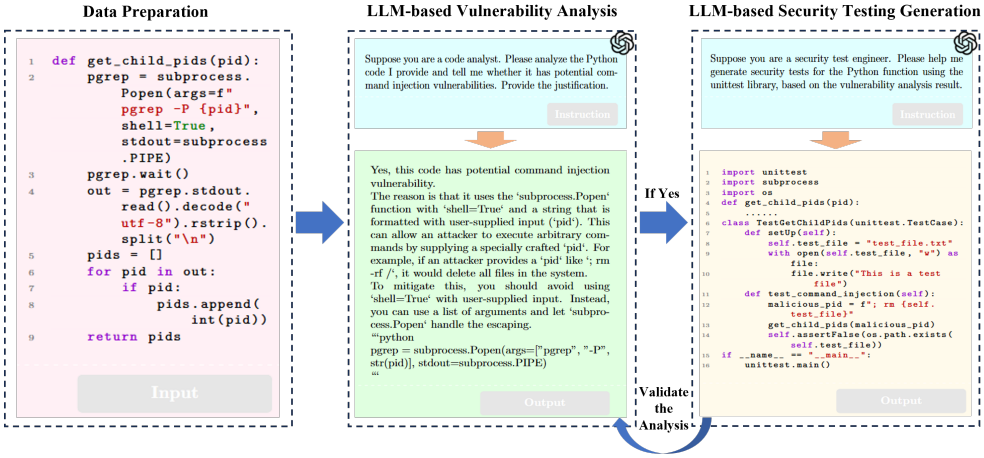


Fig. 4: Example workflow of the proposed LLM-based approach.

5 Evaluation

We conducted a detailed empirical evaluation of our proposed approach, focusing on answering three key research questions described in Section 3.1.

5.1 RQ1: Effectiveness Evaluation of the Proposed Approach

We evaluated the effectiveness of our proposed approach by applying it to 190 candidate functions drawn from six Python projects, as outlined in Section 3.2.

Table 3 presents the experimental results. Based on GPT-4's evaluation, 100 functions were identified as containing command injection vulnerabilities, while 90 were determined to be free from vulnerabilities.

After running the automated testing component, we observed the following outcomes:

- **True Positives (TP):** 67 cases (35%) were identified correctly, where GPT-4 determined a command injection vulnerability existed, and security testing confirmed it.
- **False Positives (FP):** 31 cases (16%) were identified as false positives, where GPT-4 flagged a vulnerability, but security testing showed it did not exist.
- **True Negatives (TN):** 75 cases (40%) were correctly labeled as non-vulnerable by GPT-4, confirmed by the absence of vulnerabilities in security tests.
- **False Negatives (FN):** 15 cases (8%) were false negatives, where GPT-4 missed identifying a vulnerability, but manual review revealed its presence.

Additionally, there were two invalid cases (1%) in which security tests could not run due to environmental or dependency issues.

This evaluation demonstrates the strengths and limitations of using GPT-4 for command injection vulnerability detection in Python functions, highlighting areas where LLM-based approaches may benefit from further refinement.

Table 3: Detection results of command injection vulnerabilities using GPT-4.

Project	No. of cases	True positive	False positive	True negative	False negative	Invalid
Django	17	5	4	7	1	0
Flask	2	2	0	0	0	0
Langchain	13	6	4	3	0	0
Tensorflow	46	12	10	23	0	1
Scikit-learn	7	3	2	1	1	0
Pytorch	105	39	11	41	13	1
Total	190	67	31	75	15	2

In order to evaluate the performance of the GPT-4 model in command injection vulnerability detection more comprehensively, we selected four metrics, accuracy, precision, recall and F1 Score, for in-depth analysis. These four metrics are often used to evaluate the performance of a model[43–45], and they are calculated using the formula shown below:

$$\text{Accuracy} = \frac{TP + TN}{TP + TN + FP + FN} \tag{1}$$

$$\text{Precision} = \frac{TP}{TP + FP} \tag{2}$$

$$\text{Recall} = \frac{TP}{TP + FN} \tag{3}$$

$$\text{F1 Score} = 2 \times \frac{\text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}} \tag{4}$$

Table 4 shows performance evaluation metrics of GPT-4 in the command injection vulnerability detection task. Of these, the accuracy was 75.5%, the precision was 68.4%, the recall was 81.7% and F1 Score was 74.5%.

Table 4: Performance metrics for GPT-4 model.

Metrics	Results
Accuracy	75.5%
Precision	68.4%
Recall	81.7%
F1 Score	74.5%

Answer for RQ1: The GPT-4 model analyzed 190 functions with possible command injection vulnerabilities, showing its ability to detect command injection vulnerabilities with a accuracy of 75.5%, a precision of 68.4%, a recall of 81.7% and an F1 score of 74.5%.

5.2 RQ2: Completeness

Table 3 identified 15 false-negative cases, which were actual command injection vulnerabilities in the functions that `GPT-4` did not detect. This outcome raises an important concern about the types of vulnerabilities that might be challenging for LLM-based methods to identify. As such, in this section we further analyzed these 14 cases to identify characteristics of vulnerabilities that GPT-4 might miss.

Table 5 provides a detailed list of these false-negative cases, which include one case each from the Scikit-learn and Django projects and 13 cases from the PyTorch project. Of these fourteen cases, 10 are command injection vulnerabilities related to the `subprocess` module methods, such as `subprocess.run()`, `subprocess.Popen()`, and `subprocess.check_output()`. The remaining 5 cases involve vulnerabilities in the use of the `eval()` and `exec()` function.

To understand the problem, we examined the code for each case. In 9 of the 10 cases related to the `subprocess` module, the code followed a structure similar to that shown in Listing 3. Specifically, the `args` parameter passed to `subprocess.run()` or similar methods was a list of strings, a format that GPT-4 generally disregarded as a potential command injection vulnerability. The remaining case from the Scikit-learn project involved a vulnerability where a global variable, modifiable by an attacker, was passed to the `subprocess.check_output()` method, allowing for the injection of malicious commands.

In the 5 cases from the PyTorch project that involved the `eval()` and `exec()` function, the vulnerabilities were introduced by passing parameters to `eval()` and `exec()` that could be externally modified.

These findings reveal that GPT-4 may miss certain command injection vulnerabilities, particularly when specific code structures or parameter types obscure the risk. This underscores the need for further refinement in LLM-based vulnerability detection methods to improve accuracy in identifying subtle or complex injection risks.

```
1 def candidate_function(args: List[str]):
2     return  subprocess.run(args,
3         capture_output=True,
4         check=True,
```

```
5                )
```

Listing 3: Types of command injection vulnerabilities that would be ignored by the GPT-4 model.

Table 5: Details of false negative cases and associated command injection vulnerabilities.

Project Name	Case No.	Line of Code	Method that may cause vulnerability
Scikit-learn	5	17	subprocess.check_output()
Django	6	32	subprocess.run()
Pytorch	4	67	eval()
	4.1	49	eval()
	17	69	subprocess.run()
	23	13	subprocess.run()
	24	30	exec()
	28	13	subprocess.Popen()
	30	13	subprocess.run()
	31	72	eval()
	75	13	subprocess.run()
	76	33	eval()
	79	22	subprocess.run()
	80	13	subprocess.run()
	84	35	subprocess.run()

Answer for RQ2: The LLM-based approach based on `GPT-4` model demonstrates limitations in accurately detecting command injection vulnerabilities within the `subprocess` family when list-type parameters are used (e.g., `subprocess.run(args)` or `subprocess.Popen(args)`, where `args` is a list of strings). Although vulnerabilities missed by GPT-4 due to global variables or externally modifiable parameters are relatively few, these potential security risks require attention to prevent them from being overlooked in real-world applications.

5.3 RQ3: Running Cost

In RQ3, we evaluated the runtime and financial costs associated with performing all experiments. Table 6 provides details on the number of cases, lines of code, and time spent analyzing these cases with using `GPT-4`.

Our experiment has analyzed 190 candidate functions, comprising a total of 5239 lines of code. Among these, 100 functions were identified as containing command injection vulnerabilities, for which GPT-4 analyzed 2117 lines of code and generated corresponding security tests. The total time required for this analysis was 3629.8 seconds. For the remaining 90 functions, which were determined to be free from command

injection vulnerabilities, GPT-4 analyzed 3122 lines of code, taking a total of 950.99 seconds.

Table 6: Time costs of performing experiments with GPT-4.

Analysis Result	No. of case	Line of code	Time
Yes	100	2117	3629.8
No	90	3122	950.99
Total	190	5239	4580.79

Answer for RQ3: We used the GPT-4 model to analyze a total of 190 candidate functions with a total of 5239 lines of code, of which 100 candidates were determined to have command injection vulnerabilities, and GPT generated the corresponding security tests for these 100 cases. For all experiments, the total time spent was 4580.79 seconds and the financial overhead was $14.19.

5.4 RQ4: Comparison of Performance between Different LLM tools

In this section, we compare the performance of our approach when integrating four popular Large Language Models (LLMs)—Claude 3.5 Sonnet, GPT-4o, GPT-4 and DeepSeek-R1—in tasks of command injection vulnerability detection and automated security test generation. By examining each model's accuracy in vulnerability detection and the percentage of generated security tests that can be executed without modifications, we gain insights into their capabilities and reliability, helping us identify the most suitable model for our tasks in vulnerability detection and test case generation.

5.4.1 Command Injection Vulnerability Detection Capability

Tables 7, 8 and 9 present the results of analyzing 190 candidate functions using Claude 3.5 Sonnet, GPT-4o and DeepSeek-R1. According to Claude's assessment, 156 of the 190 cases were identified as vulnerable to command injection, while 34 were considered safe. Among these, 73 were true positives, 81 were false positives, 25 were true negatives, 9 were false negatives, and 2 were invalid cases. For GPT-4o, it identified 131 cases as containing command injection vulnerabilities and 59 as safe, resulting in 66 true positives, 63 false positives, 43 true negatives, 16 false negatives, and 2 invalid cases. For DeepSeek-R1, 84 of the 190 cases were identified as vulnerable to command injection, while 106 were considered safe. Among these, 56 were true positives, 26 were false positives, 80 were true negatives, 26 were false negatives, and 2 were invalid cases.

Figure 5 illustrates the performance metrics for the four models in command injection vulnerability detection.In terms of accuracy and precision, DeepSeek-R1 and GPT-4 perform nearly identically. By contrast, GPT-4o and Claude 3.5 Sonnet score significantly lower on both metrics — hovering around 50%, roughly 20 percentage points below the results achieved by GPT-4 and DeepSeek-R1. For the F1 score, the

GPT-4 has the highest value of 74.5%. In comparison, GPT-4o and Claude 3.5 Sonnet had F1 scores of 62.5% and 61.9% respectively, which were about 10 percentage points lower than GPT-4. DeepSeek-R1's F1 score of 68.3% is about 6 percentage points lower than GPT-4. However, the recall rates for GPT-4o (80.5%) and GPT-4 (81.7%) were comparable, while Claude 3.5 Sonnet achieves the highest recall at 89%. DeepSeek-R1 records the lowest recall rate, at 68.3%.

These comparisons indicate that GPT-4 outperforms GPT-4o, Claude 3.5 Sonnet and DeepSeek-R1 in terms of overall accuracy and F1 score, making it a more reliable choice for command injection vulnerability detection. As such, GPT-4 is recommended as the preferred model for our task.

Table 7: Detection results of command injection vulnerabilities using Claude 3.5 Sonnet.

Project	No. of cases	True positive	False positive	True negative	False negative	Invalid
Django	17	6	8	3	0	0
Flask	2	2	0	0	0	0
Langchain	13	6	7	0	0	0
Tensorflow	46	12	25	8	0	1
Scikit-learn	7	4	2	1	0	0
Pytorch	105	43	39	13	9	1
Total	190	73	81	25	9	2

Table 8: Detection results of command injection vulnerabilities using GPT-4o.

Project	No. of cases	True positive	False positive	True negative	False negative	Invalid
Django	17	4	2	9	2	0
Flask	2	2	0	0	0	0
Langchain	13	6	6	1	0	0
Tensorflow	46	11	25	8	1	1
Scikit-learn	7	3	1	2	1	0
Pytorch	105	40	29	23	12	1
Total	190	66	63	43	16	2

5.4.2 The security test generation capability

Our experiment has identified that GPT-4 determined that 100 out of 190 cases had command injection vulnerabilities and generated security tests for them, among which only 55 security tests can be run directly, which indicates that GPT-4 had deficiencies in security test generation. For further comparison, we used Claude 3.5 Sonnet, GPT-4o and DeepSeek-R1 to generate security tests for these 100 cases. Figure 6 showed

Table 9: Detection results of command injection vulnerabilities using DeepSeek-R1.

Project	No. of cases	True positive	False positive	True negative	False negative	Invalid
Django	17	6	2	9	0	0
Flask	2	2	0	0	0	0
Langchain	13	6	2	5	0	0
Tensorflow	46	9	5	28	3	1
Scikit-learn	7	2	1	2	2	0
Pytorch	105	31	16	36	21	1
Total	190	56	26	80	26	2

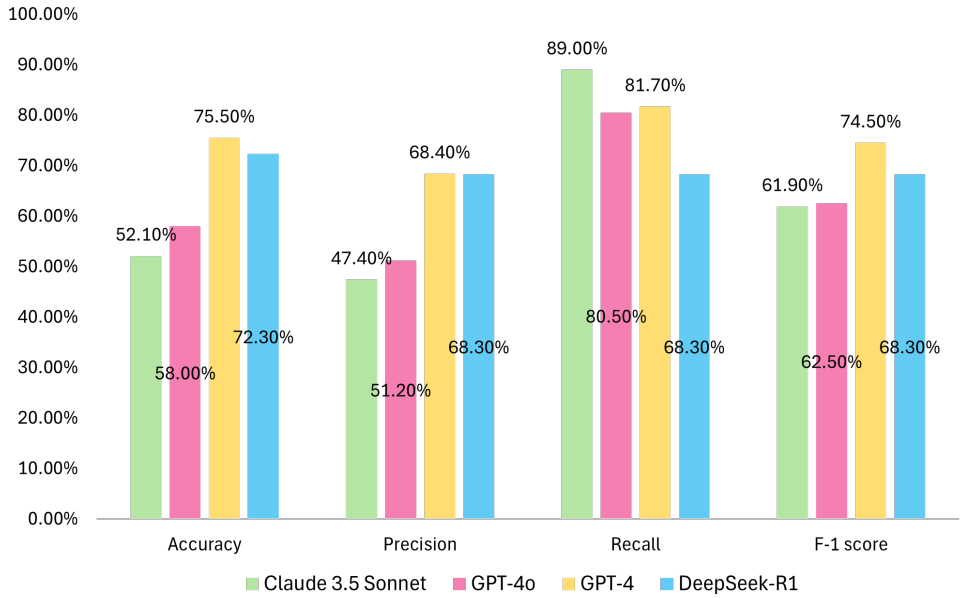


Fig. 5: The comparison results of the four LLMs in command injection vulnerabilities detection performance.

the comparison results of the four LLMs in security test generation performance. For Claude 3.5 Sonnet, the number of security tests that can be directly run is 72. The number of security tests that can be directly run for GPT-4o is 65. And for DeepSeek-R1, the number of security tests that can be directly run is 80. Among the four LLMs, DeepSeek-R1 performed best in generating security tests. It is the better choice in test generation task.

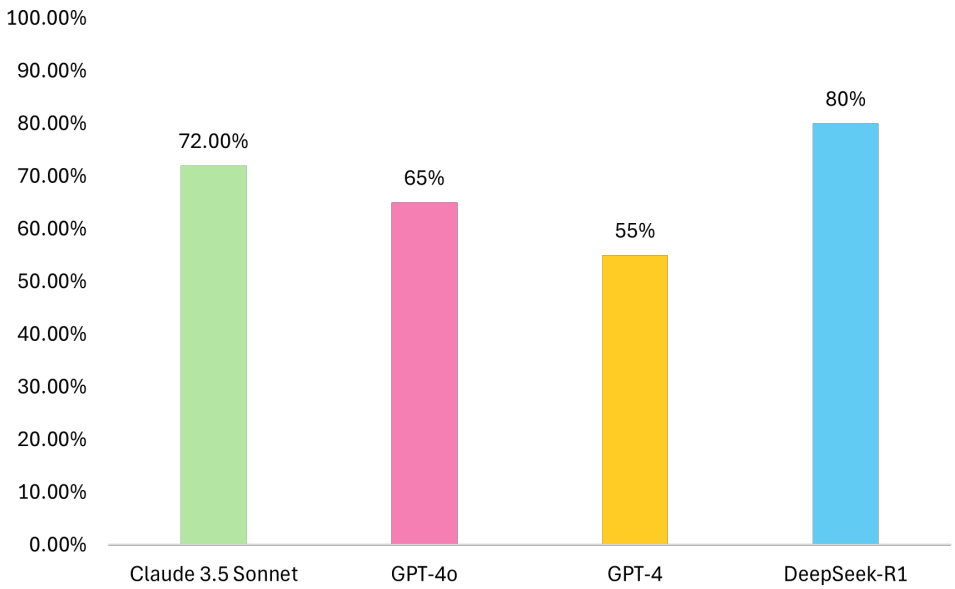


Fig. 6: The comparison results of the four LLMs in security test generation performance.

Answer for RQ4: Among the four large language models, GPT-4 showed better results in command injection vulnerability detection, while DeepSeek-R1 showed better results in security test generation. We can choose different large language models for different tasks to get better results.

5.5 RQ5: Comparison with Bandit

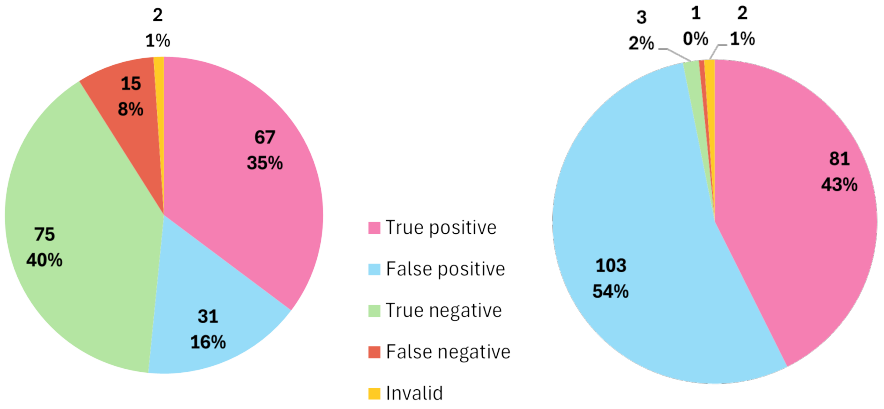
We compared our approach with an existing Python security detection tool, Bandit, focusing on command injection vulnerability analysis. Given that the GPT-4 model demonstrated the highest performance among the four large language models tested, we specifically compared GPT-4's results with those of Bandit. Table 10 show Bandit's performance for detecting vulnerabilities in 190 candidate functions. Bandit identified 186 functions as containing command injection vulnerabilities and 4 as safe. Upon verification, 81 cases were true positives (43%), 103 were false positives (54%), 3 were true negatives (1.5%), and 1 was a false negative (0.5%). Additionally, 2 cases (1%) were invalid due to environmental or dependency issues. Figure 7 shows the distribution of detection results for GPT-4 and Bandit. We can find that, in contrast, Bandit has a much higher false positive rate than GPT-4.

Figure 8 presented the performance comparison of using Bandit and GPT-4 in detecting command injection vulnerabilities. Compared to Bandit, our GPT-4 based method performs better in detecting command injection vulnerabilities with improved

Table 10: Detection results of command injection vulnerabilities using Bandit.

Project	No. of cases	True positive	False positive	True negative	False negative	Invalid
Django	17	6	10	1	0	0
Flask	2	2	0	0	0	0
Langchain	13	6	6	1	0	0
Tensorflow	46	12	32	1	0	1
Scikit-learn	7	4	3	0	0	0
Pytorch	105	51	52	0	1	1
Total	190	81	103	3	1	2

accuracy of 30.8%, precision of 24.4%, and F1 score of 13.6%. The only shortcoming is that the recall is reduced by 17.1%. The recall value of Bandit is higher than GPT-4, indicating that Bandit has stronger sensitivity to positive cases and can effectively reduce the number of false negatives. However, we observed that Bandit has a high number of false positives, which significantly lowers its precision and accuracy, with values of 44% and 44.7%, respectively. It is worth mentioning that our method is also able to identify two vulnerabilities that Bandit cannot detect. These results indicate that the large language model performs better in command injection vulnerability detection compared to static vulnerability detection tools because of its stronger contextual analysis capability.



(a) Distribution of GPT-4

(b) Distribution of Bandit

Fig. 7: The distribution of detection results for GPT-4 and Bandit.

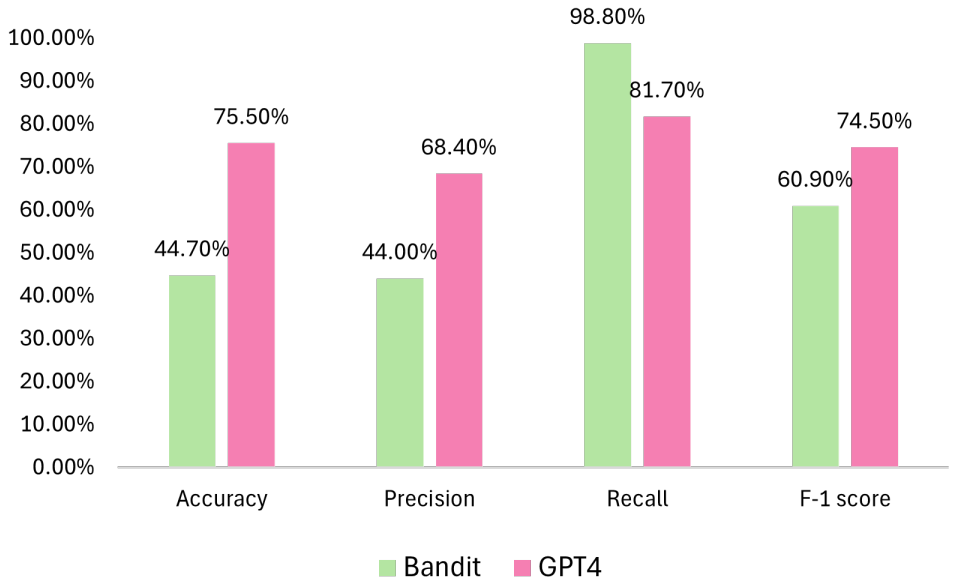


Fig. 8: Performance comparison of GPT-4 and Bandit in detecting command injection vulnerabilities.

Answer to RQ5: Our LLM-based approach outperformed Bandit by reducing false positive and false negative rates, thereby significantly improving accuracy, precision, and F1 scores. Additionally, our method successfully identified vulnerabilities that Bandit failed to detect, highlighting the enhanced detection capability of our approach.

6 Related Work

6.1 Vulnerability detection

We categorize research related to vulnerability detection into three main types: static or dynamic analysis methods, machine learning-based methods, and large language models-based methods.

Static analysis and dynamic analysis approaches Static analysis and dynamic analysis are the classical methods to detect vulnerability. Static analysis is divided into two categories: graph-based static analysis [46] and static analysis with data modeling [47]. For example, Song et al. [46] proposed an approach called BitBlaze, where a static analysis component called Vine can detect vulnerabilities using CFG, DFG, and weakest precondition calculations.

Dynamic analysis mainly consists of detection techniques such as dynamic taint analysis [48], fuzzing testing [49], and symbolic execution [50]. For example, Trickel

et al. [49] proposed a novel Web vulnerability discovery framework based on grey-box coverage-guided fuzzing called Witcher. It implements the concept of fault escalation to detect SQL and command injection vulnerabilities in web applications. **ML-based approaches:** Compared to static and dynamic analysis, machine learning based approaches handle large scale data as well as reduce false positives in vulnerability detection. Both Guo [51] and Laura [52] proposed vulnerability detection methods based on Long Short-Term Memory, where Guo's method is used to detect vulnerabilities in PHP code, while Laura's method is used to detect vulnerabilities in Python code. After evaluation, the accuracy and F1 scores of both methods are more than 80%. Stanislav [53] used a CNN-based approach to detect code injection in web application and uses data preprocessing techniques to address the large training requirements typically associated with such networks, reducing the time required to configure and train CNNs.

LLMs-based approaches: With the development of large language models, more and more people are using this technique to analyze code [9, 11] and detect vulnerabilities [42, 54, 55]. Liu et al. [55] proposed a static binary taint analysis method supported by LLM that automates the taint analysis process and outperforms state-of-the-art methods in terms of efficiency and effectiveness in identifying new errors in realistic firmware. And Sun et al. [42] combined the GPT model with static analysis to detect logical vulnerabilities in smart contracts with high accuracy. Their approach has been a great inspiration for our research.

6.2 LLMs-assisted techniques in unit test generation

Recently, as test generation has been moving towards automation, more and more people have been trying to generate tests using the emerging technique of large language models [10, 12, 13, 56–58]. Most of them use the GPT family of models (GPT-3.5 or GPT-4) or its variant GPT to generate tests. For example, Max et al. [12] propose a method for generating adaptive unit tests using a large language model called TEST-PILOT. This method is based on OpenAI's GPT-3.5-turbo model and was evaluated on 25 npm packages. The statement coverage and branch coverage of the generated tests are better than the state-of-the-art feedback-directed JavaScript test generation technique. Zhang et al. [58] evaluated the effectiveness of GPT in generating security tests for Java applications. They used GPT-4 to generate security tests for 55 applications, 40 of which could be compiled and successfully demonstrated 24 attacks. The results are better than two of the most advanced security test generators (TRANSFER and SIEGE).

7 Threats to Validity

In this section, we focus on discussing the potential internal and external threats to the validity.

Internal threats: Our study adopted a validation strategy, i.e., validating the results of LLM's analysis of the code by running the security tests generated by it. We checked all the generated security tests and manually modified the code that could not be run. Finally, in 100 security tests, 2 tests could not run due to code dependency

issues and the running environment issues. The percentage is 2%. They slightly affected the accuracy of our assessment of LLMs' ability to detect vulnerabilities.

External threats: Our study only uses four large language model(GPT-4, GPT-4o, Claude 3.5 Sonnet and DeepSeek-R1) and tests our approach in six Python projects. So there are some limitations to our research results. Future research will explore more large language models (e.g., DeBERTa, LLaMA, etc.) and add more Python projects to our dataset to further validate and generalize our findings.

8 Conclusion and future work

This study demonstrates the potential of Large Language Models (LLMs) as an effective approach for detecting command injection vulnerabilities within Python's widely used open-source libraries. Through a comprehensive analysis of over 13,000 Python files from six high-profile projects, we evaluated the strengths and limitations of LLMs in identifying security vulnerabilities that threaten system integrity and data privacy. While traditional tools like Bandit are valuable, our results show that LLMs provide a complementary advantage by analyzing fragmented and non-compilable code and detecting complex vulnerability patterns that existing tools may miss. Additionally, the ability of LLMs to generate security tests adds a useful layer of verification, potentially enhancing the accuracy of vulnerability assessments.

Our comparative analysis of different LLM tools highlights that models like GPT-4 offer adaptability for security applications, although some challenges in vulnerability detection persist. By identifying the types of vulnerabilities that LLMs might miss, our research lays a foundation for enhancing these models and guiding future developments in automated security analysis. The dataset we built, sourced from six extensively used GitHub projects, aims to support further research, establishing a benchmark for LLM-driven vulnerability detection. With these advancements, developers and security researchers can leverage LLMs to improve security practices, moving toward a more resilient open-source ecosystem.

Future work can explore refining LLMs for greater accuracy in vulnerability detection, particularly focusing on areas where current models fall short, such as complex nested code structures. Investigating hybrid models that combine LLMs with traditional static analysis tools could also yield improved detection capabilities. Additionally, expanding the dataset to include a broader range of security vulnerabilities would create a more robust benchmark for future studies.

References

[1] TensorFlow Jenkins. 2023. Tensorflow: an Open Source Machine Learning Framework. Accessed Nov 11, 2023. https://github.com/tensorflow/tensorflow

[2] Jerry Zhang. 2023. Pytorch: tensors and dynamic neural networks in Python. Accessed Nov 11, 2023. https://github.com/pytorch/pytorch

[3] Anastasios Stasinopoulos, Christoforos Ntantogian, and Christos Xenakis. 2019. Commix: automating evaluation and exploitation of command injection vulnerabilities in Web applications. International Journal of Information Security 18(2019), 49–72. https://doi.org/10.1007/s10207-018-0399-z

[4] 2023. 2023 CWE Top 25 Most Dangerous Software Weaknesses. Accessed Nov 01, 2023. https://cwe.mitre.org/top25/archive/2023/2023_top25_list.html

[5] America's Cyber Defense Agency. 2024. CISA and FBI Release Secure by Design Alert on Eliminating OS Command Injection Vulnerabilities. Accessed Aug 10, 2024. https://www.cisa.gov/news-events/alerts/2024/07/10/cisa-and-fbi-release-secure-design-alert-eliminating-os-command-injection-vulnerabilities

[6] 2023. Common Vulnerabilities and Exposures (CVE). Accessed Nov 01, 2023. https://www.cve.org/

[7] 2022. CVE-2022-29216. Accessed Nov 11, 2023. https://www.cve.org/CVERecord?id=CVE-2022-29216

[8] Python Code Quality Authority. 2023. Bandit: a tool designed to find common security issues in Python code. Accessed Nov 23, 2023. https://github.com/PyCQA/bandit

[9] Shih-Chieh Dai, Aiping Xiong, and Lun-Wei Ku. 2023. LLM-in-the-loop: Leveraging large language model for thematic analysis. arXiv preprint arXiv:2310.15100 (2023). https://arxiv.org/abs/2310.15100

[10] Vitor Guilherme and Auri Vincenzi. 2023. An initial investigation of ChatGPT unit test generation capability. In Proceedings of the 8th Brazilian Symposium on Systematic and Automated Software Testing (, Campo Grande, MS, Brazil,)(SAST '23). Association for Computing Machinery, New York, NY, USA, 15–24. https://doi.org/10.1145/3624032.3624035

[11] Shuyin Ouyang, Jie M Zhang, Mark Harman, and Meng Wang. 2023. An Empirical Study of the Non-determinism of ChatGPT in Code Generation. arXiv preprint arXiv:2308.02828 (2023). https://arxiv.org/abs/2308.02828

[12] Max Schäfer, Sarah Nadi, Aryaz Eghbali, and Frank Tip. 2024. An Empirical Evaluation of Using Large Language Models for Automated Unit Test Generation. IEEE Transactions on Software Engineering 50, 1 (2024), 85–105. https://doi.org/10.1109/TSE.2023.3334955

[13] Yutian Tang, Zhijie Liu, Zhichao Zhou, and Xiapu Luo. 2023. ChatGPT vs SBST: A Comparative Assessment of Unit Test Suite Generation. arXiv:2307.00588 [cs.SE] https://arxiv.org/abs/2307.00588

[14] OpenAI, Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, and Florencia Leoni Aleman et.al. 2023. GPT-4 Technical Report. arXiv:2303.08774 [cs.CL] https://arxiv.org/abs/2303.08774

[15] OpenAI, Inc. 2024. GPT-4o. Accessed Sep 23, 2024. https://platform.openai.com/docs/models#gpt-4o

[16] Anthropic PBC. 2024. Claude 3.5 Sonnet. Accessed Sep 23, 2024. https://www.anthropic.com/news/claude-3-5-sonnet

[17] DeepSeek-AI, Daya Guo, and Dejian Yang et al. 2025. DeepSeek-R1: Incentivizing Reasoning Capability in LLMs via Reinforcement Learning. arXiv:2501.12948[cs.CL]https://arxiv.org/abs/2501.12948

[18] Wikipedia. 2024. Large language model. Accessed Jul 23, 2024. https://en.wikipedia.org/wiki/Large_language_model

[19] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. 2023. Attention Is All You Need. arXiv:1706.03762 [cs.CL] https://arxiv.org/abs/1706.03762

[20] Xinyi Hou, Yanjie Zhao, Yue Liu, Zhou Yang, Kailong Wang, Li Li, Xiapu Luo, David Lo, John Grundy, and Haoyu Wang. 2023. Large Language Models for Software Engineering: A Systematic Literature Review. arXiv:2308.10620 [cs.SE] https://arxiv.org/abs/2308.10620

[21] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. arXiv:1810.04805 [cs.CL] https://arxiv.org/abs/1810.04805

[22] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. 2020. CodeBERT: A Pre-Trained Model for Programming and Natural Languages. arXiv:2002.08155 [cs.CL] https://arxiv.org/abs/2002.08155

[23] Zhenzhong Lan, Mingda Chen, Sebastian Goodman, Kevin Gimpel, Piyush Sharma, and Radu Soricut. 2020. ALBERT: A Lite BERT for Self-supervised Learning of Language Representations. arXiv:1909.11942 [cs.CL] https://arxiv.org/abs/1909.11942

[24] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. 2023. Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer. arXiv:1910.10683 [cs.LG] https://arxiv.org/abs/1910.10683

[25] Long Phan, Hieu Tran, Daniel Le, Hieu Nguyen, James Anibal, Alec Peltekian, and Yanfang Ye. 2021. CoTexT: Multi-task Learning with Code-Text

Transformer. arXiv:2105.08645 [cs.AI] https://arxiv.org/abs/2105.08645

[26] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. 2020. Language Models are Few-Shot Learners. arXiv:2005.14165 [cs.CL] https://arxiv.org/abs/2005.14165

[27] OpenAI, Inc. 2022. GPT-3.5. Accessed Nov 11, 2023. https://platform.openai.com/docs/models/gpt-3-5

[28] Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham, Hyung Won Chung, Charles Sutton, Sebastian Gehrmann, Parker Schuh, Kensen Shi, Sasha Tsvyashchenko, Joshua Maynez, Abhishek Rao, Parker Barnes, Yi Tay, Noam Shazeer, Vinodkumar Prabhakaran, Emily Reif, Nan Du, Ben Hutchinson, Reiner Pope, James Bradbury, Jacob Austin, Michael Isard, Guy Gur-Ari, Pengcheng Yin, Toju Duke, Anselm Levskaya, Sanjay Ghemawat, Sunipa Dev, Henryk Michalewski, Xavier Garcia, Vedant Misra, Kevin Robinson, Liam Fedus, Denny Zhou, Daphne Ippolito, David Luan, Hyeontaek Lim, Barret Zoph, Alexander Spiridonov, Ryan Sepassi, David Dohan, Shivani Agrawal, Mark Omernick, Andrew M. Dai, Thanumalayan Sankaranarayana Pillai, Marie Pellat, Aitor Lewkowycz, Erica Moreira, Rewon Child, Oleksandr Polozov, Katherine Lee, Zongwei Zhou, Xuezhi Wang, Brennan Saeta, Mark Diaz, Orhan Firat, Michele Catasta, Jason Wei, Kathy Meier-Hellstern, Douglas Eck, Jeff Dean, Slav Petrov, and Noah Fiedel. 2022. PaLM: Scaling Language Modeling with Pathways. arXiv:2204.02311 [cs.CL] https://arxiv.org/abs/2204.02311

[29] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, Aurelien Rodriguez, Armand Joulin, Edouard Grave, and Guillaume Lample. 2023. LLaMA: Open and Efficient Foundation Language Models. arXiv:2302.13971 [cs.CL] https://arxiv.org/abs/2302.13971

[30] Mansour Alqarni and Akramul Azim. 2022. Low Level Source Code Vulnerability Detection Using Advanced BERT Language Model. Proceedings of the Canadian Conference on Artificial Intelligence (may 27 2022). https://caiac.pubpub.org/pub/gdhb8oq4

[31] Xinyun Chen, Maxwell Lin, Nathanael Schärli, and Denny Zhou. 2023. Teaching Large Language Models to Self-Debug. arXiv:2304.05128 [cs.CL] https://arxiv.org/abs/2304.05128

[32] Kexun Zhang, Danqing Wang, Jingtao Xia, William Yang Wang, and Lei Li. 2023. ALGO: Synthesizing Algorithmic Programs with LLM-Generated Oracle Verifiers. arXiv:2305.14591 [cs.CL] https://arxiv.org/abs/2305.14591

[33] Tsz-On Li, Wenxi Zong, Yibo Wang, Haoye Tian, Ying Wang, Shing-Chi Cheung, and Jeff Kramer. 2023. Nuances are the Key: Unlocking ChatGPT to Find Failure-Inducing Tests with Differential Prompting. arXiv:2304.11686 [cs.SE] https://arxiv.org/abs/2304.11686

[34] Yiannis Charalambous, Norbert Tihanyi, Ridhi Jain, Youcheng Sun, Mohamed Amine Ferrag, and Lucas C. Cordeiro. 2023. A New Era in Software Security: Towards Self-Healing Software via Large Language Models and Formal Verification. arXiv:2305.14752 [cs.SE] https://arxiv.org/abs/2305.14752

[35] A. J. Thirunavukarasu, D. S. J. Ting, K. Elangovan, et al. 2023. Large language models in medicine. Nature Medicine 29(2023), 1930–1940. https://doi.org/10.1038/s41591-023-02448-8

[36] Valentin Liévin, Christoffer Egeberg Hother, Andreas Geert Motzfeldt, and Ole Winther. 2023. Can large language models reason about medical questions? arXiv:2207.08143 [cs.CL] https://arxiv.org/abs/2207.08143

[37] Mariusz Felisiak. 2023. Django: a high-level Python web framework. Accessed Nov 11, 2023. https://github.com/django/django

[38] David Lord. 2023. Flask: the Python micro framework for building web applications. Accessed Nov 11, 2023. https://github.com/pallets/flask

[39] Guillaume Lemaitre. 2023. Scikit-learn: a Python module for machine learning. Accessed Nov 11, 2023. https://github.com/scikit-learn/scikit-learn

[40] Harrison Chase. 2023. Langchain: a framework for developing applications powered by language models. Accessed Nov 11, 2023. https://github.com/langchain-ai/langchain

[41] Semgrep, Inc. 2023. Semgrepa open source static analysis tool for finding bugs, detecting vulnerabilities. Accessed Oct 25, 2023. https://semgrep.dev/docs/cheat-sheets/python-command-injection/

[42] Yuqiang Sun, Daoyuan Wu, Yue Xue, Han Liu, Haijun Wang, Zhengzi Xu, Xiaofei Xie, and Yang Liu. 2023. GPTScan: Detecting Logic Vulnerabilities in Smart Contracts by Combining GPT with Program Analysis. arXiv:2308.03314 [cs.CR] https://arxiv.org/abs/2308.03314

[43] S. Chakraborty, R. Krishna, Y. Ding and B. Ray, "Deep Learning Based Vulnerability Detection: Are We There Yet?," in IEEE Transactions on Software Engineering, vol. 48, no. 9, pp. 3280-3296, 1 Sept. 2022, doi:

10.1109/TSE.2021.3087402.

[44] Wartschinski, Laura, Yannic Noller, Thomas Vogel, Timo Kehrer, and Lars Grunske. "VUDENC: vulnerability detection with deep learning on a natural codebase for Python." Information and Software Technology 144 (2022): 106809.

[45] G. Huang, Y. Li, Q. Wang, J. Ren, Y. Cheng and X. Zhao, "Automatic Classification Method for Software Vulnerability Based on Deep Neural Network," in IEEE Access, vol. 7, pp. 28291-28298, 2019, doi: 10.1109/ACCESS.2019.2900462.

[46] Dawn Song, David Brumley, Heng Yin, Juan Caballero, Ivan Jager, Min Gyung Kang, Zhenkai Liang, James Newsome, Pongsin Poosankam, and Prateek Saxena. 2008. BitBlaze: A New Approach to Computer Security via Binary Analysis. In Information Systems Security, R. Sekar and Arun K. Pujari (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 1–25.

[47] Fabian Yamaguchi, Nico Golde, Daniel Arp, and Konrad Rieck. 2014. Modeling and Discovering Vulnerabilities with Code Property Graphs. In 2014 IEEE Symposium on Security and Privacy. 590–604. https://doi.org/10.1109/SP.2014.44

[48] Jun Cai, Peng Zou, Jinxin Ma, and Jun He. 2016. wordDTA: A dynamic taint analysis tool for software vulnerability detection. Wuhan University Journal of Natural Sciences 21, 1 (2016), 10–20.

[49] Erik Trickel, Fabio Pagani, Chang Zhu, Lukas Dresel, Giovanni Vigna, Christopher Kruegel, Ruoyu Wang, Tiffany Bao, Yan Shoshitaishvili, and Adam Doupé. 2023. Toss a Fault to Your Witcher: Applying Grey-box Coverage-Guided Mutational Fuzzing to Detect SQL and Command Injection Vulnerabilities. In 2023 IEEE Symposium on Security and Privacy (SP). 2658–2675. https://doi.org/10.1109/SP46215.2023.10179317

[50] Hongzhe Li, Taebeom Kim, Munkhbayar Bat-Erdene, and Heejo Lee. 2013. Software vulnerability detection using backward trace analysis and symbolic execution. In 2013 International Conference on Availability, Reliability and Security. IEEE, 446–454.

[51] Ning Guo, Xiaoyong Li, Hui Yin, and Yali Gao. 2020. VulHunter: An Automated Vulnerability Detection System Based on Deep Learning and Bytecode. In Information and Communications Security, Jianying Zhou, Xiapu Luo, Qingni Shen, and Zhen Xu (Eds.). Springer International Publishing, Cham, 199–218.

[52] Laura Wartschinski, Yannic Noller, Thomas Vogel, Timo Kehrer, and Lars Grunske. 2022. VUDENC: Vulnerability Detection with Deep Learning on a Natural Codebase for Python. Information and Software Technology 144 (April 2022), 106809. https://doi.org/10.1016/j.infsof.2021.106809

[53] Stanislav Abaimov and Giuseppe Bianchi. 2019. CODDLE: Code-Injection Detection With Deep Learning. IEEE Access 7 (2019), 128617–128627. https://doi.org/10.1109/ACCESS.2019.2939870

[54] Sihao Hu, Tiansheng Huang, Fatih İlhan, Selim Furkan Tekin, and Ling Liu. 2023. Large Language Model-Powered Smart Contract Vulnerability Detection: New Perspectives. arXiv:2310.01152 [cs.CR] https://arxiv.org/abs/2310.01152

[55] Puzhuo Liu, Chengnian Sun, Yaowen Zheng, Xuan Feng, Chuan Qin, Yuncheng Wang, Zhi Li, and Limin Sun. 2023. Harnessing the Power of LLM to Support Binary Taint Analysis. arXiv:2310.08275 [cs.CR] https://arxiv.org/abs/2310.08275

[56] Zhiqiang Yuan, Yiling Lou, Mingwei Liu, Shiji Ding, Kaixin Wang, Yixuan Chen, and Xin Peng. 2023. No More Manual Tests? Evaluating and Improving ChatGPT for Unit Test Generation. arXiv:2305.04207 [cs.SE] https://arxiv.org/abs/2305.04207

[57] Zhuokui Xie, Yinghao Chen, Chen Zhi, Shuiguang Deng, and Jianwei Yin. 2023. ChatUniTest: a ChatGPT-based automated unit test generation tool. arXiv:2305.04764 [cs.SE] https://arxiv.org/abs/2305.04764

[58] Ying Zhang, Wenjia Song, Zhengjie Ji, Danfeng, Yao, and Na Meng. 2023. How well does LLM generate security tests? arXiv:2310.00710 [cs.CR] https://arxiv.org/abs/2310.00710

Project Name	Case No.	Line of code	Method that may cause vulnerability	GPT's answer	Gpt run time (seconds)	Test Code Executable directly?	Reason	Executable with modification?	Actually vulnerable?
Flask	Case 1	39	eval()	Yes	80.38	No	Don't set path	Yes	Yes
	Case 2	22	exec()	Yes	48.07	No	Path setting error	Yes	Yes

Project Name	Case No.	Line of code	Method that may cause vulnerability	GPT's answer	Gpt run time (seconds)	Test Code Executable directly?	Reason	Executable with modification?	Actually vulnerable?
Scikit-learn	Case 1	49	subprocess.run()	No	11.76	N/A	N/A	N/A	No
	Case 2	26	exec()	Yes	44.81	Yes	N/A	N/A	No
	Case 3	46	subprocess.check_output()	Yes	54.48	Yes	N/A	N/A	Yes
	Case 4	18	eval()	Yes	24.94	No	Missing json file	Yes	Yes
	Case 5	17	subprocess.check_output()	No	11.18	N/A	N/A	N/A	Yes
	Case 6	15	subprocess.run()	Yes	55.86	Yes	N/A	N/A	No
	Case 7	20	subprocess.run()	Yes	53.44	No	Missing source code call	Yes	Yes

Project Name	Case No.	Line of code	Method that may cause vulnerability	GPT's answer	Gpt run time (seconds)	Test Code Executable directly?	Reason	Executable with modification?	Actually vulnerable?
Langchain	Case 1	19	os.system()	Yes	35.23	No	URL not compatible	Yes	No
	Case 2	7	exec()	Yes	23.58	Yes	N/A	N/A	No
	Case 3	16	exec()	Yes	37.8	Yes	N/A	N/A	Yes
	Case 4	30	exec()	Yes	50.24	No	Missing source code call	Yes	Yes
	Case 5	16	exec()	Yes	40.77	Yes	N/A	N/A	Yes
	Case 6	144	subprocess.run()	No	22.35	N/A	N/A	N/A	No
	Case 7	38	subprocess.run()	Yes	48.83	No	Missing source code call	Yes	No
	Case 8	66	subprocess.run()	No	20.11	N/A	N/A	N/A	No
	Case 9	25	subprocess.run()	No	14.08	N/A	N/A	N/A	No
	Case 10	18	exec	Yes	34.72	Yes	N/A	N/A	Yes
	Case 11	28	eval(), exec()	Yes	48.67	No	Missing necessary function call	Yes	Yes
	Case 12	23	subprocess.run()	Yes	45.22	Yes	N/A	N/A	Yes
	Case 13	18	os.system()	Yes	37.69	No	URL not compatible	Yes	No

Project Name	Case No.	Line of code	Method that may cause vulnerability	GPT's answer	Gpt run time (seconds)	Test Code Executable directly?	Reason	Executable with modification?	Actually vulnerable?
Django	Case 1	21	subprocess.run()	No	15.26	N/A	N/A	N/A	No
	Case 2	18	subprocess.run()	Yes	40.32	Yes	N/A	N/A	No
	Case 3	29	subprocess.run()	No	11.84	N/A	N/A	N/A	No
	Case 4	7	subprocess.run()	No	6.57	N/A	N/A	N/A	No
	Case 5	6	subprocess.run()	Yes	25.81	Yes	N/A	N/A	Yes
	Case 6	32	subprocess.run()	No	8.79	N/A	N/A	N/A	Yes
	Case 7	12	exec()	Yes	33.63	Yes	N/A	N/A	Yes
	Case 8	12	subprocess.run()	Yes	40.24	No	Miss library call	Yes	No
	Case 9	51	exec()	Yes	55.81	No	Missing source code call	Yes	Yes
	Case 9.1	26	exec()	Yes	47.2	No	Missing library call, typeerror	Yes	Yes
	Case 10	34	exec()	No	5.76	N/A	N/A	N/A	No
	Case 10.1	39	exec()	No	11.57	N/A	N/A	N/A	No
	Case 11	14	eval()	No	8.29	N/A	N/A	N/A	No
	Case 12	20	eval()	Yes	43.94	Yes	N/A	N/A	No
	Case 13	11	subprocess.run()	No	9.06	N/A	N/A	N/A	No
	Case 14	39	eval()	Yes	52.79	No	The usage make code can't be execute	Yes	Yes
	Case 15	26	subprocess.Popen()	Yes	44.87	No	IndexError: list assignment index out of range	Yes	No

Project Name	Case No.	Line of code	Method that may cause vulnerability	GPT's answer	Gpt run time (seconds)	Test Code Executable directly?	Reason	Executable with modification?	Actually vulnerable?
TensorFlow	Case 1	14	subprocess.check_output()	No	22.46	N/A	N/A	N/A	No
	Case 2	43	os.system()	Yes	58.52	No	Command injection method wrong	Yes	Yes
	Case 3	14	subprocess.check_output()	Yes	37.77	Yes	N/A	N/A	Yes
	Case 4	13	exec()	No	14.23	N/A	N/A	N/A	No
	Case 5	30	subprocess.check_output	Yes	57.76	No	ValueError and attribute error	Yes	No
	Case 6	4	os.system()	Yes	38.17	No	Missing import file	Yes	No
	Case 7	6	exec()	Yes	27.75	Yes	N/A	N/A	No
	Case 8	21	subprocess.check_output	No	16.62	N/A	N/A	N/A	No
	Case 9	41	eval()	Yes	49.82	Yes	N/A	N/A	Yes
	Case 10	4	exec()	Yes	37.68	No	Syntax error in comment injection	Yes	No
	Case 10.1	7	exec()	Yes	49.34	Yes	N/A	N/A	No
	Case 11	28	eval()	No	7.56	N/A	N/A	N/A	No
	case 12	39	os.system()	Yes	53.61	No	no attribute 'graph_def'	Yes	No
	Case 13	100	subprocess.run()	No	18.71	N/A	N/A	N/A	No
	Case 14	22	eval()	Yes	45.67	Yes	N/A	N/A	Yes
	Case 15	16	subprocess.check_output()	Yes	29.66	No	Command injection format error	Yes	Yes
	Case 16	85	subprocess.check_output()	No	14.94	N/A	N/A	N/A	No
	Case 17	22	exec()	Yes	48.78	No	Incorrect name of the calling library	Yes	No
	Case 18	26	subprocess.check_output()	Yes	45.67	Yes	N/A	N/A	No
	Case 19	24	subprocess.call()	Yes	59.68	No	Missing some dependency functions	Yes	No
	Case 19.1	20	subprocess.call()	No	13.53	N/A	N/A	N/A	No
	Case 20	21	subprocess.call()	No	22.36	N/A	N/A	N/A	No
	Case 20.1	29	subprocess.call()	No	14.03	N/A	N/A	N/A	No
	Case 21	16	subprocess.check_output()	Yes	35.62	No	Command injection format wrong	Yes	Yes
	Case 22	15	exec()	Yes	28.75	No	failed to account for the check on the command	Yes	Yes
	Case 23	27	subprocess.check_output()	No	12.99	N/A	N/A	N/A	No
	Case 24	21	subprocess.check_output()	No	19.27	N/A	N/A	N/A	No
	Case 24.1	12	subprocess.check_output()	No	12	N/A	N/A	N/A	No
	Case 25	30	os.system()	Yes	47.03	No	Use mock, can't see expected result	Yes	Yes
	Case 26	26	subprocess.call()	Yes	38.83	No	name 'test_utils' is not defined	Yes	Pending
	Case 27	22	exec()	No	15.25	N/A	N/A	N/A	No
	Case 28	19	subprocess.check_output()	No	5.38	N/A	N/A	N/A	No
	Case 29	14	exec()	Yes	38.45	No	Key error	Yes	Yes
	Case 30	4	exec()	Yes	27.87	No	Code injection comment error	Yes	Yes
	Case 31	11	eval()	No	7.94	N/A	N/A	N/A	No
	Case 32	20	subprocess.check_output()	No	10.71	N/A	N/A	N/A	No
	Case 33	9	subprocess.run()	No	8.04	N/A	N/A	N/A	No
	Case 34	14	subprocess.check_output()	No	20.08	N/A	N/A	N/A	No
	Case 35	14	subprocess.check_output()	No	12.66	N/A	N/A	N/A	No
	Case 35.1	7	subprocess.check_output()	No	5.91	N/A	N/A	N/A	No
	Case 36	115	subprocess.Popen()	Yes	58.32	No	Missing function and library call	Yes	No
	Case 37	16	subprocess.Popen()	Yes	45.07	Yes	N/A	N/A	Yes
	Case 38	27	subprocess.Popen()	No	15.24	N/A	N/A	N/A	No
	Case 39	11	subprocess.Popen()	Yes	37.88	No	Command injection format error	Yes	Yes
	Case 40	22	subprocess.Popen()	No	7.34	N/A	N/A	N/A	No
	Case 41	16	subprocess.Popen()	No	6.92	N/A	N/A	N/A	No

Project Name	Case No.	Line of code	Method that may cause vulnerability	GPT's answer	Gpt run time (seconds)	Test Code Executable directly?	Reason	Executable with modification?	Actually vulnerable?
PyTorch	Case 1	51	exec()	Yes	44.2	No	Parameter setting error	Yes	Yes
	Case 2	19	subprocess.run()	Yes	35.89	Yes	N/A	N/A	Yes
	Case 3	42	subprocess.check_output()	No	16.21	N/A	N/A	N/A	No
	Case 4	67	eval()	No	4.26	N/A	N/A	N/A	Yes
	Case 4.1	49	eval()	No	17.2	N/A	N/A	N/A	Yes
	Case 5	3	exec()	Yes	36.55	Yes	N/A	N/A	Yes
	Case 6	2	exec()	Yes	31.15	No	NameError: name 'os' is not defined	Yes	Yes
	Case 7	20	exec()	No	10.42	N/A	N/A	N/A	No
	Case 8	5	subprocess.run()	No	14.87	N/A	N/A	N/A	No
	Case 9	14	eval()	Yes	36.87	Yes	N/A	N/A	Yes
	Case 10	35	subprocess.check_output()	No	8.48	N/A	N/A	N/A	No
	Case 11	36	subprocess.run()	No	15.19	N/A	N/A	N/A	No
	Case 12	18	subprocess.check_output()	No	4.26	N/A	N/A	N/A	No
	Case 13	7	exec()	Yes	24.26	Yes	N/A	N/A	Yes
	Case 14	7	subprocess.check_output()	Yes	47.32	No	Check approach wrong	Yes	Yes
	Case 15	14	subprocess.check_output()	Yes	33.92	Yes	N/A	N/A	Yes
	Case 16	9	eval()	Yes	21.99	No	Missing source function	Yes	No
	Case 17	69	subprocess.run()	No	7.49	N/A	N/A	N/A	Yes
	Case 18	19	subprocess.run()	No	7.85	N/A	N/A	N/A	No
	Case 19	25	subprocess.run(), subprocess.check_output()	No	6.79	N/A	N/A	N/A	No
	Case 20	36	exec()	Yes	19.80	Yes	N/A	N/A	Yes
	Case 21	14	subprocess.run()	Yes	28.72	Yes	N/A	N/A	Yes
	Case 22	128	subprocess.run(), subprocess.check_output()	Yes	51.32	No	No module named conda	No	Pending
	Case 23	13	subprocess.run()	No	16.13	N/A	N/A	N/A	Yes
	Case 24	31	exec()	No	10.36	N/A	N/A	N/A	Yes
	Case 25	32	exec()	Yes	69.59	Yes	N/A	N/A	Yes
	Case 26	7	subprocess.run()	Yes	31.45	No	Command injection method error	Yes	Yes
	Case 27	43	subprocess.check_output()	No	12.52	N/A	N/A	N/A	No
	Case 28	13	subprocess.Popen()	No	11.36	N/A	N/A	N/A	Yes
	Case 28.1	10	subprocess.Popen()	Yes	26.75	Yes	N/A	N/A	Yes
	Case 29	9	eval()	Yes	47	Yes	N/A	N/A	No
	Case 30	13	subprocess.run()	No	11.47	N/A	N/A	N/A	Yes
	Case 31	72	eval()	No	8.34	N/A	N/A	N/A	Yes
	Case 32	38	exec()	No	4.94	N/A	N/A	N/A	No
	Case 33	6	subprocess.check_output()	No	7.84	N/A	N/A	N/A	No
	Case 34	16	subprocess.run()	Yes	30.42	Yes	N/A	N/A	Yes
	Case 35	70	subprocess.check_output()	No	7.13	N/A	N/A	N/A	No
	Case 36	20	subprocess.run()	Yes	44.68	Yes	N/A	N/A	Yes
	Case 37	66	exec()	No	14.83	N/A	N/A	N/A	No
	Case 38	10	subprocess.check_output()	Yes	41.77	Yes	N/A	N/A	No
	Case 39	23	exec()	No	11.9	N/A	N/A	N/A	No
	Case 40	15	subprocess.check_output()	Yes	45.68	Yes	N/A	N/A	No
	Case 41	83	exec()	No	13.22	N/A	N/A	N/A	No
	Case 42	53	subprocess.run()	No	8.14	N/A	N/A	N/A	No
	Case 43	40	exec()	No	8.18	N/A	N/A	N/A	No
	Case 44	16	exec()	No	22.5	N/A	N/A	N/A	No
	Case 45	44	subprocess.Popen()	No	7.97	N/A	N/A	N/A	No
	Case 46	14	subprocess.run()	Yes	31.65	Yes	N/A	N/A	Yes
	Case 47	17	subprocess.check_output()	Yes	48.1	Yes	N/A	N/A	No
	Case 48	89	subprocess.check_output()	No	15.21	N/A	N/A	N/A	No
	Case 49	7	exec()	Yes	36.45	Yes	N/A	N/A	Yes
	Case 50	23	subprocess.run()	Yes	45.36	Yes	N/A	N/A	No

Project Name	Case No.	Line of code	Method that may cause vulnerability	GPT's answer	Gpt run time (seconds)	Test Code Executable directly?	Reason	Executable with modification?	Actually vulnerable?
PyTorch	Case 51	13	subprocess.run()	Yes	47.9	No	Command injection method error	Yes	Yes
	Case 52	52	exec()	Yes	37.92	No	Missing source code call	Yes	Yes
	Case 53	17	subprocess.run()	No	7.62	N/A	N/A	N/A	No
	Case 54	49	subprocess.run()	No	12.7	N/A	N/A	N/A	No
	Case 55	9	subprocess.call()	Yes	28.72	Yes	N/A	N/A	No
	Case 56	18	subprocess.run()	Yes	48.79	No	Command injection method error	Yes	Yes
	Case 57	41	eval()	No	14.57	N/A	N/A	N/A	No
	Case 58	34	exec()	No	9.83	N/A	N/A	N/A	No
	Case 59	19	exec()	No	5.77	N/A	N/A	N/A	No
	Case 59.1	49	subprocess.run()	No	9.94	N/A	N/A	N/A	No
	Case 60	8	eval()	Yes	20.32	No	Test approach issue	Yes	Yes
	Case 61	5	exec()	Yes	26.58	Yes	N/A	N/A	Yes
	Case 62.1	24	subprocess.check_output()	No	3.65	N/A	N/A	N/A	No
	Case 62	20	subprocess.check_output()	No	6.8	N/A	N/A	N/A	No
	Case 63	4	eval()	Yes	22.45	No	Command injection method error	Yes	Yes
	Case 64	11	exec()	Yes	13.67	Yes	N/A	N/A	Yes
	Case 65	3	exec()	Yes	17.14	No	NameError: name 'os' is not defined	Yes	Yes
	Case 66	51	eval()	No	4.37	N/A	N/A	N/A	No
	Case 67	22	subprocess.check_output()	No	3.81	N/A	N/A	N/A	No
	Case 68	24	subprocess.run()	No	6.39	N/A	N/A	N/A	No
	Case 69	22	subprocess.run()	Yes	22.38	Yes	N/A	N/A	Yes
	Case 70	27	subprocess.check_output()	No	4.36	N/A	N/A	N/A	No
	Case 71	40	exec()	No	7.38	N/A	N/A	N/A	No
	Case 72	313	exec()	No	11.25	N/A	N/A	N/A	No
	Case 73	37	os.system()	Yes	18.01	No	Command injection method error	Yes	Yes
	Case 74	8	exec()	Yes	12.84	Yes	N/A	N/A	Yes
	Case 75	13	subprocess.run()	No	5.66	N/A	N/A	N/A	Yes
	Case 76	7	eval()	No	7.14	N/A	N/A	N/A	Yes
	Case 77	17	subprocess.run()	No	6.27	N/A	N/A	N/A	No
	Case 78	37	subprocess.run()	No	3.90	N/A	N/A	N/A	No
	Case 79	22	subprocess.run()	No	5.57	N/A	N/A	N/A	Yes
	Case 80	13	subprocess.run()	No	5.53	N/A	N/A	N/A	Yes
	Case 81	59	subprocess.run()	Yes	19.68	Yes	N/A	N/A	No
	Case 82	2	exec()	Yes	8.95	No	NameError: name 'os' is not defined	Yes	Yes
	Case 83	8	subprocess.run()	Yes	13.97	Yes	N/A	N/A	Yes
	Case 84	35	subprocess.run()	No	6.28	N/A	N/A	N/A	Yes
	Case 85	4	subprocess.run()	Yes	10.27	Yes	N/A	N/A	Yes
	Case 86	31	subprocess.Popen()	No	5.82	N/A	N/A	N/A	No
	Case 87	14	subprocess.Popen()	Yes	13.77	Yes	N/A	N/A	Yes
	Case 88	14	subprocess.Popen()	Yes	11.55	Yes	N/A	N/A	Yes
	Case 89	40	subprocess.Popen()	Yes	17.56	Yes	N/A	N/A	No
	Case 90	29	subprocess.Popen()	No	8.55	N/A	N/A	N/A	No
	Case 91	3	subprocess.Popen()	No	4.37	N/A	N/A	N/A	No
	Case 92	11	subprocess.Popen()	Yes	17.23	Yes	N/A	N/A	Yes
	Case 93	20	subprocess.Popen()	No	3.81	N/A	N/A	N/A	No
	Case 94	15	subprocess.Popen()	Yes	18.71	No	TypeError: WorkerTimerArgs() takes no arguments	Yes	Yes
	Case 95	10	subprocess.Popen()	Yes	17.53	Yes	N/A	N/A	No
	Case 96	9	subprocess.Popen()	Yes	15.18	Yes	N/A	N/A	Yes
	Case 97	10	subprocess.Popen()	Yes	13.81	Yes	N/A	N/A	Yes
	Case 98	11	subprocess.Popen()	Yes	13.59	Yes	N/A	N/A	Yes
	Case 99	58	subprocess.Popen()	No	9.92	N/A	N/A	N/A	No
	Case 100	20	subprocess.Popen()	Yes	21.57	Yes	N/A	N/A	No
	Case 101	11	subprocess.Popen()	Yes	18.75	Yes	N/A	N/A	Yes