

TALLINN UNIVERSITY OF TECHNOLOGY  
School of Information Technologies

Karl Tamberg 221954IVCM

**HARNESSING LARGE LANGUAGE MODELS FOR  
SOFTWARE VULNERABILITY DETECTION: A  
COMPREHENSIVE BENCHMARKING STUDY**

Master's Thesis

Supervisor: Hayretin Bahsi  
PhD

Tallinn 2024

TALLINNA TEHNIKAÜLIKOOL  
Infotehnoloogia teaduskond

Karl Tamberg 221954IVCM

**SUURTE KEELEMUDELITE KASUTAMINE  
TURVANÕRKUSTE TUVASTAMISEKS LÄHTEKOODIS:  
PÕHJALIK VÕRDLUSUURING**

Magistritöö

Juhendaja: Hayreddin Bahsi  
PhD

Tallinn 2024

## **Author's Declaration of Originality**

I hereby certify that I am the sole author of this thesis. All the used materials, references to the literature and the work of others have been referred to. This thesis has not been presented for examination anywhere else.

Author: Karl Tamberg

10.05.2024

## Abstract

Writing code without introducing vulnerabilities is a difficult task. To help software developers catch issues, different approaches have been proposed, including manual reviews, static analysis, dynamic analysis and deep learning techniques. Nonetheless, the number of vulnerabilities reported yearly shows an upward trend. There could be many reasons, like lack of awareness or the best practices and tools not being effective enough. To help combat some issues with traditional vulnerability detection tools, we propose using large language models (LLMs) to assist in finding vulnerabilities in source code. LLMs have shown a remarkable ability to understand and generate code, underlining their potential in code-related tasks. The aim is to test multiple state-of-the-art LLMs and identify the best prompting strategies, allowing extraction of the best value from the LLMs. We provide an overview of the strengths and weaknesses of the LLM-based approach and compare the results to those of traditional static analysis tools. We find LLMs can pinpoint many more issues than traditional static analysis tools, outperforming the traditional tools in terms of recall and F1 scores. The results should benefit software developers and security analysts who are responsible for making sure the code does not contain vulnerabilities.

The thesis is written in English and is 102 pages long, including 8 chapters, 1 figure and 16 tables.

## **Annotatsioon**

### **Suurte keelemudelite kasutamine turvanõrkuste tuvastamiseks lähtekoodis: põhjalik võrdlusuuring**

Turvanõrkusteta koodi kirjutamine on keeruline ülesanne. Välja on töötatud mitmeid tööriistu ning meetodikaid, et aidata tarkvaraarendajatel leida turvanõrkused varajastes arendustsükli etappides. Näiteks manuaalsed koodi läbivaatused, koodi staatiline analüüs, koodi dünaamiline analüüs ja erinevad süvaõppe meetodid. Sellegipoolest on iga-aastaselt raporteeritud turvanõrkuste arv kasvutrendis. Põhjuseid võib selleks olla mitmeid, näiteks teadmatus või parimate tavade ja tööriistade ebaefektiivsus. Me teeme ettepaneku suurte keelemudelite kasutamiseks turvanõrkuste tuvastamise eesmärgil. Suured keelemudelid on näidanud märkimisväärset võimekust koodi mõistmise ja genereerimise ülesannetes, rõhutades nende potentsiaali kõiksugu koodiga seotud ülesannetes. Eesmärk on testida mitmeid keelemudeleid ja tuvastada parimad päringustrateegiad, mis võimaldaksid keelemudeleid võimalikult tõhusalt kasutada. Me avaldame ülevaate keelemudelite tugevustest ning nõrkustest võrdluses traditsiooniliste staatiliste koodianalüsaatoritega. Keelemudelid on võimelised saavutama paremaid saagise ning F1 tulemusi kui traditsioonilised tööriistad. Me leiame, et keelemudelid suudavad tuvastada rohkem erinevaid probleeme kui traditsioonilised staatilise koodi kontrollimise tööriistad. Tulemused peaksid olema kasulikud tarkvaraarendajatele ja turvaanalüütikutele, kelle vastutada on see, et kood ei sisaldaks haavatavusi.

Lõputöö on kirjutatud inglise keeles ning sisaldab teksti 102 leheküljel, 8 peatükki, 1 joonist, 16 tabelit.

## List of Abbreviations and Terms

AI	Artificial Intelligence
API	Application Programming Interface
BERT	Bidirectional Encoder Representations from Transformers
CoT	Chain of Thought
CPU	Central Processing Unit
CSRF	Cross-Site Request Forgery
CSV	Comma Separated Value
CVE	Common Vulnerabilities and Exposures
CWE	Common Weakness Enumeration
DL	Deep Learning
FN	False Negative
FP	False Positive
GPT	Generative Pre-trained Transformers
GPU	Graphics Processing Unit
HTML	HyperText Markup Language
HTTP	HyperText Transfer Protocol
HTTPS	HyperText Transfer Protocol Secure
IaC	Infrastructure as Code
IDE	Intelligent Development Environment
LLM	Large Language Model
ML	Machine Learning
MRQ	Main Research Question
NLP	Natural Language Processing
OS	Operating System
OWASP	Open Worldwide Application Security Project
RAG	Retrieval Augmented Generation
RCI	Recursive Criticism and Improvement
RQ	Research Question
SAMATE	Software Assurance Metrics and Tool Evaluation
SAST	Static Application Security Testing
SQL	Structured Query Language
TN	True Negative
ToT	Tree of Thoughts

TP  
VAT

True Positive  
Value Added Tax

# Table of Contents

<b>1</b>	<b>Introduction</b>	<b>10</b>
<b>2</b>	<b>Background</b>	<b>13</b>
2.1	Manual approaches to improving code security	13
2.1.1	Manual security code review	13
2.1.2	Manual penetration testing	14
2.2	Automated approaches to improving code security	14
2.2.1	Static code analysis	14
2.2.2	Dynamic code analysis	15
2.2.3	Hybrid code analysis	16
2.2.4	Code analysis with artificial intelligence	16
2.3	Large language models	17
2.4	Prompting LLMs	18
2.4.1	Zero-shot and few-shot prompting	19
2.4.2	Chain of thought prompting	19
2.4.3	Tree of thoughts prompting	20
2.4.4	Self-consistency	20
2.4.5	ReAct prompting	20
2.4.6	Utilising hints	21
2.5	Datasets for benchmarking	21
<b>3</b>	<b>Related work</b>	<b>22</b>
3.1	Prior studies review	22
3.1.1	LLMs as good vulnerability detection tools	22
3.1.2	LLMs outperformed by other approaches	25
3.1.3	LLMs as a tool in vulnerability detection framework	27
3.1.4	LLMs with vulnerability detection as secondary focus	30
3.2	Review summary	32
3.3	Identified research gaps	33
<b>4</b>	<b>Methods</b>	<b>35</b>
4.1	LLMs and prompting strategies	35
4.2	Dataset	35
4.2.1	Dataset pre-processing	37
4.3	Evaluation methods	38



<b>5</b>	<b>Results</b>	<b>41</b>
5.1	Static code analysis tools	41
5.1.1	CodeQL	41
5.1.2	SpotBugs	43
5.2	Vulnerability detection with LLMs	44
5.2.1	Establishing baseline prompting strategy	45
5.2.2	Comparing prompting approaches from prior studies	46
5.2.3	Custom prompting strategies	48
5.2.4	Strategies requiring higher temperature values	50
5.2.5	Different models	52
5.2.6	Results per CWE category	52
5.3	Quantitative analysis summary	54
5.4	Qualitative analysis	55
5.4.1	True positive classifications	55
5.4.2	False positive classifications	57
<b>6</b>	<b>Discussion</b>	<b>60</b>
6.1	Findings	60
6.1.1	RQ1: What prompting approach is most successful with the LLMs to detect vulnerabilities?	60
6.1.2	RQ2: What advantages and disadvantages do LLMs have over existing static analysis tools?	62
6.1.3	MRQ: How would the use of off-the-shelf LLMs be able to contribute to vulnerability detection in source code?	63
6.1.4	Differences and similarities of LLMs	64
6.2	Limitations and threats to validity	64
<b>7</b>	<b>Summary</b>	<b>67</b>
<b>8</b>	<b>Future work</b>	<b>68</b>
	<b>References</b>	<b>69</b>
	<b>Appendix 1 – Non-Exclusive License for Reproduction and Publication of a Graduation Thesis</b>	<b>77</b>
	<b>Appendix 2 – Prompts</b>	<b>78</b>
	<b>Appendix 3 – Code Samples</b>	<b>96</b>
	<b>Appendix 4 – Versions</b>	<b>100</b>

## List of Figures

1	<i>An image of the CWE-36 hierarchy. . . . .</i>	36
---	--	----

## List of Tables

1	<i>CWE's overlap in MITRE top 25 and Juliet 1.3</i> . . . . .	36
2	<i>Classification logic</i> . . . . .	39
3	<i>Mapping strategy</i> . . . . .	40
4	<i>CodeQL results</i> . . . . .	41
5	<i>CodeQL-esq results per CWE</i> . . . . .	42
6	<i>SpotBugs results</i> . . . . .	43
7	<i>SpotBugs-fsb results per CWE</i> . . . . .	43
8	<i>Baseline prompting results</i> . . . . .	46
9	<i>Prompts from prior studies results</i> . . . . .	47
10	<i>Custom prompting results</i> . . . . .	48
11	<i>Results from initial testing of higher-temperature strategies</i> . . . . .	50
12	<i>Self-consistency results</i> . . . . .	51
13	<i>Other models results</i> . . . . .	52
14	<i>GPT-4 <math>p_{cot-8s}</math> prompt results per CWE</i> . . . . .	53
15	<i>Claude 3 Opus <math>p_{cot-8s}</math> prompt results per CWE</i> . . . . .	53
16	<i>Results overview</i> . . . . .	54

# 1. Introduction

As software gets integrated into many business processes, a lot of new software code is written every day. The use of high-level languages and frameworks takes a lot of responsibility off the developer's shoulders. Microsoft estimates about 70% of the vulnerabilities in their systems are due to memory safety [1]. Higher-level languages add abstraction and take care of memory management behind the scenes, improving the security posture. However, the added abstraction from high-level programming languages does not address all vulnerabilities. These problems need to be discovered by some other means.

The records hosted in the Common Vulnerabilities and Exposures (CVE) database show a trend of incline of reported vulnerabilities over the years [2]. In the past five years alone, 109,864 CVEs have been reported [2]. This means, on average, around 21,973 vulnerabilities per year and around 60 vulnerabilities in a day. Even though it is positive that the vulnerabilities are found, the problem is they are found too late – after being released.

To avoid releasing vulnerable software, good processes and tools must be used in the software development lifecycle. Many efforts have been made in this field, including the automation of some software development and deployment processes. Automatic vulnerability detection can be integrated into both the development tooling and code review processes. Most tools work as black-boxes, meaning there is no need for the user to understand how the tool works, the user just provides the code and receives the results. However, the techniques used to detect vulnerabilities or bad coding practices can be ineffective [3, 4].

The large language models (LLMs) have become capable at handling novel tasks based on minimal prompts, without requiring extensive fine-tuning as was common before 2020 [5]. Prompts allow for manipulation of the LLMs in a way that can make the models return expected output without additional task-specific training [6]. While multiple studies have tested the vulnerability detection capabilities, they do not build on the results of previous studies and do not tackle the benchmarking of prompting techniques systematically [7, 8, 9, 10]. The advantage of LLMs over traditional analysis tools is their flexibility: LLMs can be asked follow-up questions or asked to help fix the problematic code. The code generation capabilities of LLMs have been already recognised and a commercial version called Codex has been developed for code generation purposes [11].

We evaluate the performance of state-of-the-art off-the-shelf large language models (LLMs) as black-box static code analysers. The aim is to comprehensively benchmark different prompting approaches across multiple LLMs and to provide practical suggestions on using LLMs for vulnerability detection. The research questions are:

- MRQ: How would the use of off-the-shelf LLMs be able to contribute to vulnerability detection in source code?
- RQ1: What prompting approach is most successful with the LLMs to detect vulnerabilities?
- RQ2: What advantages and disadvantages do LLMs have over existing static analysis tools?

We use a synthetic dataset called Juliet Java 1.3 to evaluate existing tools against LLMs [12]. Due to the high costs of running LLMs, we compile a subset for running the experiments. The dataset is pre-processed by refactoring everything that could provide textual clues about the vulnerability type and presence. Then a strategy for automatically classifying the LLM outputs is established. Different prompting approaches are tried and the results are reported. For comparison, we use two state-of-the-art static code analysis tools, CodeQL [13] and SpotBugs [14]. SpotBugs has shown good results in comparison to other traditional static analysis tools on the Juliet dataset [15, 16] and CodeQL has been previously utilised as a comparison point for testing LLM capabilities [7].

The main contribution includes testing state-of-the-art prompting techniques on our dataset. The dataset is curated to be small enough to allow experimentation without very high costs while still containing significantly more CWE categories than most previous studies. For the initial experiments, the GPT-4 turbo model from OpenAI is utilised, as it has a relatively large context window, is relatively cheap to use and is beloved by the community [17]. Then we run the best-performing strategies on two more models, GPT-4 and Claude 3 Opus, which people have voted to be among the best ones available at the time of writing [17]. We include prompting strategies not previously considered for vulnerability detection tasks, like the tree of thoughts (ToT) [18] and self-consistency [19]. We propose novel prompts containing detailed instructions that outperform other previously proposed strategies in most settings. The results are compared with those of two traditional static analysis tools to better display the capabilities of LLMs. We discuss the strengths and weaknesses of LLM-based vulnerability detection in comparison to traditional static analysis tools. To the best of our knowledge, we are the first to include the Claude 3 Opus model from Anthropic in a vulnerability detection benchmark study. What is more, we provide detailed costs per prompting strategy and discuss the cost as a factor in choosing LLMs and prompting strategies. Cost is a factor that has not been explored in detail by prior studies in the same

domain [7, 8, 9, 10].

The first major part of the thesis covers the background and the related work. Then a detailed description of the contribution is provided, containing a detailed description of the dataset, CWE matching strategy and the results. Lastly, the results, key findings and limitations are discussed.

## **2. Background**

Most of us interact with software every day, as many important services, both private and public, rely on software solutions. Software vulnerabilities can pose significant risks if left undetected. Any vulnerable code which gets deployed has the potential to be exploited by a threat actor. Therefore, the vulnerabilities should be found in the early stages of the software development lifecycle and most importantly before the vulnerable code gets deployed into a production environment. This way the impact of the code flaws is minimal, and the fixes are easy and quick to implement. The difficulty of writing code without any flaws, which could introduce vulnerabilities, has been recognised and many different approaches to help developers find issues in their code have been developed. Not only that, but many of these tools have also been successfully used in the industry for years.

### **2.1 Manual approaches to improving code security**

Manual code review and testing are a part of everyday activities for most software developers. This helps to find issues in the code before the code gets released to the production environment. Finding issues in the early stages of the software development lifecycle means the cost of fixing the issues is low. The earlier issues are found, the faster they can be fixed and the less the users of the product are affected.

#### **2.1.1 Manual security code review**

Manual security code review is a form of static code analysis, where humans check code for security vulnerabilities. This is very time-consuming and expects the reviewer to know what vulnerabilities to look for [20]. The code reviewers are usually expected to pay attention to not only security issues but also business logic flaws and alignment with the agreed-upon standards. This makes it very likely even the most competent reviewers would miss some vulnerabilities at some point.

The effectiveness of manual code reviews has been studied by hiring 30 developers to do security-focused code reviews for code with known security vulnerabilities [21]. This research had two interesting outcomes. Firstly, none of the hired developers managed to find all the security issues, showing most developers cannot discover all relevant vulnerabilities during manual reviews [21]. Secondly, some developers managed to find security issues, which were at the time unknown even to the conductors of the research

[21]. This highlights that even though relying only on manual reviews is not reasonable, manual reviews do have their benefits and should not be completely replaced.

The effort should therefore be directed towards making the manual code reviews as little taxing for the manual reviewers as possible. This means using tools to point out issues. However, having a large set of different tools, which are configured to report every potential issue they find, can backfire. This could cause a large number of false positive results and validating all the reported issues is a taxing task for the developers [22]. Thus, it is important to find compromises between true positives and false positives [22]. This means the tooling should be configured in such a manner in which most of the reported issues are actual problems [22]. That way the developers will not feel like their time is being wasted.

### **2.1.2 Manual penetration testing**

Manual penetration testing consists of manually testing the application during runtime to find or confirm security issues. Usually, the application is viewed as a black-box and the testers provide different inputs and perform actions which are likely to cause issues if something has been misconfigured. Manual penetration often uses some tools to automate some of the steps in the testing process. This approach should be applied in the later stages of the software development lifecycle. While manual penetration has the potential to uncover issues missed in the earlier stages, it comes with multiple issues of its own. This approach is very time-consuming, relies on the tester to be familiar with all the application flows and is likely to miss some vulnerabilities [23].

## **2.2 Automated approaches to improving code security**

Automated approaches allow to discover and flag potential vulnerabilities without manual effort. There are different approaches, each with its upsides and downsides. The idea of helping developers identify issues beyond the ones flagged by the compiler has been around for a long time. The first such tool for static analysis was released in the year 2000 [20].

### **2.2.1 Static code analysis**

Automated static code analysis applies pre-defined rules or algorithms on the source code, deriving a list of potential vulnerabilities [24]. This type of scan is known as the inference method and different tools can use slightly different techniques [24]. In theory, most vulnerabilities could be detected with static analysis techniques [24]. In practice, static



code analysis tools are limited by the vulnerability types they can detect [3].

The more common vulnerabilities like different injection and overflow vulnerabilities are detected by many tools, while others such as CSRF vulnerabilities will often stay undetected [24]. These tools work by examining the source code, meaning most tools are by design programming language specific [24]. Even though most existing tools support multiple programming languages, adding and maintaining support for different languages requires extra effort. Generally, most static analysis tools are prone to produce many false positive results [24, 25]. What is more, the tools can miss the vulnerabilities related to the misuse of third-party libraries. Adding the rules to support all kinds of frameworks and programming languages is a time-consuming and tedious process.

The advantages of static analysis include not having to execute the code, meaning there are no issues related to the reachability of vulnerabilities [24]. Reachability means all of the code in the codebase can be analysed [24]. Static analysis allows for a quick turnaround for fixes, as it can point to the exact location in the code where the vulnerability lies [26]. What is more, the tooling is often simple and relatively fast to use, which means the code analysis can be done on local machines.

Drawing from Google's expertise, the ability to analyse code locally can offer a significant advantage over more complex approaches [22]. Google tested several strategies for running static code analysis and reporting the results: displaying problems in the bug dashboard, manually filing bug reports, incorporating the feedback into code review processes, and reporting errors at the compiler level [22]. The findings indicated that ideally, code analysis should occur at the compiler level, with the second best option being during the code review process [22]. This approach improves the turnaround time for finding and fixing the issues.

### **2.2.2 Dynamic code analysis**

Dynamic code analysis is the analysis of code behaviour in runtime [25]. This approach consists of running the program in specific circumstances, during which the behaviour of the program is monitored [25]. Dynamic analysis approaches can struggle with reachability, meaning not all of the code can be analysed [24]. For large programs, every if statement will exponentially raise the number of branches in code that need to be reached. Reaching all the branches can quickly get very costly and difficult. The best-known approach to dynamic analysis is fuzz testing [27].

Fuzz testing itself can be divided into two subcategories. Black-box fuzzing consists

of purposefully feeding the program invalid or random data [28]. This is a simple and effective technique but is very likely to miss many specific code flows, which depend on particular values [28]. White-box fuzzing consists of dynamically building inputs by symbolically executing code and gathering input constraints based on conditional branches in the source code [28].

Dynamic code analysis is expected to have fewer false positives than static analysis, but more false negatives [3, 26]. This is related to the number of different possibilities for each field to be tested in every API and the states the program can achieve during dynamic testing [26]. A positive aspect is that many dynamic approaches allow the analysis of applications without access to source code [26]. A known issue with this approach is that it is hard to trace vulnerabilities back to specific parts of the code [26].

### **2.2.3 Hybrid code analysis**

Hybrid code analysis is a mixture of static and dynamic approaches, allowing them to combine their results for better outcomes. For example, a hybrid code analysis approach could use the warnings from the static analysis tools as input and run dynamic analysis to either verify or discard these warnings [24]. The weakness of the prior example would be, that no more vulnerabilities would be found compared to just running static analysis, only some false positive results would be discarded [24]. In general, this technique strives to get the best of both worlds from the static and dynamic tools [24].

### **2.2.4 Code analysis with artificial intelligence**

Machine learning (ML) and deep learning (DL) approaches have been utilised to help detect vulnerabilities within source code. Most of these approaches could be categorised under static analysis, as the source code does not need to be executed to run the analysis.

Surveys looking at vulnerability analysis via ML categorise the methods into three major approaches: vulnerable code pattern-based, software metrics-based, and anomaly-based [27, 29]. The vulnerable code pattern-based approach involves using ML algorithms to learn vulnerable code patterns from samples [30]. These ML algorithms mostly use supervised learning and the extracted patterns can be used to detect and locate vulnerabilities [29]. The main drawback of this approach is the need to include domain experts in the feature extraction process, which is a time-consuming, error-prone and task-specific process [30]. The software metrics-based approach employs an ML (most often supervised) methodology to construct a predictive model, using established software metrics like the

feature set or code complexity as features [27, 29]. Subsequently, these models are used to evaluate the vulnerability status of software artefacts by analysing the software engineering metrics [27]. These metrics are not direct indicators of vulnerabilities and have not shown good results in vulnerability detection tasks [27]. Anomaly-based approaches employ an unsupervised learning strategy to automatically either derive a model representing normal behaviour or to extract rules from the source code [27]. Deviations from the established norm or rules are classified as vulnerabilities [27]. The main drawback of the anomaly-based approach is that it can only be used for detecting small sets of vulnerability types [29].

To help combat the problems of ML-based vulnerability detection approaches, multiple DL approaches have been tested. The layered structure of DL models is claimed to be better at capturing complex patterns in source code [30]. What is more, the DL approaches allow the feature extraction processes to be automated, requiring less manual labour-intensive tasks [30]. The objectives for DL-based vulnerability analysis have been divided into three main categories: binary, multi-class and fine-grained vulnerability detection [29]. Binary vulnerability detection is used to identify whether files contain vulnerabilities [29]. Multi-class detection is used to identify which vulnerability classes are present in files [29]. Fine-grained vulnerability detection is used to pinpoint the location of the vulnerability in files [29]. The main issues with current deep learning vulnerability detection systems are the focus on a single programming language and the use of API function calls to locate vulnerabilities [29]. Having the training and testing datasets only in one programming language raises the question of whether the approach can be extended to other programming languages. Using API function calls to locate vulnerabilities makes it difficult to discover the exact location of the vulnerable code within the source code. Furthermore, the datasets used in DL vulnerability prediction studies are often too simple and do not translate to real-world use cases, suffering from imbalanced data and failing to address code semantics [31].

### **2.3 Large language models**

Natural language processing (NLP) has been an area of interest for computer scientists since the 1950s, with the latest large breakthrough happening in 2018 [5]. This is when the self-supervised neural network was shown to produce good results just by getting exposed to an extremely large amount of text [5]. The self-supervised approach means the model can create its prediction challenges to train on [5]. For example by masking a word or a phrase in a text and then trying to guess it [5]. These types of deep learning models are called large language models (LLMs) [5].

The dominant approach to tackle NLP tasks has been the use of a transformer neural network [5]. This is a complex layered approach, not just a program utilising the statistics behind word associations [5]. It is argued these models learn many facts about the world, as it helps the model to solve the word prediction challenges better [5]. As a result, these models are claimed to have a good factual knowledge of the world they are exposed to [5]. Before 2020, the standard way of using language models for specific use cases was by fine-tuning the model with the help of supervised learning [5]. However, the most recent models have proven their ability to complete novel tasks with just a prompt [5]. This means the most recent LLMs only need a description of a task or a few examples to be able to understand and tackle a task.

Some of these models have recently been trained on a dataset focusing on computer code and shown good results. Most notably, OpenAI developed a Codex model with the ability to understand both the human language and computer code [11]. This model powers a commercial tool for helping developers write code called GitHub Copilot [11]. The use of GitHub Copilot has already provoked a lot of discussions in the software engineering community. There are arguments, that this tool should be used with caution, as the generated code has been shown to contain vulnerabilities [32]. However, code written by human software developers often also contains vulnerabilities. We are interested in whether LLMs are also able to find vulnerabilities in code similar to humans.

The LLMs have different model sizes, with the model size typically referring to the number of parameters or weights that the model contains. Increased model size has been shown to improve the capabilities of the model [33, 34]. The commercial LLM APIs allow users to influence the behaviour of the LLM through different API parameters. We utilise a temperature parameter, which controls the randomness of the output. The lower temperature values make the LLM provide more deterministic responses and the higher values more creative responses [35, 36].

## **2.4 Prompting LLMs**

Instructions, context, and examples passed to a language model are called prompts. Prompts allow to guide the LLMs in a way that can make the models return expected output without additional task-specific training [6]. As the behaviour of the LLM depends on the specific wording and type of the prompt, different approaches for creating the best prompts for the given task have arisen.

Prompt engineering is a term used for describing the process of creating a prompt in a manner which helps the consumer of the prompt to understand the task the best [6]. It

is used to make the LLMs behave in the most efficient way possible for the given task. Prompts can be either manually or automatically created. Manual prompt creation is very common and requires a human to come up with a task description. Automatic prompt creation usually leverages sample data. Providing the LLM samples of expected outputs for specific inputs, the LLM can be asked to create a prompt that describes the task [37]. The automatically generated prompts perform decently, however, they are often outperformed by manually created prompts [37]. The automatic approaches have gotten better over time: some show results comparable to manual prompt creation [38].

OpenAI, the company behind the GPT large language models, has published a simple guide for prompt engineering [39]. They suggest writing clear and detailed instructions, using delimiter symbols to indicate distinct parts of the prompt and specifying the output format [39]. It is also suggested to ask the LLM to adopt a persona and divide the task into smaller subtasks where possible [39]. Providing examples is also recognised to be helpful in some cases [39]. When asking the LLM to evaluate something, it is also suggested to instruct the model to come up with its own solution to a problem before doing the evaluation [39]. Asking the LLM to re-evaluate or improve its previous response is also mentioned as a potential way to improve the final output of the LLM. There exist multiple variations of the response re-evaluation tactic, like recursive criticism and improvement [40] or self-reflection [7] or self-refinement [41].

### **2.4.1 Zero-shot and few-shot prompting**

Zero-shot and few-shot prompting are terms often used in the context of prompt engineering approaches for LLMs. Zero-shot stands for an approach where the machine learning model might not be trained for such a task and the prompt does not include examples [42]. For many use cases, this approach is easier to use but requires the model to generalise, leading often to worse results than few-shot approaches [43]. Few-shot learning stands for an approach where the pre-trained model is offered a few examples in the prompt, often together with a task description [44]. This can help improve the task-fulfilling capabilities of the language model [33]. In-context learning is a term describing an approach similar to few-shot learning. In-context learning refers to first providing LLM examples of inputs and outputs and then only providing input and asking it to provide an output [33].

### **2.4.2 Chain of thought prompting**

Chain of thought (CoT) prompting has been proven to work well when asking LLMs to solve complex problems which are reasonable to tackle in multiple steps [34]. The CoT

approach can be used in conjunction with either zero-shot or few-shot approaches. A zero-shot CoT requires the prompt to tell the LLM to solve the task step-by-step. A few-shot CoT usually provides the sample output with step-by-step operations and explanations of how the output was created. While the CoT is great for some complex problems, self-reflection and recursive criticism and improvement strategies have been shown to perform better in other cases [40, 7].

### **2.4.3 Tree of thoughts prompting**

Further development of the CoT prompting called the tree of thoughts (ToT) prompting has been developed [18]. The tree of thoughts approach facilitates the exploration of each step in the problem-solving chain [18]. It makes language models evaluate various reasoning paths and makes them self-assess their choices [18]. This process enables them to determine the subsequent steps, anticipate future actions, and backtrack when needed to make informed decisions [18]. It has shown improvements over CoT approaches for some tasks [18].

### **2.4.4 Self-consistency**

Self-consistency involves sampling diverse outputs from a language model and then selecting the most consistent answer from that set [19]. This approach can help improve the consistency of LLM responses when less deterministic settings, like higher temperature values, are used with LLMs [19]. To further advance the capabilities of the self-consistency approach for the CoT prompts, the DiVeRSe approach is suggested [45]. DiVeRSe approach uses various prompts to get diverse reasoning paths, then uses a verifier function to get the most consistent result after each step [45]. In the case of vulnerability detection tasks, another LLM could be used as the verifier function, however, that would increase the costs considerably. What is more, that approach would then start to get quite similar to the ToT prompting.

### **2.4.5 ReAct prompting**

Solving some problems can benefit from having the capability to run an action, like querying an API. The idea is to allow the LLM to first reason which steps should be taken to solve a problem and then act on those steps. This approach is called ReAct prompting, where the LLM is given access to tools, which allow the LLM to run actions like querying external APIs [46]. Adding external memory, self-reflection and planning capabilities to the ReAct approach has been shown to improve the capabilities of this approach even

further [47, 48]. This can be very helpful for solving problems, where finding the solution is hard, but verifying it is easy.

### **2.4.6 Utilising hints**

There are also methods for automatically or manually improving existing prompts, no matter how the prompt was originally created. For example, using hints to enrich the prompts has been shown to improve the LLM performance and a framework has been proposed to automate the hint generation [49]. Hints help to give context to the LLM and are based on the failed predictions made by LLMs [49]. This approach shows a lot of promise for specialised tasks, however, it might be less helpful for more general prompts [49].

## **2.5 Datasets for benchmarking**

To benchmark different tools and draw conclusions, a dataset is needed. The conducted surveys show most available datasets focus on C/C++ vulnerabilities and are based on real-world sources [50]. While both real-world datasets and synthetic datasets have their advantages, synthetic datasets can offer advantages for such benchmarking studies. The most important advantage of synthetic datasets for this study is that synthetic datasets are usually structured as a compileable project. This is a strong requirement, as we are interested in running static analysis tools, which need to compile the code base before evaluating it.

The Juliet Java 1.3 dataset was chosen to conduct this particular study [12]. This synthetic dataset has been created by the Software Assurance Metrics and Tool Evaluation (SAMATE) to evaluate software vulnerability analysis tools [51]. Juliet Java test suite has over 28,000 test cases for 112 different vulnerabilities (CWEs) [12]. This dataset has been used for previous studies on static analyser effectiveness, which give a good overview of the best-performing tools currently available [16, 15].

### **3. Related work**

As the LLMs have become much more powerful in recent years, the research in this domain is still very new. Most of the papers being released in this domain are not yet peer-reviewed and published in journals or conferences, as these processes take time. Based on titles and abstracts, the most relevant papers found within this domain are included in the review. The relevant papers are identified based on the abstracts and those papers are used for forward and backward snowballing.

#### **3.1 Prior studies review**

A recent systematic literature review studies the papers published between 2018 and 2024 that focus on the use of LLMs for vulnerability detection and program repair tasks [52]. They find mostly open-source models like CodeBert are tested for vulnerability detection tasks in the given timeframe. Around 82% of the papers cover fine-tuning approaches and the remaining 18% test prompting approaches, with zero-shot prompting being the most popular tested prompting strategy. This literature review only compares the usage of zero-shot and few-shot prompting approaches and does not provide statistics about other types of prompting like CoT. Only 18% of papers covering prompting approaches and the only prompting keywords being zero-shot and few-shot highlights the gap in the literature for studies utilising more advanced prompting strategies.

##### **3.1.1 LLMs as good vulnerability detection tools**

A comprehensive study evaluating the effectiveness of LLMs for detecting vulnerabilities in source code has been conducted before [7]. Two synthetic datasets, OWASP and Juliet, and one real-world dataset, CVEFixes, are used for benchmarking. The datasets span two programming languages, Java and C++. GPT-4 and two CodeLlama models are used with different prompting approaches. The scope is narrowed to focus on MITRE's top 25 vulnerabilities. The best F1 scores on synthetic Java datasets are produced using the GPT-4 model utilising a "Dataflow analysis-based prompt with Self Reflection" [7], producing F1 scores between 0.79 and 0.86. It must be noted the authors reduce the challenge to binary classification, providing the expected CWE value in the prompt and asking if that vulnerability is present. For the real-world Java dataset, however, CodeLlama models performed better than GPT-4, producing an F1 score of 0.55 compared to the 0.48 of the GPT-4 model. Across all datasets, GPT-4 shows the best performance on average. The



performance of LLMs is slightly better on the C++ datasets, however, the differences are negligible. For comparison with a static code analysis tool, CodeQL is used. The results show CodeQL performing slightly better than GPT-4 on the Juliet Java dataset, GPT-4 performing slightly better on the OWASP dataset and GPT-4 performing noticeably better on the Juliet C/C++ dataset. For comparison with a deep-learning tool, the LineVul vulnerability prediction tool is used on C/C++ datasets. On real-world datasets, LineVul produces poor results, however, on a synthetic dataset, LineVul produces an impressive 1.0 F1 score. The authors hypothesise this to be related to the repeating code patterns in synthetic datasets. What is more, the authors try fine-tuning smaller LLMs like GPT-3.5 and CodeLlama-7B, showing major performance improvements on synthetic datasets, but very limited improvements on real-world datasets. The improvements of fine-tuned LLMs on synthetic datasets are likely also related to the repeating code patterns. The binary classification prompt used is somewhat unfair, as it is compared to existing tools like CodeQL. CodeQL does fine-grained classification, which is a much more difficult task than binary classification. What is more, while the Juliet Java dataset contains a large number of samples, the chosen samples contain vulnerabilities only from four different CWE categories. Only four prompting strategies are discussed in detail and the costs associated with each approach are not discussed.

A handful of prompting strategies have been systematically tested for vulnerability detection purposes [8]. Java and C/C++ datasets are used and the results of GPT-4 are compared with two existing deep learning tools. The GPT-4 outperforms both tools for a given dataset even with the most basic prompting strategies. For the simple prompts, role-based prompting is shown to yield better results than prompts without a role. What is more, GPT-4 is shown to be biased towards the wording used in the prompt. That means asking the LLM if the code is buggy is more likely to produce the answer "yes" than "no". For a more elaborate strategy, API call sequence and dataflow descriptions are provided to the LLM. The authors show the order of the items provided to the LLM plays an important role. The best results are achieved by the prompt that provides first the description of the task, then the API calls descriptions and lastly the code snippet. For the CoT approaches, the LLM is first asked to explain the intent of the provided code and then to verify whether the code is vulnerable. Interestingly, this did not yield better results than previous strategies. The authors treat vulnerability detection as a binary classification challenge, asking only if the code is buggy, not what bugs the code contains. The Java experiments are done on a diverse dataset containing 50 different CWE categories. The temperature value used in the study is not discussed, which means most likely the default value of one is used.

Different OpenAI GPT models with zero-shot prompting have been tested for vulnerability detection tasks on six different real-world software projects [9]. A real-world dataset

containing multiple public scientific repositories is used. The results are compared with static analysis tools Snyk<sup>1</sup> and Fortify<sup>2</sup>. It is found that GPT-4 performs noticeably better than any of its counterparts, including the static analysis tools. However, the study provided no precision, recall and F1 scores for different approaches, with the main focus on true positive results. Interestingly, the research identified that requesting a fix for the vulnerability forces the model to explain its vulnerability identification and justify most inaccuracies or hallucinated responses. The reported low false positive rate was also notable, as most other works using zero-shot prompting have reported high false positive rates. This might be related to both the wording of the prompt and the fact the LLM was asked to also find a fix for the reported vulnerability. The temperature value used is not discussed, however, the default value for ChatGPT is one, which is most likely what is used. Other prompting strategies are not discussed in detail and the cost aspect is not discussed.

The performance of CodeBert, GPT-3.5 and GPT-4 for real-world vulnerability datasets spanning multiple programming languages has been compared [10]. The vulnerability detection is considered a binary classification challenge, with GPT-4 showing F1 scores of 76.4%, outperforming CodeBert and GPT-3.5 by a significant margin. Different simple few-shot prompts are tested, with the main difference between prompts being the provided examples. The best prompting approach for GPT-4 is a few-shot solution, with the task description containing examples from the MITRE top 25 vulnerabilities. The dataset itself contains 78 different CWE categories. The temperature values are not provided and costs associated with prompting strategies are not discussed in detail.

ChatGPT's abilities for finding OWASPs top 10 vulnerabilities in PHP source code have been tested [53]. The dataset containing 92 vulnerable files is compiled by the authors specifically for this research. For comparison purposes, 11 different static analysis tools are tasked with finding vulnerabilities from the same dataset. Zero-shot prompting is used to query the ChatGPT, which results in ChatGPT correctly identifying between 62% and 68% of vulnerabilities. The best of the tested static analysis tools are only able to detect around 32% of vulnerabilities. Even combining multiple static analysers would still only have a rate of 53% in the best case. However, very high false positives for the ChatGPT approach are reported, around 91%. Unfortunately, there is no discussion on whether different prompting approaches are tried to help lower the false positives. The authors claim the high true positive percentage shows ChatGPT has potential in vulnerability detection tasks. No exact numbers for precision, recall and F1 scores are provided. The F1 score in particular would have been interesting to see, as it also encompasses false positive and false negative

---

<sup>1</sup><https://snyk.io/>

<sup>2</sup><https://www.opentext.com/products/fortify-static-code-analyzer>

results. The temperature value is not discussed, but as ChatGPT is used, most likely the temperature is left on the default value of one. The non-deterministic nature of LLMs is discussed and the experiments are run twice to show the variance in results. The task is set as a multi-class classification and a simple role-based zero-shot prompt is used.

Vulnerability semantics guided prompting (VSP), which is a form of a few-shot CoT prompting, has been proposed for vulnerability analysis tasks [54]. The idea behind this prompting approach is to ask the LLM to only focus on relevant parts of the code and analyse the data and control flows. Three types of tasks are tested: binary vulnerability classification (vulnerability identification), multi-class vulnerability classification (vulnerability discovery) and vulnerability repair. The evaluation involves utilising the GPT-3.5, Llama2, and Falcon models with the temperature value set to zero. Two separate datasets are employed for this analysis: one is synthetically generated, while the other is based on real-world data. The study focuses on examining five distinct vulnerabilities: CWE-125, CWE-190, CWE-416, CWE-476, and CWE-787. Overall, the results are better on the synthetic dataset, with the main reason most likely related to insufficient context in the real-world dataset. Overall, five prompts are tested, containing zero-shot, few-shot, CoT and the proposed VSP strategies. The proposed VSP prompting shows good F1 scores across all tasks and models. While the general outline of the VSP prompt is provided, the exact wording with the examples is not provided for the tested prompts.

The security code reviews of infrastructure as code (IaC) repositories have been explored as areas where LLMs could assist [55]. The findings revealed this approach can detect most issues the traditional static analyser can, with the added advantage of proposing fixes. The mentioned drawbacks include the pricing model of commercial LLMs and the time it takes to get an answer compared to static analysis tools. Another issue pointed out is that the AI model makes assumptions, which might confuse users. The authors base the results and evaluation on a very small set of limited examples. Even so, the LLM still misses some vulnerabilities and reports some wrong or misleading ones. The prompt engineering topic is not mentioned in the paper, which means the provided template could most likely be improved, which in turn would also improve the results. The ability to give out suggestions and explanations is a large upside, which not many previous papers focus on. For experiments, the GPT-4 model is used with a temperature value of 0.5.

### **3.1.2 LLMs outperformed by other approaches**

The authors of Codex, the LLM that also powers Github Copilot, also consider applying the Codex model for vulnerability discovery [56]. They find the model does not perform well in comparison to static analysis tools for vulnerability detection. While no bench-

marking results or techniques are discussed, it is claimed their testing does not reveal any cases where Codex outperforms static analysis tools. However, more capable models are recognised as potentially performing better in identifying vulnerabilities and the need for further research in the area is emphasised.

GPT-3 and GPT-3.5 have been tested as black-box static analysers for finding vulnerabilities in Java code [57]. The benchmarking is done on a rather small real-world dataset containing 120 samples, where half of the samples are vulnerable. The vulnerable samples contain one of five different CWE categories. Both binary and multi-class classifications are attempted. The results show the chosen LLM models performed no better than a dummy classifier. However, the authors do not explore many different prompting approaches. Different zero-shot prompts are tried with the temperature set to zero, but they are all rather simplistic, which might be one reason behind the poor results. What is more, the results are only compared to a dummy classifier, as opposed to a static analysis tool. It would have been interesting to see how well a static analysis tool performs on their dataset and if that would have performed better than a dummy classifier. The authors proposed that GPT-4 should also be tried for such tasks as it could achieve better results. They also acknowledge improvements to prompting approaches could improve the results.

The capabilities of eight open-source and three commercial LLMs have been studied for detecting vulnerabilities in a real-world dataset [58]. The dataset is rather small, containing 100 files, half of which are vulnerable. Five binary classification prompts are tested, with the basic and few-shot prompts showing the best results in most settings. Some CoT strategies, like few-shot CoT are tested, but these do not show good results. They find all tested prompts on all LLMs perform rather poorly for vulnerability detection tasks. While GPT-4 and StarChat- $\beta^3$  show the best results, they only slightly outperform the random-guess baseline. They find in most cases the description of the problem presented by LLMs contains errors. The ability of the LLMs to pinpoint the line of code containing vulnerabilities is tested and the abilities are compared to those of human developers using the DbgBench<sup>4</sup> dataset. This contains 27 bugs and for this task, the GPT-3.5 model shows the best performance, correctly locating four problems. On average, the human developers performed significantly better than the tested LLMs, with the best human correctly locating all 27 problems.

The LLMs are shown to struggle with providing correct, understandable, concise, consistent and compliant responses when tasked with reviewing code for vulnerabilities [59]. This conclusion is reached after conducting tests with Gemini Pro, GPT-3.5 and GPT-4 LLMs

---

<sup>3</sup><https://huggingface.co/HuggingFaceH4/starchat-beta>

<sup>4</sup><https://dbgbench.github.io/>

on a real-world code review dataset. For vulnerability detection, five different prompts are tested. A prompt tasking the LLM to find vulnerabilities and provide the category as one of the ten high-level research view vulnerabilities (CWE-1000) is shown to perform the best. This could be seen as a simplified multi-class classification, as the LLM only needs to select between ten high-level categories. The temperature used with the LLMs is not disclosed. In case the default temperature value of one is used, it could contribute to the answers containing unrelated unnecessary information and being incompliant with the instructions. What is more, the prompting strategy could be the reason behind the vague answers from the LLM. The first level in the CWE research view by definition only contains very vague definitions of vulnerabilities.

The ability of GPT models to automatically pinpoint vulnerable code sections, recognise and explain the vulnerabilities, estimate the severity and suggest fixes has been studied [60]. The testing is done on a real-world C/C++ dataset containing over 190,000 functions. Both GPT-4 and GPT-3.5-turbo models perform significantly worse than other tested open-source tools, which include CodeBERT, GraphCodeBERT, AIBugHunter and VulExplainer. The AIBugHunter outperforms other tested approaches in both the binary and multi-class classification tasks. However, it seems ChatGPT is not fine-tuned for given tasks, whereas the other tools are. What is more, the temperature values are not discussed, and the prompting strategies seem rather basic and not well-refined.

### **3.1.3 LLMs as a tool in vulnerability detection framework**

LLMs and other machine learning approaches can be leveraged in conjunction with existing static analysis tools. For example, LLMs have been shown to be able to enhance off-the-shelf taint analysis products [61]. Taint analysis is an approach where the data flows are observed to help discover vulnerabilities related to unsanitised data handling. The taint analysis product using a rule-based approach is set up to overestimate the sources, leading to many false positive results. After that, four different machine learning approaches, one of which is a LLM, are used. The machine learning step leverages natural language information, like method and variable names, to help verify whether the source is a true source. The results of the machine learning approaches are compared to baseline results leveraging frequency calculation and regular expressions. In many cases, all the proposed machine-learning approaches scored higher than the baseline. The authors are able to demonstrate the effectiveness of their approach by finding multiple previously unknown vulnerabilities in real-world projects. The authors discussed two main limitations. Firstly, the focus is on five high-severity vulnerability types. Secondly, the single-character variable names are excluded from the training and testing data, as they do not contain enough information for natural language processing. One of the four approaches uses an

off-the-shelf LLM – Codex. While Codex is among the approaches requiring the least effort to be set up and in some cases outperforms the baseline, overall, the Codex results are among the poorer ones. This highlights the need to compare different approaches, as more specific machine learning models can often perform better than general-purpose AI models. However, the more specific models also often require more effort to be set up. Another problem with the Codex model is the slow performance during the testing phase when compared to other approaches. A potential way to enhance the results, which was not explored by the authors, would be to rename the single-character variable names to more meaningful ones. LLMs such as Codex could help perform that step.

LLMs have shown promising results in automating fuzz testing for TensorFlow<sup>5</sup> and PyTorch<sup>6</sup> libraries [62]. Using a two-step approach, first LLMs generate seed programs and then those are utilised for evolutionary fuzzing. The resulting programs are then run on both CPU and GPU back-ends of deep learning systems and their outputs are compared. The focus is on significant differences in the outputs or a crash during execution. This shows success in identifying numerous previously unknown bugs in well-known deep-learning libraries. While the focus of the study is to identify software bugs in general, the methodology successfully uncovered several security vulnerabilities as well. The study claims to be the first to employ state-of-the-art LLMs for end-to-end fuzzing input generation, achieving full automation in the process. The findings demonstrate that LLMs can help perform both generation-based and mutation-based fuzzing, highlighting the potential of these models in software testing and vulnerability detection. The full automation of a fuzzing tool shows the potential of the black-box LLMs for automating complex code analysis processes. However, the given research only tests and reports on the results of using this approach for two Python deep-learning libraries. Though the approach can be extended to test systems in other domains, achieving similar results is improbable in a system lacking the ability to compare calculation results. This means for the use cases with a single back-end, only the system crashes could automatically be identified as bugs, reducing the effectiveness.

CodeAgent is an LLM-driven framework proposed for the detection of code inconsistencies, vulnerabilities and adherence to style standards [63]. CodeAgent divides tasks between different roles, where in every step two roles collaborate to achieve the task. The CodeAgent approach utilises multiple agents (LLMs), CoT prompting and a supervision agent (another LLM) that validates all the responses. For the vulnerability detection subtask, the CodeAgent is shown to outperform simple prompting strategies on well-established LLMs like CodeBERT, GPT-3.5 and GPT-4. However, the temperature values

---

<sup>5</sup><https://www.tensorflow.org/>

<sup>6</sup><https://pytorch.org/>

used for the LLMs are not provided. What is more, prompting strategies used with plain LLM models are not discussed. The cost aspect is not discussed in detail either, but using a large number of different LLMs with different roles is likely to be much more expensive than using a simple prompt on an LLM.

A vulnerability detection framework LLM4Vuln has been proposed, which is supposed to decouple and enhance the reasoning capabilities of LLMs for code analysis [64]. LLM4Vuln is divided into four components: knowledge retrieval, tool invocation, prompt schemes and instruction following. For knowledge retrieval, vector databases are used to provide up-to-date vulnerability knowledge and two different querying approaches are tested. Three external tools are provided: first for retrieving a function definition, second for retrieving all parent classes and third for retrieving a variable definition. They set the temperature for GPT-4 to one, which means different runs could produce different results. It is shown that adding vulnerability knowledge (vector database) improves LLMs' ability to reason about vulnerabilities, but necessitates careful design: short, summarised knowledge enhancement seems to provide the most benefits. The tools provided to the LLMs do not enhance the detection rate in most cases. The CoT prompting strategy is shown to improve precision in most cases, however, the improvements are not consistent. The authors do not discuss whether this could be related to the high-temperature values used. Other open-source LLMs are tested, which show inferior results. The inferior performance of those models is attributed to their lack of reasoning ability. The LLM4Vuln framework is tested on smart contracts, leading to the authors identifying nine zero-day vulnerabilities in two real-world projects.

Another approach called GPTScan is developed specifically for finding business logic vulnerabilities in smart contracts by combining the GPT language model with program analysis [65]. The idea is to use the LLM to understand the code and find potentially vulnerable functions. Static analysis is then used for validation and for reducing false positives. The GPT-3.5-turbo model with temperature set to zero is used over GPT-4 due to its lower costs. A "mimic-in-the-background" prompt is proposed, which asks the LLM to come up with multiple answers in the background and then respond with the most common answer. The effectiveness of the "mimic-in-the-background" prompt is not evaluated. The effectiveness of GPTScan is evaluated on three datasets comprising real-world smart contract projects. GPTScan shows good results in all datasets, with a low rate of false positives and a high F1 score, of up to 80%. The tool allowed the authors to uncover nine previously unknown vulnerabilities.

A framework called LLB is proposed for identifying and rectifying vulnerabilities in Android application source code [66]. The authors show GPT-4 model on the Ghera dataset

can detect and remediate vulnerabilities using prompt engineering and retrieval augmented generation (RAG) techniques. The sources for RAG are proposed to be additional code from the codebase, code analysis results or documentation and additional information. The Llama 2 model performance is stated to be significantly worse than that of GPT-4. Leaking hints like variables named "secure" are shown to have a noticeable impact on the results, highlighting the need for pre-processing of datasets used for experiments with LLMs. The given implementation of RAG performs worse than the simpler approach where only a summary of the flaws that are of interest are included. It is hypothesised the negative impact of using RAG could be caused by hallucinations, as for the RAG approach, file content summaries are generated by LLMs. The authors use the learnings from their experiments to develop a Python package LLB, allowing simple Android vulnerability scans with LLMs via the command line.

GPT language models have been used to detect improper isolation of compartmentalization (CWE-653) vulnerabilities in Angular applications [67]. The LLM is used as a black-box static analysis tool, with the temperature parameter set to zero for a deterministic approach. The dataset contains a random selection of 12 large public Angular projects. The approach consists of five steps: minifying the codebase, detecting sensitive elements, mapping project files, protection level discovery and aggregating the results. LLMs are used for the sensitive element detection, project file mapping and protection level discovery steps. Few-shot and CoT prompting approaches are used to achieve the best possible results. The results show that GPT-4 performs noticeably better than GPT-3.5 and even though GPT-4 is slower, the time difference is not big. More than 88% vulnerability detection rate is achieved, with no false positive results. The authors provide no exact numbers for precision, recall and F1 scores. As a limitation, the authors recognise when the evaluated project significantly diverges from the development conventions of the Angular framework, there is an increased likelihood that GPT-4 might misinterpret or struggle to detect contextual relationships.

### **3.1.4 LLMs with vulnerability detection as secondary focus**

The effectiveness of ChatGPT as a code analysis tool has been explored [68]. The study focuses on five code-related problems, which contain vulnerability analysis and dynamic vulnerability discovery. The authors find ChatGPT shows promise as a tool for security-oriented analysis of source code. The research discovers that the quality of the LLM responses heavily relies on good naming of the variables and functions in the source code. ChatGPT shows a very good ability to detect vulnerabilities for simple short code snippets following adequate naming conventions. ChatGPT is also able to help explain how the flaw could be exploited and offer fixes. However, not all suggestions made are correct. For



example, ChatGPT can explain how canary value could be used to detect stack overflow, however, the implementation offered is incorrect. Running the same experiments on obfuscated code yielded much worse results, showing LLMs heavily rely on well-defined names to produce good results.

The options for using ChatGPT in aiding developers to write secure software have been investigated [69]. ChatGPT is used for human-machine interactions in the context of code-writing and vulnerability-fixing tasks. Five different code-writing challenges are selected to conduct the experiment. The results show ChatGPT has not only the potential to be used as a teaching tool but also as a tool to help professional software developers. The authors discuss the use of LLMs to explain the issues identified by static analysis tools to help developers with any experience level. While they do not explicitly test LLMs in conjunction with static analysis tools, they show LLMs are somewhat capable of identifying and fixing vulnerabilities. The largest downside of their approach is only testing LLMs on five different challenges. As three of them were successful, a 60% success rate is reported, however, it could be argued the sample size is too small to draw meaningful conclusions.

The use of LLMs has been explored to assist with overall code reviews and to suggest best practices of infrastructure code [70]. The focus is on helping novices fix the bigger issues before involving the quality assurance department. While the paper does not explicitly focus on security, it does find the LLMs to be a viable option to help conduct infrastructure code reviews. However, this research is very limited: it only considers Kubernetes deployment manifest file analysis, the task is reduced to binary classification and the result validations are lacking. There is no comparison of the results with existing tools, even though existing tools are mentioned. What is more, there is no explanation why the few-shot learning approach is used and whether the zero-shot approach is considered or tested.

LLMs have been integrated to automate the guided bug replay process for Android applications [71]. The bug reports containing steps to reproduce can be difficult to follow due to ambiguity, and imprecision and can have steps missing. To simplify the job of developers ChatGPT is used to translate user-provided bug reports into step-by-step bug replay guides. Using few-shot learning in conjunction with CoT reasoning on bug reports, automated bug replay capabilities are achieved with more than 81% success rate. It is discussed that ChatGPT could be asked to also return the confidence it has in its solution to improve human and AI communication. This could also translate to vulnerability detection: if the GPT model can provide the confidence it has in its response, some false positives could potentially be disregarded.

Another area, for which LLMs have been proposed as a solution is the detection of malicious code in open-source packages. As open-source packages get widely used in the industry, a compromised or vulnerable open-source package can cause a lot of harm. A four-step approach leveraging LLMs to help detect vulnerable code has been proposed [72]. This approach consists of collecting code examples, turning the code examples into a vector representation, detecting anomalies and ranking the anomalies. The transformer-based language model is used in the second step: for transforming the code snippets to vector representation. The approach shows promising results for some scenarios, however, does not do well where the injected code is either quite long or has very abstract functionality. One of the limitations mentioned by the authors is that it is necessary to convert functions to abstract syntax tree (AST) representation. For future research, they suggest using more comprehensive representations. The researchers use the Code2Seq model but encourage future works to explore the use of different models, such as CodeBERT and Codex. The results show high detection precision on the given dataset, which includes real-world vulnerabilities. However, the setup and ease of use are not demonstrated and covered in enough detail. Using the newest GPT-4 in conjunction with existing tooling to detect malicious packages has shown good results [73].

### 3.2 Review summary

Studies have been conducted to evaluate the performance of LLMs as stand-alone vulnerability detectors and as tools in a multi-step approach. The datasets used to conduct the experiments vary: some use real-world datasets [57, 59, 67], others synthetic datasets [7, 54]. Some studies treat vulnerability detection as a binary classification challenge [7, 8, 10], and others as a multi-class classification challenge [53, 54, 57]. The used LLMs vary, with the GPT and Codex models being the most popular commercial models used [7, 8, 9, 10, 53, 54, 55, 56, 57, 59, 60, 61, 64, 65, 66, 67, 68, 71], with the Google Gemini model being included in some cases as well [58, 59]. BERT and Llama are the open-source models most often used for comparisons [7, 10, 54, 58, 66, 68]. Overall, the GPT-4 model has been shown to outperform the open-source and commercial models. The prompting approaches differ significantly, with simple zero-shot and CoT prompts being among the most popular ones. Most proposed CoT prompts just contain a few steps, like asking the LLM to first explain the code and then to find the vulnerabilities [8].

As the models, temperature values, datasets and prompts differ, the results vary significantly as well. Some have found that LLMs perform much worse than static analysers [57]. Others have shown LLMs can outperform static analysis tools [9]. Notably, in most cases, the LLMs seem to have higher false positive rates than static analysis tools. The main issues with the LLMs seem to be related to the costs of using the LLM models and the time it

takes to get a reply from the model. What is more, the models have an input character limit and the assumptions made by the language model can sometimes be more confusing than helpful [55]. The strengths of using LLMs to detect vulnerabilities are related to the ease of use and good true positive rate [9]. The LLMs can be used to detect vulnerabilities in any programming language, although the performance may vary.

LLMs have been proven helpful in extending the capabilities of existing tools and helping with the automation of static and dynamic analysis [61, 62]. The attempts to discard false positive results using different deep learning techniques also showed some success [61]. The results of the Codex LLM discarding false positive results were overshadowed, as other proposed approaches worked better in the given scenario [61]. However, it must be considered newer models might perform better in similar situations. The setup and ease of use did seem to favour the LLMs, as it did not require any data labelling efforts beforehand.

### **3.3 Identified research gaps**

Most research so far does not employ a systematic approach to evaluating the performance of the LLMs for vulnerability detection. The challenge is often formulated as a binary classification challenge and the prompting strategies are usually rather simplistic. The exact LLM versions used are often not mentioned, making it very difficult to reproduce the results. What is more, the temperature values are often not provided and even the studies that do provide the values, often fail to address why that value is chosen. Most studies use real-world datasets but provide the results of LLM without comparing the results to any traditional tools. The lack of a good comparison point makes it hard to understand how well the LLM performed.

Many studies focus only on the binary classification task without providing good justification for that decision. What is more, binary classification can be more difficult to verify, as the code might contain some previously undetected or incidentally introduced vulnerabilities. This problem is not addressed by some studies [8, 10], whereas others do address this problem by including the expected CWE identifier in the prompt and asking whether this vulnerability is present [7, 58]. The studies considering multi-class classification do not discuss using any matching strategies to be able to consider multiple CWE categories correct for some vulnerabilities [53].

There is some systematic testing of the effects the prompting strategy can have on the results [8]. However, while they provide interesting insights into prompting strategies, most of the tested prompts are still rather simple in nature. What is more, the existing

studies do not use any prompting strategies proposed in previous studies for comparison points. The costs related to different prompts are not discussed in detail.

There is a lot of focus on whether LLMs can effectively pinpoint vulnerabilities, often overshadowing the importance of proper comparative analysis. The attention is disproportionately directed towards true positives, while there is limited focus on false positives and false negatives [9, 53]. The precision, recall and F-scores are sometimes omitted, requiring more effort from the reader to understand how good the proposed approach is. What is more, the results are often not compared to existing tools. This is important because the dataset might contain problems that are difficult to detect and even if the LLM performs poorly, it might still outperform existing tools in some cases.

## 4. Methods

### 4.1 LLMs and prompting strategies

To conduct static application security testing (SAST) with LLMs, we use three models: GPT-4 and GPT-4 turbo models from OpenAI<sup>1</sup> and Claude 3 Opus model from Anthropic<sup>2</sup>. The exact versions for the LLMs and other relevant tools are given in Appendix 4 – Versions. For most experiments, the temperature parameter of the LLM is set to zero. The temperature parameter is used to control the randomness of the output and using a zero value makes the output as deterministic as possible [35]. This means anyone running the same experiments with the same settings should get very similar results to us.

LLMs require a description of the task they are expected to perform and these descriptions are called prompts. It has induced a separate field of study to find the best prompting approaches for different types of tasks. We use different zero-shot and few-shot prompts and test different strategies, where we request the LLMs to review their responses. We test CoT, ToT and self-consistency strategies and report on the results. We conduct quantitative analysis of the tested prompts' results and provide qualitative analysis of the best results.

### 4.2 Dataset

The full Java Juliet 1.3 dataset contains vulnerabilities from 112 different CWEs. Among others, the dataset covers categories like CWE-546, which refers to suspicious comments. To focus on high-severity issues, only vulnerabilities from the MITRE top 25 [74] are chosen, similarly to previous research [7]. It must be noted the CWE list is very detailed and many weaknesses in that list are very similar or closely related. To help understand how different CWEs are related, MITRE provides multiple views, which show the relationships between CWEs. The CWE-1000 (Research Concepts) view provided by MITRE displays the relationships in a hierarchy, where every CWE can have one parent and multiple children [75]. An example of the relationships for CWE-36 (absolute path traversal weakness) can be seen in Figure 1. The parent of CWE-36 is CWE-22 and the children of CWE-36 are CWE-37, CWE-38, CWE-39 and CWE-40. As the Java Juliet 1.3 dataset only contains four CWEs from the 2023 MITRE top 25 list [74], we also include the subcategories (the children) of the MITRE top 25 vulnerabilities. For example, the CWE-

---

<sup>1</sup><https://openai.com/>

<sup>2</sup><https://www.anthropic.com/>

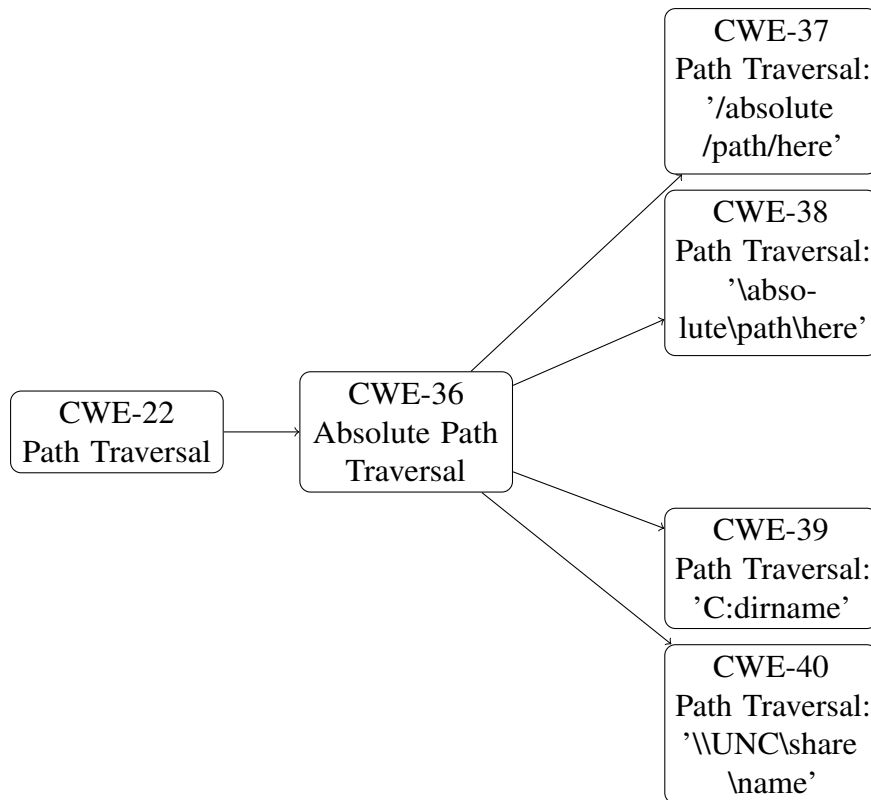


Figure 1. An image of the CWE-36 hierarchy.

22 (path traversal weakness) is in the MITRE top 25 list but not in the Juliet dataset [74]. However, the Juliet dataset contains the subcategories of CWE-22, namely CWE-23 (relative path traversal weakness) and CWE-36 (absolute path traversal weakness), which we include. Table 1 displays all the CWEs present in the dataset in bold format. Using this strategy allows us to extend our dataset from the four exact matches to 17 distinct CWEs.

Table 1. CWE's overlap in MITRE top 25 and Juliet 1.3

Rank	ID	Present in Juliet 1.3	Has subcategories in Juliet 1.3
1	CWE-787	No	No
2	CWE-79	No	<b>CWE-80, CWE-81, CWE-83</b>
3	<b>CWE-89</b>	Yes	No
4	CWE-416	No	No
5	<b>CWE-78</b>	Yes	No
6	CWE-20	No	<b>CWE-129, CWE-606</b>
7	CWE-125	No	No
8	CWE-22	No	<b>CWE-23, CWE-36</b>
9	CWE-352	No	No
10	CWE-434	No	No
11	CWE-862	No	No

12	<b>CWE-476</b>	Yes	No
13	CWE-287	No	<b>CWE-259, CWE-321, CWE-256, CWE-523, CWE-549</b>
14	<b>CWE-190</b>	Yes	No
15	CWE-502	No	No
16	CWE-77	No	<b>CWE-78</b>
17	CWE-119	No	No
18	CWE-798	No	<b>CWE-259, CWE-321</b>
19	CWE-918	No	No
20	CWE-306	No	No
21	CWE-362	No	No
22	CWE-269	No	No
23	CWE-94	No	No
24	CWE-863	No	<b>CWE-566</b>
25	CWE-276	No	No

#### 4.2.1 Dataset pre-processing

The Java code does not need to be compiled to be analysed by LLMs. However, to run the traditional static analysis tools, we need the code to be compiled. To simplify the compiling of large projects, multiple build tools are available. The default distribution of the Java Juliet 1.3 dataset is using Ant<sup>3</sup> as the build tool, which we upgrade to Gradle<sup>4</sup>. As many traditional SAST tools offer Gradle plugins, this upgrade simplifies the building process and the running of different tools. The code reformatting tools included in IntelliJ<sup>5</sup> are used to reformat the files, which helps to ensure all files use a similar format.

Prior research has shown leaking some relevant keywords in the code, like variables named "secure", could influence the output of the LLMs [66]. To avoid introducing this bias, these types of hints are removed from the dataset. The original dataset contains comments explaining the vulnerabilities, so all comments are removed. As the file and class name both include the vulnerability identifier, all the files and classes are re-named. All the function and variable names containing any hints (like "good" or "bad") are also changed. The package name is not changed to simplify the analysis of SAST tool results. However, the package name is always overwritten when working with LLMs, before the file is sent to the LLM for analysis.

<sup>3</sup><https://ant.apache.org/>

<sup>4</sup><https://gradle.org/>

<sup>5</sup><https://www.jetbrains.com/idea/>

To give as much context to the LLMs for vulnerability detection as possible, we test the detection capabilities on the file level. By default, the Juliet dataset is meant for running the vulnerability classification on either function or line level. Similarly to previous research, the non-standard test cases spanning multiple files or only containing vulnerable examples are removed [7]. The remaining samples are split into two: a good and a bad file. This way, all the context needed to detect a vulnerability is contained inside a single file.

After removing the test cases spanning multiple files and splitting the remaining files into two, we are left with 15,174 files, half of them secure and the other half vulnerable. Due to the high cost of running LLMs, we are unable to do the experiments with all of the files. Thus, as the last step, a random subset of the files is selected. As the CWE-523 and CWE-549 categories are the smallest containing just 34 files each, random 34 files are selected for all 17 CWE categories represented in the dataset. This way we end up having the same number of samples for every CWE in our dataset. The random selection is done in the following manner. Firstly, we pick a random file from our dataset. Secondly, if the file is vulnerable, we find the non-vulnerable version, and if the file is not vulnerable, we find the vulnerable version. We repeat this process 17 times for each CWE, adding both files to the subset. As there are 17 unique CWEs in the dataset, together  $(17 + 17) \times 17 = 578$  files are chosen. Half of the 578 files are vulnerable and the other half are not.

The full pre-processed dataset is available in version control software [76]. The custom scripts used for the pre-processing are available in version control software in the "dataset-normalization" package [77].

### 4.3 Evaluation methods

The authors of the Juliet dataset publicly acknowledge the code provided in the dataset might include other unrelated vulnerabilities [78]. Thus, for vulnerable files, the result is considered true positive only when the targeted vulnerability is found in the file. If no vulnerabilities are found or if the found vulnerabilities do not include the targeted vulnerability, the result is classified as a false negative. For non-vulnerable files, the result is considered true negative if the targeted vulnerability is not discovered in the file. If the targeted vulnerability is found, the result is classified as a false positive.

We look at vulnerability detection as a multi-class classification challenge. The analysis is expected to report not only whether the file is vulnerable, but also to correctly detect the CWE identifier of the vulnerability. The Juliet dataset is labelled, which allows us to classify the results as true positive, false positive, true negative or false negative, as shown in Table 2. If the expected vulnerability is detected, then the classification is positive.



Table 2. *Classification logic*

	<b>Expected vulnerability detected</b>	<b>Expected vulnerability not detected</b>
<b>Vulnerability present</b>	True Positive (TP)	False Negative (FN)
<b>Vulnerability not present</b>	False Positive (FP)	True Negative (TN)

Using those measures, we can compare the performance of different tools by calculating accuracy, precision, recall and F1 scores. The F1 score is also called the harmonic mean of recall and precision. The formulas to calculate these values are as follows [79]:

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

$$Precision = \frac{TP}{TP + FP}$$

$$Recall = \frac{TP}{TP + FN}$$

$$F1 = 2 \times \frac{precision \times recall}{precision + recall} = \frac{2 \times TP}{(2 \times TP) + FP + FN}$$

As shown in Figure 1, many CWEs point to very similar flaws. The "Research Concepts" view provided by MITRE gives a good overview of the relationships between vulnerabilities [75]. For example, the CWE-36 (Absolute Path Traversal) is present in our dataset. The parent and the children of CWE-36 all point to different variations of absolute path traversal weakness. To fairly assess the results, we employ a strategy similar to what has been used for evaluating SAST tools previously [4, 80]. This strategy allows the parent CWE and the child CWEs to also be considered to be a correct classification based on the MITRE "Research Concepts" view. The only exception we make is related to the highest level CWEs in the "Research Concepts" view, which are called pillars. If the parent of the CWE is of type pillar, we do not count that as the correct classification, as the pillar descriptions can be very broad. For example, the parent of CWE-476 (NULL Pointer Dereference) is CWE-710 (Improper Adherence to Coding Standards), which contains many different vulnerabilities. The mapping we use is shown in Table 3. Interestingly, even though this strategy has been employed in static code analyser benchmarks [4, 80], it has not been previously utilised in LLM benchmarking studies [7, 8, 9, 10]. This is most likely related to either a lack of awareness or the complexity it adds to the evaluation process.

Table 3. *Mapping strategy*

<b>Original CWE</b>	<b>Acceptable CWEs</b>
CWE-23	CWE-22, CWE-23, CWE-24, CWE-25, CWE-26, CWE-27, CWE-28, CWE-29, CWE-30, CWE-31, CWE-32, CWE-33, CWE-34
CWE-36	CWE-22, CWE-36, CWE-37, CWE-38, CWE-39, CWE-40
CWE-78	CWE-77, CWE-78
CWE-80	CWE-79, CWE-80
CWE-81	CWE-79, CWE-81
CWE-83	CWE-79, CWE-82, CWE-83
CWE-89	CWE-89, CWE-564, CWE-943
CWE-129	CWE-129, CWE-1285
CWE-190	CWE-190, CWE-680
CWE-256	CWE-256, CWE-522
CWE-259	CWE-259, CWE-798
CWE-321	CWE-321, CWE-798
CWE-476	CWE-476
CWE-523	CWE-522, CWE-523
CWE-549	CWE-522, CWE-549
CWE-566	CWE-566, CWE-639
CWE-606	CWE-606, CWE-1284

## 5. Results

### 5.1 Static code analysis tools

Previous benchmarks of static code analysis tools have shown YASCA<sup>1</sup> and FindBug<sup>2</sup> to perform well on the Juliet dataset [16]. While both tools are no longer supported and have been deprecated, FindBug has a community-developed successor project called SpotBugs<sup>3</sup>. SpotBugs can be used with plugins originally meant for FindBug. SpotBugs is actively maintained and improved by the community. Other previous studies have used CodeQL as the comparison point when evaluating LLMs [7]. Both CodeQL and SpotBugs have shown strong results on synthetic datasets, making them a good comparison point for this study [81]. Like most static code analysis tools, both CodeQL and SpotBugs need the dataset to be compileable to run the scan.

#### 5.1.1 CodeQL

CodeQL is a tool that treats code like data and allows anyone to write queries against the code [82]. The software is written and maintained by GitHub and the community, with the queries being open-source [82]. CodeQL supports ten different programming languages [83]. Out of the box, CodeQL provides three different configurations for Java code analysis: the default configuration, the extended security configuration, and the extended security and quality configuration. CodeQL provides mappings for CWE-IDs, which significantly simplifies analysing the results [84]. All three different configurations are tested, with the results displayed in Table 4. The default configuration is denoted as CodeQL-d, the extended security configuration is denoted as CodeQL-es and the extended security and quality configuration is denoted as CodeQL-esq.

Table 4. *CodeQL results*

	<b>TP</b>	<b>FP</b>	<b>TN</b>	<b>FN</b>	<b>Accuracy</b>	<b>Precision</b>	<b>Recall</b>	<b>F1</b>
CodeQL-d	76	5	284	213	0.623	<b>0.938</b>	0.263	0.411
CodeQL-es	127	20	269	162	0.685	0.864	0.439	0.583
CodeQL-esq	137	23	266	152	<b>0.697</b>	0.856	<b>0.474</b>	<b>0.61</b>

<sup>1</sup><https://github.com/scovetta/yasca>

<sup>2</sup><https://findbugs.sourceforge.net/>

<sup>3</sup><https://github.com/spotbugs/spotbugs>

The default configuration seems to be configured to produce as good precision as possible. This makes sense, as many false positive results could get taxing to deal with for the developers. However, the other configurations perform better in terms of accuracy, recall and F1 values. The extended security and quality configuration produces the best results, achieving an F1 score of 0.61. Unfortunately, the previous study benchmarking LLM vulnerability detection capabilities against CodeQL does not disclose the exact tested or used configurations [7].

We also provide the results per CWE for the most successful approach, the CodeQL-esq run, in Table 5. From the 17 CWEs in our dataset, CodeQL is unable to detect any vulnerabilities for six CWEs. From these six, two CWEs, namely CWE-566 and CWE-606, are not supported by the CodeQL mapping strategy [84]. This means CodeQL had no mapping strategy to detect the CWE itself nor any of the parent or child CWEs.

Table 5. *CodeQL-esq results per CWE*

<b>CWE</b>	<b>TP</b>	<b>FP</b>	<b>TN</b>	<b>FN</b>	<b>Accuracy</b>	<b>Precision</b>	<b>Recall</b>	<b>F1</b>
CWE-23	11	0	17	6	0.824	1	0.647	0.786
CWE-36	8	0	17	9	0.735	1	0.471	0.64
CWE-78	17	5	12	0	0.853	0.773	1	0.872
CWE-80	14	0	17	3	0.912	1	0.824	0.903
CWE-81	0	0	17	17	0.5	0	0	0
CWE-83	14	0	17	3	0.912	1	0.824	0.903
CWE-89	17	2	15	0	0.941	0.895	1	0.944
CWE-129	5	0	17	12	0.647	1	0.294	0.455
CWE-190	7	0	17	10	0.706	1	0.412	0.583
CWE-256	0	0	17	17	0.5	0	0	0
CWE-259	17	13	4	0	0.618	0.567	1	0.723
CWE-321	17	0	17	0	1	1	1	1
CWE-476	10	3	14	7	0.706	0.769	0.588	0.667
CWE-523	0	0	17	17	0.5	0	0	0
CWE-549	0	0	17	17	0.5	0	0	0
CWE-566	0	0	17	17	0.5	0	0	0
CWE-606	0	0	17	17	0.5	0	0	0

### 5.1.2 SpotBugs

SpotBugs is a tool for finding bugs in Java code [85]. SpotBugs supports plugins, from which the Find Security Bugs<sup>4</sup> plugin is used, similarly to prior benchmarks [81]. The results of running the analysis are summarised in Table 6. The default configuration is denoted as SpotBugs-d and the configuration with the Find Security Bugs plugin is denoted as SpotBugs-fsb. The documentation only provides some CWE mappings for the vulnerabilities detected by the Find Security Bugs plugin, the rest required manually mapping the results to CWE identifiers.

Table 6. *SpotBugs results*

	<b>TP</b>	<b>FP</b>	<b>TN</b>	<b>FN</b>	<b>Accuracy</b>	<b>Precision</b>	<b>Recall</b>	<b>F1</b>
SpotBugs-d	39	20	269	250	0.533	0.661	0.135	0.224
SpotBugs-fsb	152	23	266	137	<b>0.723</b>	<b>0.869</b>	<b>0.526</b>	<b>0.655</b>

The use of Find Security Bugs significantly improves the vulnerability detection capabilities of the SpotBugs tool. The SpotBugs-fsb configuration performs better than the default configuration for accuracy, precision, recall and F1 scores. It also performs better than the best CodeQL approach, showing higher scores in most measurements. The only measure where CodeQL outperforms SpotBugs is precision when the CodeQL default configuration is used.

We also provide the results per CWE for the most successful approach, the SpotBugs-fsb run, in Table 7. From the 17 CWEs in our dataset, SpotBugs is unable to detect any vulnerabilities for six CWEs. Five of these are the same CWE categories that CodeQL is unable to detect. Interestingly, where CodeQL was unable to detect any vulnerabilities for the CWE-81, SpotBugs achieved perfect results with that CWE category. However, CodeQL was able to correctly identify seven out of the 17 CWE-190 vulnerabilities, whereas SpotBugs found none.

Table 7. *SpotBugs-fsb results per CWE*

<b>CWE</b>	<b>TP</b>	<b>FP</b>	<b>TN</b>	<b>FN</b>	<b>Accuracy</b>	<b>Precision</b>	<b>Recall</b>	<b>F1</b>
CWE-23	13	0	17	4	0.882	1	0.765	0.867
CWE-36	15	0	17	2	0.941	1	0.882	0.938
CWE-78	16	2	15	1	0.912	0.889	0.941	0.914
CWE-80	17	0	17	0	1	1	1	1

<sup>4</sup><https://find-sec-bugs.github.io/>

CWE-81	17	0	17	0	1	1	1	1
CWE-83	17	1	16	0	0.971	0.944	1	0.971
CWE-89	17	17	0	0	0.5	0.5	1	0.667
CWE-129	4	3	14	13	0.529	0.571	0.235	0.333
CWE-190	0	0	17	17	0.5	0	0	0
CWE-256	0	0	17	17	0.5	0	0	0
CWE-259	14	0	17	3	0.912	1	0.824	0.903
CWE-321	12	0	17	5	0.853	1	0.706	0.828
CWE-476	10	0	17	7	0.794	1	0.588	0.741
CWE-523	0	0	17	17	0.5	0	0	0
CWE-549	0	0	17	17	0.5	0	0	0
CWE-566	0	0	17	17	0.5	0	0	0
CWE-606	0	0	17	17	0.5	0	0	0

## 5.2 Vulnerability detection with LLMs

For the experiments, we use the GPT-4 turbo model with the temperature set to 0 unless explicitly stated otherwise. There are two reasons for opting to use the GPT-4 turbo model for the majority of the tests. Firstly, the GPT-4 turbo model costs per token are significantly cheaper than those of the GPT-4 or Claude 3 Opus. Secondly, the GPT-4 turbo has a larger context window. Similarly to prior studies, the temperature parameter of the LLM is set to zero where not stated otherwise [7, 57, 67]. The temperature parameter is used to control the randomness of the output and using a zero value makes the output as deterministic as possible [35]. This means anyone running the same experiments with the same settings should get very similar results to us.

It is important to note the LLMs are accessed through an API and used as a black box. OpenAI and Anthropic APIs are used in conjunction with the LangChain Python library [86] to conduct the experiments. Both OpenAI and Anthropic use a token-based pricing structure, which means they request money for every input and output token. For the models used, output tokens cost between two to five times more than input tokens. This provides an incentive to give as much context as possible to the model as input. It also allows to cut costs by explicitly stating the format and length of the expected output. The result tables contain cost and time columns, which are aggregated values over the whole dataset (578 files). The cost is in dollars and excludes VAT. To save on the costs, we remove the indentation from the Java files before sending them to the LLM. The time is marked in hours and it must be noted the time it takes to run the same prompt through the

same LLM seems to vary notably. What is more, both OpenAI and Anthropic assign tiers based on the amount of money paid to them and the length of time one has been a user of the API [87, 88]. On lower tiers, one can run into rate-limiting issues, which can cause variance in the running time [87, 88]. Another cause could be related to the amount of simultaneous requests received by the service provider. Thus the time noted here is not a reliable measure but more of an indication to show how much time the analysis took in our settings.

All the full prompts are available in Appendix 2 – Prompts. To better display the tables with results, all prompts are given shortened names which are referenced in the text, tables and appendix. All the scripts used for testing and analysing the results are available in version control software under the "llm-analyze", "scat-analyze" and "statistics" packages [77].

### 5.2.1 Establishing baseline prompting strategy

To first establish a baseline, a basic prompt, which we denote as  $p_b$ , is compiled using the best practices suggested by OpenAI for prompt engineering [39]. The LLM is asked to adopt the persona of a security researcher, the instructions are clear on what is expected and what the output should be. First, the same basic prompt is used three times to see how big the variances are with LLM set to its most deterministic mode (temperature set to 0). As can be seen in Table 8, the results for using the same prompt produce very similar outcomes.

There have been suggestions to ask LLMs to re-evaluate their responses to achieve better results and different approaches for this have been proposed [7, 39, 40, 41]. Four different approaches are tested, where the output of the basic prompt is given back to the LLM and the LLM is asked to re-evaluate and improve its previous response. All the approaches are quite similar but they focus on slightly different aspects. The first approach is called recursive criticism and improvement (RCI), which we denote as  $p_{b-rci}$  [40]. RCI builds on the initial prompt and answer from the LLM, asking the LLM to find problems with its initial answer and then improve it based on the problems. The second improvement tactic is called self-refinement, denoted as  $p_{b-sr}$  [41]. Self-refinement asks to provide overall feedback (instead of just focusing on problems) on the previous answer and to improve the initial response. For the third approach, we asked the LLM to provide feedback and response in one go and called it short self-refinement ( $p_{b-ssr}$ ). The short self-refinement has the advantage of being faster and cheaper, as the LLM is invoked only twice and fewer tokens are used. This approach is the most similar to what has been utilised by previous research on the topic [7]. As can be seen in Table 8, the RCI strategy performed the best.

To test if we could achieve similar results with cheaper costs, lastly, a strategy we called short RCI ( $p_{b-srcci}$ ), is tried. The idea behind short RCI is to try and find out how the criticism and improvements in one step perform in comparison to the original.

Table 8. *Baseline prompting results*

Strategy	TP	FP	TN	FN	Accuracy	Precision	Recall	F1	Cost	Time
$p_b$	134	131	158	155	0.505	0.506	0.464	0.484	4.38\$	0.9h
$p_b$	133	130	159	156	0.505	0.506	0.46	0.482	4.39\$	1h
$p_b$	130	133	156	159	0.495	0.494	0.45	0.471	4.39\$	0.9h
$p_{b-rci}$	140	49	240	149	<b>0.657</b>	<b>0.741</b>	<b>0.484</b>	<b>0.586</b>	17.47\$	4.2h
$p_{b-sr}$	135	91	198	154	0.576	0.597	0.467	0.524	22\$	6.8h
$p_{b-ssr}$	119	70	219	170	0.585	0.63	0.412	0.498	9.72\$	1.9h
$p_{b-srcci}$	133	80	209	156	0.592	0.624	0.46	0.53	11.07\$	2.6h
$p_{b-sc}$	133	133	156	156	0.5	0.5	0.46	0.479	13.16\$	2.7h

As seen in Table 8, the short RCI tactic ( $p_{b-srcci}$ ) shows second-best results. While the short RCI tactic does provide significant time and cost savings in comparison to the full RCI, it performs more poorly in accuracy, precision, recall and F1 scores. We hypothesise the difference in results is related to the RCI prompt allowing the LLM to generate relevant clues, and build the correction on those clues. Asking the LLM to do both in the same step will give the LLM less input for the final verdict, which seems to affect the results negatively.

Lastly, a prompting strategy called self-consistency is tested, which we denote as  $p_{b-sc}$  [19]. For this, the basic prompt is run three times and only the files that are classified positive two or three times are counted as positive. This performs worse than the RCI, self-refinement and self-reflection strategies, most likely due to using a temperature value of zero. With higher temperature values, self-consistency could provide more benefits, as the higher temperature would cause the output to have more randomness, which the self-consistency strategy could help control.

## 5.2.2 Comparing prompting approaches from prior studies

While different prompting strategies have been proposed by prior studies, they have not been compared amongst each other on the same dataset. We try different approaches that have shown good results in previous studies and report on the results to show which one performs the best on our dataset. The results are displayed in Table 9, where we have the basic prompt and the basic prompt utilising RCI as the comparison point.



Table 9. Prompts from prior studies results

Strategy	TP	FP	TN	FN	Accuracy	Precision	Recall	F1	Cost	Time
$p_b$	134	131	158	155	0.505	0.506	0.464	0.484	4.38\$	0.9h
$p_{b-rci}$	140	49	240	149	0.657	0.741	0.484	0.586	17.47\$	4.2h
$p_{as}$	146	123	166	143	0.54	0.543	0.505	0.523	5.11\$	0.7h
$p_{as-rci}$	131	59	230	158	0.625	0.689	0.453	0.547	18.48\$	2.4h
$p_{rf}$	161	168	121	128	0.488	0.489	0.557	0.521	11.86\$	6.2h
$p_{rf-rci}$	153	112	177	136	0.571	0.577	0.529	0.552	41.15\$	18h
$p_{fs20}$	147	150	139	142	0.495	0.495	0.509	0.502	17.39\$	0.4h
$p_{fs6}$	157	132	157	132	0.543	0.543	0.543	0.543	6.07\$	0.5h
$p_{fs6-rci}$	168	146	143	121	0.538	0.535	0.581	0.557	34.95\$	6.4h
$p_{dfa}$	150	102	187	139	0.583	0.595	0.519	0.555	9.33\$	4.8h
$p_{dfa-rci}$	171	57	232	118	<b>0.697</b>	<b>0.75</b>	<b>0.592</b>	<b>0.662</b>	34.58\$	20.5h

LLMs can have difficulties extracting the API call sequences from code [8]. Adding the API call sequence to the prompt along with the code has been shown to improve the LLM vulnerability detection capabilities [8]. To test that, API call sequence extraction capabilities are created<sup>5</sup> and the API sequence is provided to the LLM along with the code. This prompt is denoted as  $p_{as}$  and it does outperform the basic prompting strategy. We also test the RCI approach on the results ( $p_{as-rci}$ ) and while it does significantly lower the amount of false positive results, it also lowers the amount of true positive results. The RCI strategy does have a positive effect on the results, raising the accuracy, precision and F1 scores. Nonetheless, the basic prompting strategy benefits more from the RCI strategy and thus this approach does not outperform the baseline basic prompt utilising the RCI strategy.

It has been hypothesised, that LLMs might be able to perform better if they are not only asked to find vulnerabilities, but to also provide a fix for them [9]. To test that, we use a prompt that requires the LLM to detect vulnerabilities and to provide a fix for the found issue, denoted as  $p_{rf}$  for require fix prompting. The same prompt is tested utilising the RCI strategy as well, which we denote as  $p_{rf-rci}$ . Similarly to the API call sequence prompt, the requiring fix approach outperforms the basic prompt but sees very little improvement with the RCI strategy. Overall, this approach does not outperform the baseline basic prompt utilising the RCI strategy. What is more, the require fix prompting strategy ends up being among the most costly strategies we test. This is due to the fact that the output tokens are expensive and the output is expected to contain the fixed code.

GPT-4 has shown good results in vulnerability detection tasks in few-shot settings. There

<sup>5</sup><https://github.com/katamb/thesis-scripts/blob/main/llm-analyze/tools/ast/ApiCallExtractor.py>

are many potential ways to do few-shot prompting: using different examples, having different amounts of examples and changing the proportion of vulnerable or non-vulnerable samples. We follow the previous study, utilising examples provided by MITRE and including examples from the top 25 vulnerabilities [10]. We have an even amount of vulnerable and non-vulnerable samples and we try with different sample sizes. First, we include all CWEs from the MITRE top 25 where they have a Java code example available. Overall, there are ten such CWEs and as we also include a fixed version of all the samples, we end up with 20 examples overall, thus we name this prompt  $p_{fs20}$ . Of the ten included CWEs, only two exactly match the CWEs in our dataset. To see how providing less examples affects the results, we also try a prompt containing six simple samples overall, three vulnerable and three non-vulnerable, which we call  $p_{fs6}$ . There we use one CWE, which is also present in our dataset and two that are not. This approach yields better results, most likely due to having less distracting code samples in the context. We are able to slightly improve the F1 score with RCI strategy ( $p_{fs6-rci}$ ), however, this not only raises the number of true positive but also the false positive results.

Asking LLMs to analyse the dataflow has been shown to improve the vulnerability detection results [7]. This is similar to CoT prompting, as the LLM is asked to analyse the dataflows and sanitisers, but without explicitly stating to think step-by-step. To test that approach, we run a dataflow analysis prompt, denoted as  $p_{dfa}$ , similarly to [7]. While the given paper uses a self-refinement strategy to improve the results [7], we use the RCI approach ( $p_{dfa-rci}$ ), as we saw it perform better on the baseline results. The main difference between our approach and that of the original paper [7] is that they used LLMs for binary classification, whereas we use them for multi-class classification. The dataflow analysis prompt with RCI stands out as the best-performing approach. It outperforms the previous best approaches for the accuracy, precision, recall and F1 scores. What is more, this approach also outperforms the best CodeQL and SpotBugs results. While it is not the costliest strategy we test, it must be noted the main downside of this strategy is that it is rather costly to run.

### 5.2.3 Custom prompting strategies

Testing the approaches proposed in previous studies allowed us to get good results. To see if these results could be further improved, multiple additional approaches are tested. To provide a better overview, the best results from the baseline testing and the previous studies approaches have been provided in Table 10 among the new results.

Table 10. *Custom prompting results*

Strategy	TP	FP	TN	FN	Accuracy	Precision	Recall	F1	Cost	Time
$p_b$	134	131	158	155	0.505	0.506	0.464	0.484	4.38\$	0.9h

$p_{b-rci}$	140	49	240	149	0.657	0.741	0.484	0.586	17.47\$	4.2h
$p_{dfa}$	150	102	187	139	0.583	0.595	0.519	0.555	9.33\$	4.8h
$p_{dfa-rci}$	171	57	232	118	<b>0.697</b>	<b>0.75</b>	<b>0.592</b>	<b>0.662</b>	34.58\$	20.5h
$p_{dfa-h}$	163	110	179	126	0.592	0.597	0.564	0.58	8.64\$	4.7h
$p_{dfa-h-rci}$	165	61	228	124	0.68	0.73	0.571	0.641	35.47\$	39.5h
$p_{cot-dfa}$	142	72	217	147	0.621	0.664	0.491	0.565	12.76\$	4.5h
$p_{cot-dfa-rci}$	146	65	224	143	0.64	0.692	0.505	0.584	43.61\$	11.2h
$p_{cot-dfa-srci}$	140	63	226	149	0.633	0.69	0.484	0.569	34.12\$	7.6h
$p_{cot-8s}$	160	88	201	129	0.625	0.645	0.554	0.596	13.4\$	4.9h
$p_{cot-8s-rci}$	161	85	204	128	0.631	0.654	0.557	0.602	45.94\$	11.7h
$p_{cr}$	144	116	173	145	0.548	0.554	0.498	0.525	12.49\$	5.3h
$p_{cr-rci}$	142	110	179	147	0.555	0.563	0.491	0.525	43.83\$	12h

As the dataflow analysis prompt with RCI ( $p_{dfa-rci}$ ) showed the best results, modifications of the prompt are tested to try and improve the results. First, some more hints are added to the prompt, which we denote as  $p_{dfa-h}$ . This is a small change, where the name of the programming language (Java) is added and the wording is slightly changed. The wording is changed to explicitly mention the file might not contain any vulnerabilities at all. This did show slight improvements over the original prompt ( $p_{dfa}$ ). However, testing this approach with RCI strategy ( $p_{dfa-h-rci}$ ) did not bring as big improvements as the original. While we managed to achieve an F1 score similar to the one of the original dataflow analysis with RCI, we are unable to improve the results.

To see if asking to think step-by-step offers any improvements, the dataflow analysis prompt is modified into a CoT prompt, denoted as  $p_{cot-dfa}$ . Just as with adding the hints, we can get some improvements before the RCI strategy is applied. However, the RCI strategy ( $p_{cot-dfa-rci}$ ) does not offer as many improvements as it did for the original dataflow analysis prompt. As this ended up being the most expensive approach, we tried to see if we could lower costs without impacting the performance. To test this, experiments are conducted using the dataflow analysis CoT prompt with RCI methodology. However, the LLM is asked to summarise the problems in five sentences or less to save on the output tokens. This prompt is denoted  $p_{cot-dfa-srci}$ . While this approach is indeed somewhat cheaper and faster, it performed worse than the same approach without limitations.

A separate CoT strategy is developed to more closely mimic what software engineers would manually check during the code review process. This consists of eight steps, thus the prompt is named CoT eight-step, denoted as  $p_{cot-8s}$ . For previous CoT approaches, we have seen very small improvements with the RCI strategy. In this case, the RCI once again has very small impact on the F1 score. This is likely due to the CoT approach already

enriching the context enough for the LLM, which means the RCI strategy does not add anything more meaningful. However, the CoT eight-step is able to produce fourth-best results overall with far smaller costs than those of the better-performing strategies.

Treating the exercise as code review seemed to perform rather well. Motivated by that observation, another prompt for a similar approach is developed. This time the LLM is given a checklist of questions and asked to treat this as a code review exercise, denoted as  $p_{cr}$  for the code review prompt. These questions contain quite generic questions that software engineers should think about when reviewing the code. The checklist is taken from the internet and slightly supplemented [89]. This approach performs much poorer than the CoT 8-step approach in accuracy, precision, recall and F1 scores. Interestingly, this is the only approach which did not benefit from using the RCI strategy ( $p_{cr-rci}$ ), showing the same F1 score both before and after using the RCI strategy.

## 5.2.4 Strategies requiring higher temperature values

Some prompting strategies benefit from using higher temperature values. We see that with the temperature value set to zero, the self-consistency approach does not provide many benefits. That makes sense, as the idea of self-consistency is to be able to get consistent responses with higher temperatures [19]. Another strategy that should benefit from higher temperature values is the tree of thoughts (ToT) [18]. ToT strategy is somewhat similar to the CoT, however, for every step multiple potential responses are generated. These responses are then evaluated and the best one is chosen. As both the self-consistency and ToT have shown good results with temperature values of 0.7, that is the temperature we use [18, 19].

To test these methods, we use the CoT 8-step prompt, as it provides the best results before applying any further steps (like the RCI strategy). As running these strategies is expensive, we start with testing a few select CWEs to better understand if they could outperform the previous attempts. Firstly, the CWE-23 is chosen, which has a very good detection rate with a plain CoT 8-step prompt. Secondly, CWE-129 is chosen, which has an average detection rate when using a plain CoT 8-step prompt. Thirdly, CWE-549 is chosen, which is not detected at all by the basic CoT 8-step strategy.

Table 11. Results from initial testing of higher-temperature strategies

Strategy	CWE	TP	FP	TN	FN	Accuracy	Precision	Recall	F1	Cost	Time
$p_{cot-8s}$ $t = 0$	CWE-23	17	2	15	0	0.941	0.895	1	0.944	2.38\$	0.8h
	CWE-129	14	15	2	3	0.471	0.483	0.824	0.609		
	CWE-549	0	0	17	17	0.5	0	0	0		

$p_{cot-8s-sc}$ $t = 0.7$	CWE-23	17	1	16	0	0.971	0.944	1	0.971	7.31\$	2.5h
	CWE-129	14	14	3	3	0.5	0.5	0.824	0.622		
	CWE-549	0	0	17	17	0.5	0	0	0		
$p_{tot-8s}$ $t = 0.7$	CWE-23	17	11	6	0	0.676	0.607	1	0.756	49.08\$	5.9h
	CWE-129	11	5	12	6	0.676	0.688	0.647	0.667		
	CWE-549	0	0	17	17	0.5	0	0	0		

For the ToT strategy, we generate three potential candidates for each step and then have three different evaluators evaluate the responses for each step. The response that gets chosen by the most evaluators is picked. This is similar to the approach the authors of the ToT paper utilised in the creative writing task [18]. For the testing, we forked the repository created by the authors of the ToT paper [90] and added a task for code analysis [91]. For the strategy to work properly, we need to make slight modifications to the prompt and we need to have another prompt for the evaluation step. We call the main prompt  $p_{tot-8s}$  and the evaluation prompt  $p_{tot-8s-eval}$ . In our case, as the strategy had eight steps, we generated three candidates for each step and had three evaluators. This means we made  $8 \times (3 + 3) = 48$  calls to the LLM just for analysing a single file. As displayed in Table 11, for the evaluation of CWE-129, the strategy does provide slight improvements. However, for CWE-23, the results are significantly worse. Based on these results and the high costs, we do not dive deeper into testing the ToT strategy. It must be noted this strategy does have potential for future research, as there are many things to configure, like the evaluation strategy, the number of responses generated etc.

The self-consistency approach for the CoT 8-step prompt is denoted as  $p_{cot-8s-sc}$ . We run the same prompt three times and only count the classifications positive when the file is classified as positive two or three times. In Table 11, we can see the self-consistency approach with higher temperatures provides slight improvements for two of the three CWEs. As the self-consistency shows improved results, we try the self-consistency strategy for the whole dataset. This resulted in a noticeable improvement in the results, especially regarding lowering the false positives. The results are displayed in Table 12.

Table 12. *Self-consistency results*

Strategy	TP	FP	TN	FN	Accuracy	Precision	Recall	F1	Cost	Time
$p_{cot-8s}, t = 0$	160	88	201	129	0.625	0.645	0.554	0.596	13.4\$	4.9h
$p_{cot-8s-rci}, t = 0$	161	85	204	128	0.631	0.654	0.557	0.602	45.94\$	11.7h
$p_{cot-8s-sc}, t = 0.7$	164	60	229	125	0.68	0.732	0.567	0.639	40.81\$	15.4h

## 5.2.5 Different models

As all the testing so far has been conducted on the GPT-4 turbo model, we also wanted to see how it compares to other commercial LLMs, that are beloved by the users [17]. At the time of writing, the GPT-4 non-turbo model and Claude 3 Opus models are among the highest-ranking ones, so we also test these. The Google Gemini model is also considered. However, as it is not officially available in Europe at the time of running the experiments, it is not included [92]. For testing with other models, the temperature value of zero is used.

Both these models are more expensive to run, so we only run a few strategies that show good results with the GPT-4 turbo model. We choose the dataflow analysis prompt, the dataflow analysis prompt with RCI and the CoT 8-step prompt. The dataflow prompt with the RCI post-processing is the best-performing one and the CoT 8-step prompt is the best-performing strategy without any post-processing steps.

The results are displayed in Table 13. Interestingly, both of the other models show better results with the CoT 8-step prompt. The GPT-4 model with the CoT 8-step prompt outperforms all the other approaches, showing F1 scores of 0.672. The results of the Claude 3 Opus model are also intriguing. The CoT 8-step strategy shows the lowest false positive rate we have seen among the LLMs whilst maintaining a respectable true positive rate. What is more, the RCI strategy improves the scores in all tests conducted on the GPT-4 turbo model. On the Claude 3 Opus model, the RCI strategy seems to reduce the performance.

Table 13. *Other models results*

	Strategy	TP	FP	TN	FN	Accuracy	Precision	Recall	F1	Cost	Time
GPT-4 turbo	<i>pdfa</i>	150	102	187	139	0.583	0.595	0.519	0.555	9.33\$	4.8h
	<i>pdfa-rci</i>	171	57	232	118	0.697	0.75	0.592	0.662	34.58\$	20.5h
	<i>pcot-8s</i>	160	88	201	129	0.625	0.645	0.554	0.596	13.4\$	4.9h
GPT-4	<i>pdfa</i>	154	98	191	135	0.597	0.611	0.533	0.569	19.35\$	1.5h
	<i>pdfa-rci</i>	148	37	252	141	0.692	0.8	0.512	0.624	66.79\$	3.6h
	<i>pcot-8s</i>	174	55	234	115	<b>0.706</b>	0.76	<b>0.602</b>	<b>0.672</b>	23.32\$	2h
Claude 3 Opus	<i>pdfa</i>	141	71	218	148	0.621	0.665	0.488	0.563	19.32\$	3.1h
	<i>pdfa-rci</i>	112	50	239	177	0.607	0.691	0.388	0.497	65.29\$	10.8h
	<i>pcot-8s</i>	137	18	271	152	<b>0.706</b>	<b>0.884</b>	0.474	0.617	26.87\$	3.8h

## 5.2.6 Results per CWE category

The best results in terms of recall and F1 scores are achieved with the GPT-4 model using the CoT 8-step prompt. We provide an overview of the performance per CWE class in

Table 14. Notably, four of the CWEs are undetected, whereas both static analysers are unable to detect six CWEs. What is more, for all other CWEs, the number of true positives is larger than the number of false positives.

Table 14. *GPT-4*  $p_{cot-8s}$  prompt results per CWE

CWE	TP	FP	TN	FN	Accuracy	Precision	Recall	F1
CWE-23	17	3	14	0	0.912	0.85	1	0.919
CWE-36	15	2	15	2	0.882	0.882	0.882	0.882
CWE-78	17	3	14	0	0.912	0.85	1	0.919
CWE-80	12	0	17	5	0.853	1	0.706	0.828
CWE-81	6	0	17	11	0.676	1	0.353	0.522
CWE-83	17	1	16	0	0.971	0.944	1	0.971
CWE-89	17	11	6	0	0.676	0.607	1	0.756
CWE-129	16	15	2	1	0.529	0.516	0.941	0.667
CWE-190	12	4	13	5	0.735	0.75	0.706	0.727
CWE-256	2	0	17	15	0.559	1	0.118	0.211
CWE-259	13	5	12	4	0.735	0.722	0.765	0.743
CWE-321	16	0	17	1	0.971	1	0.941	0.97
CWE-476	14	11	6	3	0.588	0.56	0.824	0.667
CWE-523	0	0	17	17	0.5	0	0	0
CWE-549	0	0	17	17	0.5	0	0	0
CWE-566	0	0	17	17	0.5	0	0	0
CWE-606	0	0	17	17	0.5	0	0	0

The best results in terms of precision are achieved with the Claude 3 Opus model using the CoT 8-step prompt. While this Claude 3 Opus model does not achieve as high recall values for most CWEs as GPT-4, it produces far fewer false positive results. What is more, it is able to correctly classify CWE-549 in one case, leaving just three CWEs that are not correctly classified.

Table 15. *Claude 3 Opus*  $p_{cot-8s}$  prompt results per CWE

CWE	TP	FP	TN	FN	Accuracy	Precision	Recall	F1
CWE-23	13	0	17	4	0.882	1	0.765	0.867
CWE-36	11	0	17	6	0.824	1	0.647	0.786
CWE-78	13	0	17	4	0.882	1	0.765	0.867
CWE-80	11	0	17	6	0.824	1	0.647	0.786
CWE-81	8	3	14	9	0.647	0.727	0.471	0.571

CWE-83	14	1	16	3	0.882	0.933	0.824	0.875
CWE-89	15	1	16	2	0.912	0.938	0.882	0.909
CWE-129	7	3	14	10	0.618	0.7	0.412	0.519
CWE-190	4	0	17	13	0.618	1	0.235	0.381
CWE-256	6	0	17	11	0.676	1	0.353	0.522
CWE-259	13	3	14	4	0.794	0.813	0.765	0.788
CWE-321	16	5	12	1	0.824	0.762	0.941	0.842
CWE-476	5	2	15	12	0.588	0.714	0.294	0.417
CWE-523	0	0	17	17	0.5	0	0	0
CWE-549	1	0	17	16	0.529	1	0.059	0.111
CWE-566	0	0	17	17	0.5	0	0	0
CWE-606	0	0	17	17	0.5	0	0	0

### 5.3 Quantitative analysis summary

The most important results are given in Table 16. The best overall precision is achieved by CodeQL with its default configuration, achieving a precision score of 0.938. The best overall results from traditional static analysis tools are shown by SpotBugs with the Find Security Bugs plugin. This shows the best overall accuracy score of 0.723 while maintaining respectable precision, recall and F1 scores. We provide the best prompting strategy for each of the tested LLMs as well. The GPT-4 turbo performs best with the  $p_{dfa-rci}$  prompt, outperforming the static analysis tools in terms of F1 score. The GPT-4 performs best with the  $p_{cot-8s}$  prompt, showing the best recall and F1 scores overall. The Claude 3 Opus model performs best with the  $p_{cot-8s}$  prompt, showing the lowest false positive rate out of the tested LLM approaches. In terms of price and time, the traditional tools provide a much better value.

Table 16. Results overview

	TP	FP	TN	FN	Accuracy	Precision	Recall	F1	Cost	Time
CodeQL-d	76	5	284	213	0.623	<b>0.938</b>	0.263	0.411	0\$	<1m
CodeQL-esq	137	23	266	152	0.697	0.856	0.474	0.61	0\$	<1m
SpotBugs-fsb	152	23	266	137	<b>0.723</b>	0.869	0.526	0.655	0\$	<1m
GPT-4 turbo $p_{dfa-rci}$	171	57	232	118	0.697	0.75	0.592	0.662	34.58\$	20.5h
GPT-4 $p_{cot-8s}$	174	55	234	115	0.706	0.76	<b>0.602</b>	<b>0.672</b>	23.32\$	2h
Claude 3 Opus $p_{cot-8s}$	137	18	271	152	0.706	0.884	0.474	0.617	26.87\$	3.8h



## 5.4 Qualitative analysis

For static analysis tools to provide value, they must not only point at a vulnerability but also explain their findings. We compare the outputs from CodeQL, SpotBugs and two LLM model outputs. The best-performing approach for each tool is used for the comparison. For CodeQL, the output of CodeQL-esq configuration is used, for SpotBugs, the output of SpotBugs-fsb configuration is used. For LLM, the GPT-4 and Claude 3 Opus models with CoT 8-step prompting results are used. The choices of LLMs are based on the best overall recall, precision and F1 scores. All the relevant code samples are available in Appendix 3 – Code Samples.

As the outputs from the tools can be quite long, we focus on two files. We look at one vulnerable file that all the approaches correctly classified as vulnerable and one non-vulnerable file that all approaches incorrectly classified as vulnerable. For the file selection, we found only one such file, which was not vulnerable but incorrectly classified to be vulnerable by all four approaches. For the correctly classified file, we randomly selected one file that was correctly identified to be vulnerable by all four approaches.

### 5.4.1 True positive classifications

The file named "J20736" in the dataset is vulnerable to CWE-78: OS command injection. All approaches we cover correctly identify the file to be vulnerable. The vulnerable function assigns the variable "data" a user-provided value, which is then used to execute the system command.

The CodeQL-esq scan reports the issue correctly. The problem is reported as an "Uncontrolled command line", with a good human-readable description and the line numbers. The result is in CSV format, with the values in order: name, description, severity, message, path, start line, start column, end line, end column [93].

```
"Uncontrolled command line", "Using externally controlled strings in a command line is vulnerable to malicious changes in the strings.", "error", "This command line depends on a [{"user-provided value"}|"relative:///src/testcases/CWE78_OS_Command_Injection/J20736.java:13:65:13:88"]].", "/src/testcases/CWE78_OS_Command_Injection/J20736.java", "31", "53", "31", "68"
```

The SpotBugs-fsb scan correctly identifies the issue and the line number. The first letter in the result refers to the severity of the issue, where "H" denotes high severity. The second letter denotes the type of the problem, where "S" stands for security issue. Then there is a description of the problem and lastly, the file name and line number are given.

```
H S SECCI: This usage of java/lang/Runtime.exec(Ljava/lang/String;)Ljava/lang/Process;
can be vulnerable to Command Injection At J20736.java:[line 31]
```

The GPT-4 LLM (using  $p_{cot-8s}$ ) correctly identifies the relevant issue. The verdict contains the correct CWE identifier and the correct description. The description is concise and easy to follow, in a nice human-readable format.

1. Identify Weaknesses: The code appears to have a command injection vulnerability.
2. Review User Input Handling: The code takes user input from the HTTP request query string without any validation or sanitization and uses it in a system command.
3. Analyze Data Flow: The untrusted data from the HTTP request query string is tokenized and if a token starts with "id=", the rest of the token is used as data. This data is then appended to a system command and executed.
4. Check for Mitigations: There are no mitigations in place to prevent command injection. The user input is not validated or sanitized before being used in a system command.
5. Evaluate Conditional Branching: There is no dead code in the program. All branches can be reached.
6. Assess Error Handling: The code does not handle any errors that may occur during the execution of the system command.
7. Identify Code Leaking Secrets: The code does not contain any secrets.
8. Provide verdict:
  - vulnerability: YES | vulnerability type: CWE-78 (OS Command Injection) | The code executes a system command with unsanitized user input, leading to a command injection vulnerability.

The Claude 3 Opus LLM (using  $p_{cot-8s}$ ) also finds the issue and provides just as nice human-readable description as the GPT-4 model.

Here is my analysis of the provided Java code:

1. Identify Weaknesses: The code may be vulnerable to OS command injection (CWE-78) if the 'data' variable is influenced by user input.
2. Review User Input Handling: The 'data' variable is populated from the query string parameter "id" without any validation or sanitization. This allows user input to directly influence the 'data' variable.
3. Analyze Data Flow: The untrusted 'data' variable is appended to the 'osCommand' string which is then passed to Runtime.exec() to execute a system command. There is no sanitization of 'data' before using it in the command.
4. Check for Mitigations: There are no input validation or sanitization steps to mitigate command injection. The 'data' parameter is used unsafely in the execution of a system command.
5. Evaluate Conditional Branching: The code has a conditional branch based on PRIVATE\_STATIC\_FINAL\_TRUE which is always true. The else block setting data to null is unreachable dead code and not relevant.

6. Assess Error Handling: The code does not contain any explicit error handling for the command execution.

7. Identify Code Leaking Secrets: The code does not appear to leak any secrets.

8. Verdict:

vulnerability: YES | vulnerability type: CWE-78 |

In summary, the code is vulnerable to OS command injection (CWE-78) because the 'data' variable from the "id" query string parameter is used without sanitization to construct a system command that is executed. An attacker could inject arbitrary OS commands via the "id" parameter.

Overall, the file was correctly identified to be vulnerable to OS command injection by all four approaches. All four outputs are clear on the vulnerability type, with the LLM response being the most verbose.

## 5.4.2 False positive classifications

The file named "J23877" in the dataset is not vulnerable to CWE-89: SQL injection. However, all the approaches incorrectly identify the file to be vulnerable. The file contains two non-vulnerable functions, one with a good source and a bad sink, and the other with a bad source and a good sink. While these functions do not follow the best practices, neither function can be exploited.

The CodeQL-esq scan reports two issues related to SQL injection. The problem we are interested in is reported as a "Query built by concatenation with a possibly-untrusted string". The description states that the value used in the query "may be untrusted". It is indeed correct that using string concatenation is not the best practice for SQL queries and prepared statements should be used instead. However, in this case, the code is not vulnerable, as the variable can not be set by the user. The result is still counted as a false positive, as the problem is reported as SQL injection instead of not following the best practices.

```
"Query built by concatenation with a possibly-untrusted string", "Building a SQL or Java Persistence query by concatenating a possibly-untrusted string is vulnerable to insertion of malicious code.", "error", "Query built by concatenation with [\""this expression\"|\"relative:///src/testcases/CWE89_SQL_Injection/s02/J23877.java :24:101:24:108\""], which may be untrusted.", "/src/testcases/CWE89_SQL_Injection/s02/J23877.java", "24", "47", "24", "114"
```

```
"Query built by concatenation with a possibly-untrusted string", "Building a SQL or Java Persistence query by concatenating a possibly-untrusted string is vulnerable to insertion of malicious code.", "error", "Query built by concatenation with [\""this expression\"|\"relative:///src/testcases/CWE89_SQL_Injection/s02/J23877.java :62:101:62:108\""], which may be untrusted.", "/src/testcases/CWE89_SQL_Injection/s02/J23877.java", "62", "47", "62", "114"
```

The SpotBugs-fsb reports one SQL injection-related issue. The issue is reported as a medium level security issue. The wording hints at a possible SQL injection, which in the given case can not happen and thus is counted as a false positive.

```
M S SQL: testcases.CWE89_SQL_Injection.s02.J23877.processG2B(HttpServletRequest,
    HttpServletResponse) passes a nonconstant String to an execute or addBatch method on
    an SQL statement At J23877.java:[line 62]
```

The GPT-4 LLM (using  $p_{cot-8s}$ ) marks the function with the bad source to be vulnerable, as the code does not use a prepared statement for the SQL query. The analysis fails to notice the value concatenated to the SQL statement is hard coded and thus is not considered a vulnerability.

1. Identify Weaknesses: The code has potential SQL Injection vulnerabilities.
2. Review User Input Handling: The user input is taken directly from the HTTP request parameters and used in SQL queries without proper validation or sanitization in the `processB2G` method.
3. Analyze Data Flow: The untrusted data from the HTTP request parameters is directly used in the SQL query in the `processB2G` method.
4. Check for Mitigations: The code uses `PreparedStatement` in the `processB2G` method which mitigates the SQL Injection vulnerability. However, in the `processG2B` method, the code uses `Statement` which is vulnerable to SQL Injection.
5. Evaluate Conditional Branching: There is no dead code in the given code.
6. Assess Error Handling: Errors are logged with a warning level, but the execution continues. This could potentially lead to further errors or unexpected behavior.
7. Identify Code Leaking Secrets: The code does not seem to contain any secrets.
8. Provide verdict:
  - vulnerability: YES | vulnerability type: CWE-89 (SQL Injection) | The `processG2B` method is vulnerable to SQL Injection as it uses `Statement` to execute SQL queries.
  - vulnerability: NO | vulnerability type: CWE-89 (SQL Injection) | The `processB2G` method is not vulnerable to SQL Injection as it uses `PreparedStatement` to execute SQL queries.

The Claude 3 Opus LLM (using  $p_{cot-8s}$ ) also marks the function with the bad source to be vulnerable, as the code does not use a prepared statement for the SQL query. However, in this case, the LLM correctly identifies the code is not currently exploitable.

Here is my analysis of the provided Java code:

1. Identify Weaknesses: The code may be vulnerable to SQL injection in the processG2B method. It directly concatenates user input into SQL queries without proper sanitization.

```
2. Review User Input Handling: In processG2B, the 'data' variable is hardcoded to "foo"
   regardless of the conditional branching. It does not come from user input. In
   processB2G, the 'data' variable comes from the 'name' request parameter, which is
   user input. However, it is properly parameterized in the SQL query using a
   PreparedStatement.

3. Analyze Data Flow: In processG2B, the hardcoded 'data' value flows unsanitized into
   the SQL query. In processB2G, the user input 'data' is safely passed to the SQL
   query via a parameterized PreparedStatement.

4. Check for Mitigations: The processB2G method mitigates SQL injection by using a
   parameterized PreparedStatement. The processG2B method has no mitigations and
   concatenates the 'data' value directly into the query.

5. Evaluate Conditional Branching: The conditional branching in both methods does not
   affect the data flow or vulnerability. The same code is executed regardless of the
   branch taken.

6. Assess Error Handling: Database and SQL errors are caught and logged at the WARNING
   level. The application continues execution after logging.

7. Identify Code Leaking Secrets: No sensitive information like passwords, keys or
   secrets are leaked in the provided code.

8. Verdict:
vulnerability: YES | vulnerability type: CWE-89 | details: Potential SQL injection in
   processG2B due to direct concatenation of hardcoded data into SQL query.
vulnerability: NO | vulnerability type: CWE-89 | details: SQL injection mitigated in
   processB2G by using parameterized PreparedStatement.

In summary, there is a potential SQL injection vulnerability in the processG2B method,
   but it is not currently exploitable since the concatenated data is hardcoded and not
   influenced by user input. The processB2G method is not vulnerable to SQL injection
   due to proper use of parameterized queries.
```

Overall, all approaches incorrectly identify an SQL injection vulnerability in the code. The wording of the problem in the case of CodeQL and SpotBugs hints that string concatenation should not be used in SQL statements. Similarly, Claude 3 Opus model correctly mentions that string concatenation should not be used. GPT-4 incorrectly states the function to be vulnerable to SQL injection. We classify all results as false positives, as the code is not exploitable, but all approaches report SQL injection vulnerabilities. The Claude 3 Opus model provides the best verdict, correctly noticing that the code is currently not exploitable. However, it still reports: "vulnerability: YES | vulnerability type: CWE-89", which we count as a positive classification.

## 6. Discussion

### 6.1 Findings

#### 6.1.1 RQ1: What prompting approach is most successful with the LLMs to detect vulnerabilities?

Many different prompting strategies have been proposed by previous studies for vulnerability detection with LLMs [7, 8, 9, 10]. The studies use different datasets, they try to detect different CWEs and some frame the problem as binary classification, others as multi-class classification task [7, 8, 9, 10]. Compiling a dataset of 578 Java files covering 17 different CWEs allows us to compare different prompting strategies and models to see which performs the best for multi-class classification tasks. The reason for testing the multi-class classification capabilities is that this most closely mimics what is required for vulnerability detection in the real-world setting. The traditional static analysis tools are even more user-friendly, providing the exact line numbers for the vulnerability, which is called fine-grained classification. We believe LLMs come close to that ability, as they explain the problem in a human-readable format. However, we do not ask for exact line numbers to simplify the evaluation of the correctness of the responses.

Most prior studies utilise GPT-3.5 and GPT-4 models, with the latter outperforming the former in vulnerability detection tasks [55, 60, 67]. Unfortunately, in most cases, the exact versions of the LLMs are not provided, which makes it difficult to compare the results. We run all our initial testing on the GPT-4 turbo model with zero temperature values. After establishing the best-performing prompt from previous studies, we suggest other prompting strategies based on the research in the prompt engineering field. We try prompting approaches that have shown promise on different types of tasks and adapt them for vulnerability detection tasks. We test the CoT [34] and different self-refinement strategies [41]. What is more, we do some testing with prompting approaches requiring higher temperature values like self-consistency [19] and ToT [18]. Finally, we test the most promising approaches on more expensive models, where the best overall results are shown with a CoT 8-step prompt devised by us. To the best of our knowledge, ToT and self-consistency have not been tested for vulnerability detection tasks before. Some self-refinement strategies have been tested before [7] but we try different approaches for self-refinement [40, 7, 41] and find RCI strategy works best for our use case. While CoT prompting has been suggested before for vulnerability detection tasks [8], these efforts

are very different from ours. The CoT prompt proposed previously considers a two-step approach: the first step is to explain the code and the second is to find issues with the code [8]. Another previously suggested CoT strategy tests few-shot CoT approaches [58]. The CoT 8-step prompt we propose lists eight steps that should be taken to find potential issues in source code.

For compiling the CoT 8-step prompt, we first ask the GPT-4 turbo model, which steps should be taken to find vulnerabilities in the source code. Based on the responses, a prompt is compiled that asks the LLM to take steps similar to what a human would do when reviewing source code. This prompt is then tested on a few files and improved based on the results. While this approach does not outperform the dataflow analysis prompt on the GPT-4 turbo model, it shows the best results on other models we test. This shows that different prompting techniques can have advantages on different models and the prompt should be adapted to the used LLM.

The best results achieved with LLMs slightly outperform CodeQL and SpotBugs analysis results. Depending on the LLM used, different prompting approaches can be more successful. We suggest using the CoT 8-step prompt proposed by us with either GPT-4 or Claude 3 Opus model. It must be noted that in our case half the files in the dataset are vulnerable. In the real world, there will likely be multiple non-vulnerable files for every vulnerable one. This is also likely one of the reasons why for example the CodeQL default configuration had a low recall, but high precision value. High precision means fewer false alarms and better user experience. Industry experts seem to place a high value on having as few false positives as possible, with Google noting false positives should make up less than 10% of all reported issues [22].

If the expectation is to find as many vulnerabilities as possible and the amount of false positive results is not as important, then we suggest using the GPT-4 model. This approach has the best recall and F1 scores, which means on paper, this is the best approach. If the expectation is to get as few false positive results as possible, we suggest using the Claude 3 Opus model. Claude 3 Opus model produces impressively few false positive results and could thus be the most pleasant to use in real-world scenarios. This is also the only approach that gets close to the 10% figure ( $\frac{18 \times 100}{155} = 11.6\%$ ) of false positives that Google expects [22]. The GPT-4 turbo model using the dataflow analysis prompt with the RCI strategy also performs well. Although the cost per token for the GPT-4 turbo model is significantly cheaper, the RCI strategy adds extra complexity, requiring more tokens. This makes the best-performing strategy on the GPT-4 turbo model more expensive than the better-performing GPT-4 or Claude 3 Opus models. Thus we recommend using the GPT-4 and Claude 3 Opus models for vulnerability detection tasks.

### **6.1.2 RQ2: What advantages and disadvantages do LLMs have over existing static analysis tools?**

The most important disadvantage of LLMs over existing static analysis tools is the cost, both monetary and time-wise. While the high cost of LLMs is mentioned often when discussing LLMs for vulnerability detection, to the best of our knowledge, the cost of different prompting approaches has not been discussed in detail. We provide details on the costs for every prompting strategy, allowing for comparison by not only the performance but also cost. The cost factor is considered when making recommendations and should be considered when opting to use commercial LLMs in vulnerability detection tasks. Depending on the dataset, different approaches could make sense. If the dataset is large, some prompting strategies, like ToT, become impractical due to their high costs. We also find that even though the RCI strategy usually helps to improve the results of OpenAI models, it significantly increases the costs, usually by a factor of three or more.

The CodeQL and SpotBugs are both easy to set up and run in their default configurations. The CodeQL is easy to run with the extended configuration but setting SpotBugs up to use the FindSecBugs plugin requires some more effort. Both tools are free to use and complete the analysis in under a minute for our dataset. This means they are orders of magnitude faster than the fastest LLM-based approaches. Even though the LLM-based approach slightly outperforms the traditional tools based on the F1 score, the difference is rather small. The main problem with the LLM-based approach is the time it takes to run and the monetary cost associated. The biggest strength of the LLM-based analysis is the ability to detect most vulnerabilities.

Both the static analysis tools completely missed six vulnerability classes from the 17 in our dataset. For LLMs, most analyses missed four CWE classes, with the Claude 3 Opus not having any correct positive classifications on only three of the CWE classes. Analysing the LLM responses manually, we see that the descriptions often point out the correct issue in the code but fail to associate the correct CWE to it. The first CWE missed by all LLM analyses is the CWE-523, which in the case of our dataset means that HTTP protocol is used where HTTPS protocol should be used. Both suggested approaches describe the issue correctly on multiple occasions, but provide a CWE identifier, which is considered incorrect by us. For example, the LLMs often classify the problem as CWE-319, which points to cleartext transmission of sensitive data. While the issues are similar, the matching strategy we use allows us to only count the parents and children of the expected CWE as correct. Based on the CWE research view (CWE-1000), CWE-319 is neither a child nor a parent of CWE-523, meaning we do not count it as a positive classification. The second



CWE missed by most LLMs is the CWE-549. In our dataset, this problem exists due to Java code generating an HTML form with a password field, where the field is marked of type “text” instead of the expected “password” type. While this problem is correctly classified once, in most cases the description matches the problem, but the CWE identifier provided by LLM is not considered correct. Similarly to CWE-523, the CWE-319 is often offered as a CWE identifier when we expect CWE-522 or CWE-523. The third CWE missed by all LLM analyses is the CWE-566. The CWE-566 means the code uses the user-provided value as the primary key in querying the database without checking the user access rights. While this is a valid vulnerability, correctly identifying the issue often requires business context knowledge. The LLMs at times do mention the user-provided value is used in the SQL query, but the issue is classified as CWE-89: SQL injection. Classifying the issue as SQL injection can be confusing, as prepared statements are used and whether the code is vulnerable depends on the context. The fourth CWE missed by all LLM analyses is the CWE-606, which is caused by using user inputs in loop conditions without proper validation. The LLMs are also able to point at the problem correctly in many instances but identify the issue as CWE-400, which points to uncontrolled resource consumption. Overall, we find the LLMs can correctly describe the problems for 16 CWE categories out of the 17 in our dataset. For comparison, CodeQL and SpotBugs can detect 11 CWE categories out of the 17. This shows the LLMs have an advantage in finding vulnerabilities not supported by traditional tools.

### **6.1.3 MRQ: How would the use of off-the-shelf LLMs be able to contribute to vulnerability detection in source code?**

While the LLMs show good ability in detecting many different types of vulnerabilities, they are significantly more expensive and time-consuming to run than traditional static analysis tools. Thus we believe that static code analysers like CodeQL and SpotBugs are currently still better for everyday use for developers. They run quickly, are easy to set up and are free to use. They are configured to produce rather few false positives and provide quite good descriptions of problems together with line numbers. However, using LLMs for security analysis could be justified in some cases, like during security audits. The LLMs show the ability to find a larger variety of issues and explain them rather well. Another advantage of using an LLM-based approach includes not needing the code to be compileable. This allows for an easy analysis of some parts of the code without providing access to the full codebase.

### 6.1.4 Differences and similarities of LLMs

Previous studies mostly utilise different OpenAI models, like GPT-3.5 and GPT-4 [7, 8, 54, 55, 57, 60, 67]. Some other studies also incorporate open-source models from the Llama [7, 54], BERT [10, 68] and Falcon series [54], with Google's Gemini model being also included in some cases [59, 58]. To the best of our knowledge, we are the first to include Claude 3 model from Anthropic in the comparisons.

The cost structure of commercial LLM API usage is similar between models, requiring users to pay for input and output tokens. Output tokens are significantly more expensive. So from the cost perspective, it is reasonable to add as much context into the input and ask to provide short answers. To save costs and to be able to automatically evaluate the analysis results, we specify the format in which to provide the final verdict. The format we use is: "vulnerability: <YES or NO> | vulnerability type: <CWE\_ID> |". Usually, the LLMs follow the provided format, but not always. Interestingly, different models have different deviations from the provided format but are consistent with the deviations. The GPT-4 turbo model often adds decorators like: "vulnerability: **\*\*YES\*\*** | vulnerability type: **\*\*CWE-89\*\***" or provides a vulnerability description instead of the keyword "vulnerability". The GPT-4 model sometimes provides the CWE identifier with an underscore: "CWE\_89". The Claude 3 Opus model often uses line changes instead of the "|" sign. Overall, for the better performing strategies, we manually inspect the responses that contain the word "YES" and make sure the format of the response would not affect the results. There are some rare cases where the model provided "vulnerability: MAYBE" or "vulnerability: POSSIBLE" instead of "YES" or "NO". As we know that the context needed for discovering the vulnerability should be given in the file, we count these as "NO". There are very few such classifications and in most cases, they refer to irrelevant CWEs. The testing we conduct with higher temperature values utilising self-consistency and ToT strategies has a lot more responses that do not adhere to the format. This is likely related to the temperature increase making the responses more random, affecting the output format. This means the testing with higher temperature values would require either a better automatic mapping strategy or more manual effort to benchmark.

## 6.2 Limitations and threats to validity

The dataset is synthetic and the vulnerability detection is tested on file level, not line or function level. This was done on purpose, as for some vulnerabilities, it can be hard to assign a specific line to the vulnerability. Similarly, most previous studies on the topic test binary or multi-class classification capabilities, not fine-grained ones.

We use a dataset, where the vulnerabilities can be detected based on a single file. Due to the context length limitations, this is the best-case scenario for LLM vulnerability detection, as the code samples are rather short. This allows us to avoid adding a lot of relevant code to the prompt or adding capabilities for the LLM to fetch the relevant code. The detection capabilities of any of the tested tools might not directly translate to more complex real-world projects. In our case, half the files are vulnerable and the other half are not. In real-world projects, most files do not contain vulnerabilities. Even the well-established SAST tools, which scan the whole codebase at once, have been shown to perform significantly worse on real-world datasets when compared to synthetic ones [81]. However, there have been studies using LLMs for vulnerability detection using real-world datasets, which have shown good results [10, 64]. What is more, it is likely the capabilities of the LLMs will improve for large more complex codebases, where multiple files need to be analysed at once. The token limits have been increasing rapidly in the past years, with Google announcing having successfully tested context windows of up to ten million tokens [94]. The larger context window can help translate the results seen on synthetic datasets to real-world projects.

17 unique CWE categories are covered, which map to 11 CWEs from MITRE's top 25. While this might not cover all important vulnerability categories, it does cover a large portion of what has been classified as the most dangerous by MITRE [95]. We only focus on Java programming language, however, based on prior research, the approaches likely translate reasonably well to other programming languages [7]. Our contribution includes benchmarking on a unique dataset containing more CWEs than most previous studies. Most prior studies have focused on five or fewer CWEs to evaluate the results [7, 54, 57, 67].

The Juliet dataset might be present in the training data of the LLM, which could affect the results. To mitigate that issue as much as possible, we conduct extensive pre-processing of the dataset. During the pre-processing, file structure, file naming, the names of variables and the names of functions are changed. Furthermore, the package names are also hidden from the LLM and the indentation and format of the files are altered. These modifications should make it more difficult for the model to base the predictions on what has been present in the training data, as the files differ noticeably.

We employ a strategy for mapping CWEs which does not expect the CWE identifier to be identified exactly, similar to studies of static code analysers [4, 80]. By using the CWE-1000 research view, we can in most cases match the provided CWE to the expected one, if the issue is identified correctly. However, we notice that for four of the CWEs in our dataset, this strategy does not work well. Manually reviewing the LLM responses, we see

correctly identified issues, which we can not automatically consider correct. Future work could evaluate options for more accurately matching CWEs. One option for that could be using another LLM to provide all CWEs that match the output description. Similar strategies have been tried previously [64]. Another option would be to employ manual steps for evaluating the results, although this would be very time-consuming and could introduce biases. While both proposed approaches would increase true positive results, they would also increase the false positive count. That is because more files would be classified as containing the expected vulnerability and among those files would also be non-vulnerable ones.

Due to their high costs, we do not further evaluate the capabilities of GPT-4 and Claude 3 Opus models. The performance of these models could likely be further improved with higher temperatures utilising a self-consistency strategy, just like we saw improvements for the GPT-4 turbo model. We are able to show LLMs can indeed compete with classical static analysis tools and actually outperform them in some settings. Most notably, the LLMs seem to be able to detect a much wider array of different types of issues in code.

To limit the scope, we focus on static code analysis. Incorporating other methods like dynamic analysis or other DL methods is out of the scope. Fine-tuning LLMs is out of scope, as the focus is on the prompting techniques. We recognise that the LLMs and the rulesets of the static code analysis tools can change over time. The exact versions of the tools and datasets used are denoted in the Appendix 4 – Versions.

Data leakage to the LLM owners and/or hosting services should be considered before using commercial LLM APIs. Before sending any confidential data to any of the LLM providers, policies should be in place to ensure the safety of the confidential data. The OpenAI privacy policy states that by default, the data provided by the users of ChatGPT can be used to train the models, however, there is an opt-out option [96]. To mitigate the privacy and cost concerns, local LLMs could be used. There are frameworks like Ollama<sup>1</sup> that allow running open-source models locally. However, the performance of these models should be tested, as currently the commercial models are voted to be more powerful than the open-source ones [17].

---

<sup>1</sup><https://ollama.com/>

## 7. Summary

We consider the use of state-of-the-art LLMs for vulnerability detection tasks and compare the results with two traditional static code analysers. The purpose is to discover if LLMs could help in detecting vulnerabilities in source code. We are interested in whether LLMs have advantages or disadvantages over existing static code analysis tools. We run experiments and use comparative analysis techniques to evaluate the performance of different approaches. We consider different prompt engineering techniques previously not tested for vulnerability detection tasks. We find off-the-shelf LLMs show remarkable abilities in file-level vulnerability detection tasks. The success of a particular prompting strategy is dependent on the underlying LLM. The GPT-4 turbo model shows the best performance with dataflow analysis prompt utilising the RCI strategy. Meanwhile, GPT-4 and Claude 3 Opus models show better performance with a CoT 8-step prompt. The best prompting approaches outperform the static code analysis tools based on recall and F1 scores. The advantages of LLMs over static analysis tools include the ability to detect a larger variety of different vulnerabilities and a higher amount of true positive classifications. The disadvantages of LLMs include slower running time, higher costs, non-deterministic results and a higher amount of false positives. Thus we can show LLMs show a remarkable ability in vulnerability detection and multi-class classification tasks when all required information is provided in the context.

## 8. Future work

The LLMs are getting a lot of attention and have improved a lot over the past few years. As long as they continue to improve and more capable models are released, the capabilities of these new models should be tested. A comparison of the performance and cost between commercial models and open-source models would be an interesting area to explore. Overall, more combinations of prompting strategies and LLMs should be tested to discover the best approaches. The ToT prompting strategy allows for many variations of different prompts and parameters, remaining a compelling area for future research. Fine-tuning the LLMs has been tried on some smaller models [7], but could be an interesting area for further exploration.

Using other datasets for testing the proposed prompting approaches could provide valuable information on how well different prompts perform on average. Ideally, the testing would be done on a proprietary dataset, which we know not to be present in the training data. That would help in keeping the results unbiased. Including more programming languages and testing the LLM capabilities for fine-grained vulnerability detection could be another vertical to explore. Future studies could improve upon the CWE matching strategy, to make sure the correctly described problems are counted as positive classifications, without relying solely on the LLM to provide an acceptable CWE identifier.

The potential synergies between traditional tools and LLMs could be further researched. The LLMs can generate fixes for code but the quality of these fixes has not been evaluated in detail [9]. What is more, it would be intriguing to test the capabilities of LLMs for generating tests to prove the vulnerability is present. This could be done to reduce the amount of false positive classifications.

## References

- [1] *We need a safer systems programming language*. 2019. URL: <https://msrc.microsoft.com/blog/2019/07/we-need-a-safer-systems-programming-language/>.
- [2] CWE. *Metrics*. [Accessed: 30-09-2023]. URL: <https://www.cve.org/About/Metrics>.
- [3] Nurul Haszeli Ahmad, SA Aljunid, and J-I Ab Manan. “Preventing Exploitation on Software Vulnerabilities: Why Most Static Analysis Is Ineffective”. In: *Conferences on Engineering and Technology Education*. 2010.
- [4] Katerina Goseva-Popstojanova and Andrei Perhinschi. “On the capability of static code analysis to detect security vulnerabilities”. In: *Information and Software Technology* 68 (2015), pp. 18–33.
- [5] Christopher D. Manning. “Human Language Understanding & Reasoning”. In: *Daedalus* 151.2 (May 2022), pp. 127–138. ISSN: 0011-5266. DOI: 10.1162/daed\_a\_01905. eprint: [https://direct.mit.edu/daed/article-pdf/151/2/127/2060607/daed\\_a\\_01905.pdf](https://direct.mit.edu/daed/article-pdf/151/2/127/2060607/daed_a_01905.pdf). URL: [https://doi.org/10.1162/daed\\_a\\_01905](https://doi.org/10.1162/daed_a_01905).
- [6] Pengfei Liu et al. “Pre-Train, Prompt, and Predict: A Systematic Survey of Prompting Methods in Natural Language Processing”. In: 55.9 (Jan. 2023). ISSN: 0360-0300. DOI: 10.1145/3560815. URL: <https://doi.org/10.1145/3560815>.
- [7] Avishree Khare et al. *Understanding the Effectiveness of Large Language Models in Detecting Security Vulnerabilities*. 2023. arXiv: 2311.16169 [cs.CR].
- [8] Chenyuan Zhang et al. *Prompt-Enhanced Software Vulnerability Detection Using ChatGPT*. 2023. arXiv: 2308.12697 [cs.SE].
- [9] David Noever. *Can Large Language Models Find And Fix Vulnerable Software?* 2023. arXiv: 2308.10345 [cs.SE].
- [10] Xin Zhou, Ting Zhang, and David Lo. *Large Language Model for Vulnerability Detection: Emerging Results and Future Directions*. 2024. arXiv: 2401.15468 [cs.SE].
- [11] Wojciech Zaremba and Greg Brockman. *OpenAI Codex*. [Accessed: 30-09-2023]. URL: <https://openai.com/blog/openai-codex>.

- [12] *Juliet Java 1.3*. [Accessed: 12-12-2023]. 2017. URL: <https://samate.nist.gov/SARD/test-suites/111>.
- [13] CodeQL. *CodeQL*. [Accessed: 07-04-2024]. URL: <https://codeql.github.com/>.
- [14] SpotBugs. *SpotBugs*. [Accessed: 07-04-2024]. URL: <https://spotbugs.github.io/>.
- [15] Arvinder Kaur and Ruchikaa Nayyar. “A Comparative Study of Static Code Analysis tools for Vulnerability Detection in C/C++ and JAVA Source Code”. In: *Procedia Computer Science* 171 (2020). Third International Conference on Computing and Network Communications (CoCoNet’19), pp. 2023–2029. ISSN: 1877-0509. DOI: <https://doi.org/10.1016/j.procs.2020.04.217>. URL: <https://www.sciencedirect.com/science/article/pii/S1877050920312023>.
- [16] Richard Amankwah et al. “Bug detection in Java code: An extensive evaluation of static analysis tools using Juliet Test Suites”. In: *Software: Practice and Experience* 53.5 (2023), pp. 1125–1143. DOI: <https://doi.org/10.1002/spe.3181>. eprint: <https://onlinelibrary.wiley.com/doi/pdf/10.1002/spe.3181>. URL: <https://onlinelibrary.wiley.com/doi/abs/10.1002/spe.3181>.
- [17] HuggingFace. *LMSYS Chatbot Arena Leaderboard*. [Accessed: 02-04-2024]. URL: <https://huggingface.co/spaces/lmsys/chatbot-arena-leaderboard>.
- [18] Shunyu Yao et al. *Tree of Thoughts: Deliberate Problem Solving with Large Language Models*. 2023. arXiv: 2305.10601 [cs.CL].
- [19] Xuezhi Wang et al. “Self-consistency improves chain of thought reasoning in language models”. In: *arXiv preprint arXiv:2203.11171* (2022).
- [20] Gary McGraw. “Automated Code Review Tools for Security”. In: *Computer* 41.12 (2008), pp. 108–111. DOI: [10.1109/MC.2008.514](https://doi.org/10.1109/MC.2008.514).
- [21] Anne Edmundson et al. “An Empirical Study on the Effectiveness of Security Code Review”. In: *Engineering Secure Software and Systems*. Ed. by Jan Jürjens, Benjamin Livshits, and Riccardo Scandariato. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 197–212. ISBN: 978-3-642-36563-8.
- [22] Caitlin Sadowski et al. “Lessons from Building Static Analysis Tools at Google”. In: *Communications of the ACM (CACM)* 61 Issue 4 (2018), pp. 58–66. URL: <https://dl.acm.org/citation.cfm?id=3188720>.



- [23] Norah Ahmed Alzubairik and Gary Wills. “Automated penetration testing based on a threat model”. In: *2016 11th International Conference for Internet Technology and Secured Transactions (ICITST)*. 2016, pp. 413–414. DOI: 10.1109/ICITST.2016.7856742.
- [24] Hossain Shahriar and Mohammad Zulkernine. “Mitigating Program Security Vulnerabilities: Approaches and Challenges”. In: *ACM Computing Surveys - CSUR 44* (June 2012), pp. 1–46. DOI: 10.1145/2187671.2187673.
- [25] Jernej Novak, Andrej Krajnc, and Rok Žontar. “Taxonomy of static code analysis tools”. In: *The 33rd International Convention MIPRO*. 2010, pp. 418–422.
- [26] Korhan Akcura et al. “Static Versus Dynamic Source Code Analysis”. In: ().
- [27] Seyed Mohammad Ghaffarian and Hamid Reza Shahriari. “Software Vulnerability Analysis and Discovery Using Machine-Learning and Data-Mining Techniques: A Survey”. In: 50.4 (Aug. 2017). ISSN: 0360-0300. DOI: 10.1145/3092566. URL: <https://doi.org/10.1145/3092566>.
- [28] Ella Bounimova, Patrice Godefroid, and David Molnar. “Billions and billions of constraints: Whitebox fuzz testing in production”. In: *2013 35th International Conference on Software Engineering (ICSE)*. 2013, pp. 122–131. DOI: 10.1109/ICSE.2013.6606558.
- [29] Abubakar Omari Abdallah Semasaba et al. “Literature survey of deep learning-based vulnerability analysis on source code”. In: *IET Software* 14.6 (2020), pp. 654–664. DOI: <https://doi.org/10.1049/iet-sen.2020.0084>. eprint: <https://ietresearch.onlinelibrary.wiley.com/doi/pdf/10.1049/iet-sen.2020.0084>. URL: <https://ietresearch.onlinelibrary.wiley.com/doi/abs/10.1049/iet-sen.2020.0084>.
- [30] Guanjun Lin et al. “Software Vulnerability Detection Using Deep Neural Networks: A Survey”. In: *Proceedings of the IEEE* 108.10 (2020), pp. 1825–1848. DOI: 10.1109/JPROC.2020.2993293.
- [31] Saikat Chakraborty et al. “Deep Learning Based Vulnerability Detection: Are We There Yet?” In: *IEEE Transactions on Software Engineering* 48.9 (2022), pp. 3280–3296. DOI: 10.1109/TSE.2021.3087402.
- [32] Hammond Pearce et al. “Asleep at the Keyboard? Assessing the Security of GitHub Copilot’s Code Contributions”. In: *2022 IEEE Symposium on Security and Privacy (SP)*. 2022, pp. 754–768. DOI: 10.1109/SP46214.2022.9833571.
- [33] Tom Brown et al. “Language models are few-shot learners”. In: *Advances in neural information processing systems* 33 (2020), pp. 1877–1901.

- [34] Jason Wei et al. *Chain-of-Thought Prompting Elicits Reasoning in Large Language Models*. 2023. arXiv: 2201.11903 [cs.CL].
- [35] OpenAI. *API reference: Chat*. [Accessed: 25-02-2024]. URL: <https://platform.openai.com/docs/api-reference/chat>.
- [36] Anthropic. *Create a Message*. [Accessed: 30-03-2024]. URL: [https://docs.anthropic.com/claude/reference/messages\\_post](https://docs.anthropic.com/claude/reference/messages_post).
- [37] Or Honovich et al. *Instruction Induction: From Few Examples to Natural Language Task Descriptions*. 2022. arXiv: 2205.10782 [cs.CL].
- [38] Yongchao Zhou et al. “Large Language Models Are Human-Level Prompt Engineers”. In: (2022). arXiv: 2211.01910 [cs.LG].
- [39] OpenAI. *OpenAI Codex*. [Accessed: 09-01-2024]. URL: <https://platform.openai.com/docs/guides/prompt-engineering/strategy-test-changes-systematically>.
- [40] Geunwoo Kim, Pierre Baldi, and Stephen McAleer. *Language Models can Solve Computer Tasks*. 2023. arXiv: 2303.17491 [cs.CL].
- [41] Aman Madaan et al. *Self-Refine: Iterative Refinement with Self-Feedback*. 2023. arXiv: 2303.17651 [cs.CL].
- [42] Yongqin Xian et al. “Zero-Shot Learning—A Comprehensive Evaluation of the Good, the Bad and the Ugly”. In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 41.9 (2019), pp. 2251–2265. DOI: 10.1109/TPAMI.2018.2857768.
- [43] Jason Wei et al. *Finetuned Language Models Are Zero-Shot Learners*. 2022. arXiv: 2109.01652 [cs.CL].
- [44] Robert L. Logan IV et al. *Cutting Down on Prompts and Parameters: Simple Few-Shot Learning with Language Models*. 2021. arXiv: 2106.13353 [cs.CL].
- [45] Yifei Li et al. “Making Language Models Better Reasoners with Step-Aware Verifier”. In: *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Ed. by Anna Rogers, Jordan Boyd-Graber, and Naoaki Okazaki. Toronto, Canada: Association for Computational Linguistics, July 2023, pp. 5315–5333. DOI: 10.18653/v1/2023.acl-long.291. URL: <https://aclanthology.org/2023.acl-long.291>.
- [46] Shunyu Yao et al. *ReAct: Synergizing Reasoning and Acting in Language Models*. 2023. arXiv: 2210.03629 [cs.CL].
- [47] Noah Shinn et al. *Reflexion: Language Agents with Verbal Reinforcement Learning*. 2023. arXiv: 2303.11366 [cs.AI].

- [48] Andy Zhou et al. *Language Agent Tree Search Unifies Reasoning Acting and Planning in Language Models*. 2023. arXiv: 2310.04406 [cs.AI].
- [49] Hong Sun et al. *AutoHint: Automatic Prompt Optimization with Hint Generation*. 2023. arXiv: 2307.07415 [cs.CL].
- [50] Yuhao Lin et al. “Vulnerability Dataset Construction Methods Applied To Vulnerability Detection: A Survey”. In: *2022 52nd Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN-W)*. 2022, pp. 141–146. DOI: 10.1109/DSN-W54100.2022.00032.
- [51] Tim Boland and Paul E Black. “Juliet 1. 1 C/C++ and java test suite”. In: *Computer* 45.10 (2012), pp. 88–90.
- [52] Xin Zhou et al. *Large Language Model for Vulnerability Detection and Repair: Literature Review and the Road Ahead*. 2024. arXiv: 2404.02525 [cs.SE].
- [53] Omer Said Ozturk et al. “New Tricks to Old Codes: Can AI Chatbots Replace Static Code Analysis Tools?” In: *Proceedings of the 2023 European Interdisciplinary Cybersecurity Conference*. EICC ’23. Stavanger, Norway: Association for Computing Machinery, 2023, pp. 13–18. ISBN: 9781450398299. DOI: 10.1145/3590777.3590780. URL: <https://doi.org/10.1145/3590777.3590780>.
- [54] Yu Nong et al. *Chain-of-Thought Prompting of Large Language Models for Discovering and Fixing Software Vulnerabilities*. 2024. arXiv: 2402.17230 [cs.CR].
- [55] Nenad Petrović. “Chat GPT-Based Design-Time DevSecOps”. In: *2023 58th International Scientific Conference on Information, Communication and Energy Systems and Technologies (ICEST)*. 2023, pp. 143–146. DOI: 10.1109/ICEST58410.2023.10187247.
- [56] Mark Chen et al. *Evaluating Large Language Models Trained on Code*. 2021. arXiv: 2107.03374 [cs.LG].
- [57] Anton Cheshkov, Pavel Zadorozhny, and Rodion Levichev. *Evaluation of ChatGPT Model for Vulnerability Detection*. 2023. arXiv: 2304.07232 [cs.CR].
- [58] Benjamin Steenhoek et al. *A Comprehensive Study of the Capabilities of Large Language Models for Vulnerability Detection*. 2024. arXiv: 2403.17218 [cs.SE].
- [59] Jiaxin Yu et al. *Security Code Review by LLMs: A Deep Dive into Responses*. 2024. arXiv: 2401.16310 [cs.SE].
- [60] Michael Fu et al. *ChatGPT for Vulnerability Detection, Classification, and Repair: How Far Are We?* 2023. arXiv: 2310.09810 [cs.SE].
- [61] Yiu Wai Chow, Max Schäfer, and Michael Pradel. *Beware of the Unexpected: Bimodal Taint Analysis*. 2023. arXiv: 2301.10545 [cs.SE].

- [62] Yinlin Deng et al. *Large Language Models are Zero-Shot Fuzzers: Fuzzing Deep-Learning Libraries via Large Language Models*. 2023. arXiv: 2212.14834 [cs.SE].
- [63] Daniel Tang et al. *CodeAgent: Collaborative Agents for Software Engineering*. 2024. arXiv: 2402.02172 [cs.SE].
- [64] Yuqiang Sun et al. *LLM4Vuln: A Unified Evaluation Framework for Decoupling and Enhancing LLMs' Vulnerability Reasoning*. 2024. arXiv: 2401.16185 [cs.CR].
- [65] Yuqiang Sun et al. *GPTScan: Detecting Logic Vulnerabilities in Smart Contracts by Combining GPT with Program Analysis*. 2023. arXiv: 2308.03314 [cs.CR].
- [66] Noble Saji Mathews et al. *LLbezpeky: Leveraging Large Language Models for Vulnerability Detection*. 2024. arXiv: 2401.01269 [cs.CR].
- [67] Zoltán Szabó and Vilmos Bilicki. "A New Approach to Web Application Security: Utilizing GPT Language Models for Source Code Inspection". In: *Future Internet* 15.10 (2023). ISSN: 1999-5903. DOI: 10.3390/fi15100326. URL: <https://www.mdpi.com/1999-5903/15/10/326>.
- [68] Zhilong Wang et al. "The Effectiveness of Large Language Models (Chatgpt and Codebert) for Security-Oriented Code Analysis". In: *Available at SSRN 4567887* ().
- [69] Tiago Espinha Gasiba et al. "I'm Sorry Dave, I'm Afraid I Can't Fix Your Code: On ChatGPT, CyberSecurity, and Secure Coding". In: *4th International Computer Programming Education Conference (ICPEC 2023)*. Ed. by Ricardo Alexandre Peixoto de Queirós and Mário Paulo Teixeira Pinto. Vol. 112. Open Access Series in Informatics (OASISs). Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2023, 2:1–2:12. ISBN: 978-3-95977-290-7. DOI: 10.4230/OASISs.ICPEC.2023.2. URL: <https://drops.dagstuhl.de/opus/volltexte/2023/18498>.
- [70] Giacomo Lanciano et al. "Analyzing Declarative Deployment Code with Large Language Models". In: *CLOSER 2023* (2023), pp. 289–296.
- [71] Sidong Feng and Chunyang Chen. *Prompting Is All You Need: Automated Android Bug Replay with Large Language Models*. 2023. arXiv: 2306.01987 [cs.SE].
- [72] Chen Tsfaty and Michael Fire. *Malicious Source Code Detection Using Transformer*. 2022. arXiv: 2209.07957 [cs.CR].
- [73] Thomas Claburn. *So you want to integrate OpenAI's bot. Here's how that worked for software security scanner Socket*. [Accessed: 10-12-2023]. 2023. URL: [https://www.theregister.com/AMP/2023/03/30/socket\\_chatgpt\\_malware/](https://www.theregister.com/AMP/2023/03/30/socket_chatgpt_malware/).

- [74] MITRE. *2023 CWE Top 25 Most Dangerous Software Weaknesses*. [Accessed: 02-02-2024]. URL: [https://cwe.mitre.org/top25/archive/2023/2023\\_top25\\_list.html](https://cwe.mitre.org/top25/archive/2023/2023_top25_list.html).
- [75] MITRE. *CWE VIEW: Research Concepts*. [Accessed: 26-02-2024]. URL: <https://cwe.mitre.org/data/definitions/1000.html>.
- [76] Karl Tamberg. *Processed Juliet Java Dataset*. <https://gitlab.cs.ttu.edu/katamb/juliet-top-25> (TalTech GitLab URL). 2024. URL: <https://github.com/katamb/juliet-top-25>.
- [77] Karl Tamberg. *Scripts for LLM Experiments*. <https://gitlab.cs.ttu.edu/katamb/thesis-scripts> (TalTech GitLab URL). 2023. URL: <https://github.com/katamb/thesis-scripts>.
- [78] NSA. *Juliet Test Suite v1.2 for Java User Guide*. [Accessed: 07-02-2024]. URL: [https://samate.nist.gov/SARD/downloads/documents/Juliet\\_Test\\_Suite\\_v1.2\\_for\\_Java\\_-\\_User\\_Guide.pdf](https://samate.nist.gov/SARD/downloads/documents/Juliet_Test_Suite_v1.2_for_Java_-_User_Guide.pdf).
- [79] Pasi Fränti and Radu Marinescu-Istodor. “Soft precision and recall”. In: *Pattern Recognition Letters* 167 (2023), pp. 115–121. ISSN: 0167-8655. DOI: <https://doi.org/10.1016/j.patrec.2023.02.005>. URL: <https://www.sciencedirect.com/science/article/pii/S0167865523000296>.
- [80] Stephan Lipp, Sebastian Banescu, and Alexander Pretschner. “An empirical study on the effectiveness of static C code analyzers for vulnerability detection”. In: *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*. 2022, pp. 544–555.
- [81] Kaixuan Li et al. “Comparison and Evaluation on Static Application Security Testing (SAST) Tools for Java”. In: *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ESEC/FSE 2023. New York, NY, USA: Association for Computing Machinery, 2023, pp. 921–933. DOI: [10.1145/3611643.3616262](https://doi.org/10.1145/3611643.3616262). URL: <https://doi.org/10.1145/3611643.3616262>.
- [82] GitHub. *About code scanning with CodeQL*. [Accessed: 20-02-2024]. URL: <https://docs.github.com/en/code-security/code-scanning/introduction-to-code-scanning/about-code-scanning-with-codeql>.
- [83] CodeQL. *Supported languages and frameworks*. [Accessed: 24-02-2024]. URL: <https://codeql.github.com/docs/codeql-overview/supported-languages-and-frameworks/>.
- [84] CodeQL. *CWE coverage for Java and Kotlin*. [Accessed: 24-02-2024]. URL: <https://codeql.github.com/codeql-query-help/java-cwe/>.

- [85] SpotBugs. *Introduction*. [Accessed: 24-02-2024]. URL: <https://spotbugs.readthedocs.io/en/latest/introduction.html#>.
- [86] Harrison Chase. *LangChain*. Oct. 2022. URL: <https://github.com/langchain-ai/langchain>.
- [87] OpenAI. *Rate limits*. [Accessed: 03-05-2024]. URL: <https://platform.openai.com/docs/guides/rate-limits/usage-tiers>.
- [88] Anthropic. *Rate limits*. [Accessed: 03-05-2024]. URL: <https://docs.anthropic.com/claude/reference/rate-limits>.
- [89] Michaela Greiler. *Security code review checklist*. [Accessed: 18-02-2024]. URL: <https://www.awesomecodereviews.com/checklists/secure-code-review-checklist/>.
- [90] Shunyu Yao et al. *Tree of Thoughts: Deliberate Problem Solving with Large Language Models*. 2023. URL: <https://github.com/princeton-nlp/tree-of-thought-llm>.
- [91] Karl Tamberg. *Tree of Thoughts: Deliberate Problem Solving with Large Language Models*. <https://gitlab.cs.ttu.edu/katamb/tree-of-thought-llm-ca> (TalTech GitLab URL). 2024. URL: <https://github.com/katamb/tree-of-thought-llm-ca>.
- [92] Google. *Available languages and regions for Google AI Studio and Gemini API*. [Accessed: 09-05-2024]. URL: <https://ai.google.dev/gemini-api/docs/available-regions>.
- [93] CodeQL. *CodeQL CLI CSV output*. [Accessed: 25-02-2024]. URL: <https://docs.github.com/en/code-security/codeql-cli/using-the-advanced-functionality-of-the-codeql-cli/csv-output>.
- [94] Sundaran Pichai and Demis Hassabis. *Our next-generation model: Gemini 1.5*. [Accessed: 01-03-2024]. URL: <https://blog.google/technology/ai/google-gemini-next-generation-model-february-2024/#sundar-note>.
- [95] MITRE. *2023 CWE Top 25 Methodology*. [Accessed: 01-03-2024]. URL: [https://cwe.mitre.org/top25/archive/2023/2023\\_methodology.html](https://cwe.mitre.org/top25/archive/2023/2023_methodology.html).
- [96] OpenAI. *Privacy policy*. [Accessed: 01-03-2024]. URL: <https://openai.com/policies/privacy-policy>.

# Appendix 1 – Non-Exclusive License for Reproduction and Publication of a Graduation Thesis<sup>1</sup>

I Karl Tamberg

1. Grant Tallinn University of Technology free licence (non-exclusive licence) for my thesis “Harnessing Large Language Models for Software Vulnerability Detection: A Comprehensive Benchmarking Study”, supervised by Hayretdin Bahsi
  - 1.1. to be reproduced for the purposes of preservation and electronic publication of the graduation thesis, incl. to be entered in the digital collection of the library of Tallinn University of Technology until expiry of the term of copyright;
  - 1.2. to be published via the web of Tallinn University of Technology, incl. to be entered in the digital collection of the library of Tallinn University of Technology until expiry of the term of copyright.
2. I am aware that the author also retains the rights specified in clause 1 of the non-exclusive licence.
3. I confirm that granting the non-exclusive licence does not infringe other persons’ intellectual property rights, the rights arising from the Personal Data Protection Act or rights arising from other legislation.

10.05.2024

---

<sup>1</sup>The non-exclusive licence is not valid during the validity of access restriction indicated in the student’s application for restriction on access to the graduation thesis that has been signed by the school’s dean, except in case of the university’s right to reproduce the thesis for preservation purposes only. If a graduation thesis is based on the joint creative activity of two or more persons and the co-author(s) has/have not granted, by the set deadline, the student defending his/her graduation thesis consent to reproduce and publish the graduation thesis in compliance with clauses 1.1 and 1.2 of the non-exclusive licence, the non-exclusive license shall not be valid for the period.

## Appendix 2 – Prompts

### Basic prompt ( $p_b$ ):

```
Human: You are a security researcher, expert in detecting security vulnerabilities.
Provide response only in following format:
vulnerability: <YES or NO> | vulnerability type: <CWE ID> | vulnerability name: <CWE
NAME> | explanation: <short explanation for prediction>.
Use N/A in other fields if there are no vulnerabilities. Do not include anything
else in response.
Is the following code snippet prone to any security vulnerability?
```{code}```
AI: <response>
```

### Basic prompt with recursive criticism and improvement ( $p_{b-rci}$ ):

```
Human: You are a security researcher, expert in detecting security vulnerabilities.
Provide response only in following format:
vulnerability: <YES or NO> | vulnerability type: <CWE ID> | vulnerability name: <CWE
NAME> | explanation: <short explanation for prediction>.
Use N/A in other fields if there are no vulnerabilities. Do not include anything
else in response.
Is the following code snippet prone to any security vulnerability?
```{code}```
AI: <response>
Human: Review your previous answer and find problems with that answer. Feel free to
answer in any format you prefer.
AI: <response>
Human: Based on the problems found, improve your initial answer:
Provide response only in following format:
vulnerability: <YES or NO> | vulnerability type: <CWE ID> | vulnerability name: <CWE
NAME> | explanation: <reason for the initial analysis being correct or wrong>.
Use N/A in other fields if there are no vulnerabilities. Do not include anything
else in response.
AI: <response>
```

### Basic prompt with short recursive criticism and improvement ( $p_{b-srci}$ ):

```
Human: You are a security researcher, expert in detecting security vulnerabilities.
Provide response only in following format:
vulnerability: <YES or NO> | vulnerability type: <CWE ID> | vulnerability name: <CWE
NAME> | explanation: <short explanation for prediction>.
Use N/A in other fields if there are no vulnerabilities. Do not include anything
else in response.
Is the following code snippet prone to any security vulnerability?
```{code}```
AI: <response>
Human: Review your previous answer and find problems with that answer. Based on the
problems found, improve your initial answer:
Provide response only in following format:
vulnerability: <YES or NO> | vulnerability type: <CWE ID> | vulnerability name: <CWE
NAME> | explanation: <reason for the initial analysis being correct or wrong>.
```



Use N/A in other fields if there are no vulnerabilities. Do not include anything else in response.

AI: <response>

### Basic prompt with self-refinement ( $p_{b-sr}$ ):

Human: You are a security researcher, expert in detecting security vulnerabilities. Provide response only in following format:  
vulnerability: <YES or NO> | vulnerability type: <CWE ID> | vulnerability name: <CWE NAME> | explanation: <short explanation for prediction>.  
Use N/A in other fields if there are no vulnerabilities. Do not include anything else in response.

Is the following code snippet prone to any security vulnerability?

```
```{code}```
```

AI: <response>

Human: Act like an expert in cybersecurity. Please provide detailed feedback about the previous response and try to analyze if the response is correct. Feel free to respond in any format you prefer, but try to keep the response concise.

AI: <response>

Human: Based on the problems found, improve your initial answer:

Provide response only in following format:

vulnerability: <YES or NO> | vulnerability type: <CWE ID> | vulnerability name: <CWE NAME> | explanation: <reason for the initial analysis being correct or wrong>.

Use N/A in other fields if there are no vulnerabilities. Do not include anything else in response.

AI: <response>

### Basic prompt with short self-refinement ( $p_{b-ssr}$ ):

Human: You are a security researcher, expert in detecting security vulnerabilities. Provide response only in following format:  
vulnerability: <YES or NO> | vulnerability type: <CWE ID> | vulnerability name: <CWE NAME> | explanation: <short explanation for prediction>.  
Use N/A in other fields if there are no vulnerabilities. Do not include anything else in response.

Is the following code snippet prone to any security vulnerability?

```
```{code}```
```

AI: <response>

Human: Is this analysis correct?

Provide response only in following format:

vulnerability: <YES or NO> | vulnerability type: <CWE ID> | vulnerability name: <CWE NAME> | explanation: <reason for the initial analysis being correct or wrong>.

Use N/A in other fields if there are no vulnerabilities. Do not include anything else in response.

AI: <response>

### API sequence prompt ( $p_{as}$ ):

Human: I want you to act as a vulnerability detection system. I will provide you with the original program and the API call sequence, and you will act upon them.

Provide response only in following format:

vulnerability: <YES or NO> | vulnerability type: <CWE ID> | vulnerability name: <CWE NAME> | explanation: <short explanation for prediction>

...

```
API call sequence: {api-flow-seq}
Is the following program buggy?
```java
{code}
```
AI: <response>
```

### API sequence prompt with RCI ( $p_{as-rci}$ ):

```
Human: I want you to act as a vulnerability detection system. I will provide you with
the original program and the API call sequence, and you will act upon them.
Provide response only in following format:
vulnerability: <YES or NO> | vulnerability type: <CWE ID> | vulnerability name: <CWE
NAME> | explanation: <short explanation for prediction>
...

API call sequence: {api-flow-seq}
Is the following program buggy?
```java
{code}
```
AI: <response>
Human: Review your previous answer and find problems with that answer. Feel free to
answer in any format you prefer.
AI: <response>
Human: Based on the problems found, improve your initial answer:
Provide response only in following format:
vulnerability: <YES or NO> | vulnerability type: <CWE ID> | vulnerability name: <CWE
NAME> | explanation: <reason for the initial analysis being correct or wrong>.
Use N/A in other fields if there are no vulnerabilities. Do not include anything
else in response.
AI: <response>
```

### Require fix prompt ( $p_{rf}$ ):

```
Human: Act as the world's greatest static code analyzer for java programming language. I
will give you a code snippet, and you will analyze it for vulnerabilities.
Give the output in a format: vulnerabilities detected with cwe id's as a numbered
list, and proposed fixes as a separate numbered list. If no vulnerabilities are
found, output: "Code is not vulnerable".
---
{code}
AI: <response>
```

### Require fix prompt with RCI ( $p_{rf-rci}$ ):

```
Human: Act as the world's greatest static code analyzer for java programming language. I
will give you a code snippet, and you will analyze it for vulnerabilities.
Give the output in a format: vulnerabilities detected with cwe id's as a numbered
list, and proposed fixes as a separate numbered list. If no vulnerabilities are
found, output: "Code is not vulnerable".
---
{code}
AI: <response>
Human: Review your previous answer and find problems with that answer. Feel free to
answer in any format you prefer.
```

AI: <response>  
Human: Based on the problems found, improve your initial answer:  
Provide response only in following format:  
vulnerability: <YES or NO> | vulnerability type: <CWE ID> | vulnerability name: <CWE NAME> | explanation: <reason for the initial analysis being correct or wrong>.  
Use N/A in other fields if there are no vulnerabilities. Do not include anything else in response.  
AI: <response>

### Few-shot approach with 20 examples ( $p_{fs20}$ ):

Human: You are a security researcher, expert in detecting security vulnerabilities. You are given some Java code. You must analyse it for vulnerabilities and provide a response only in following format:  
vulnerability: <YES or NO> | vulnerability type: <CWE ID> | explanation: <short explanation for prediction>.  
Use N/A in other fields if there are no vulnerabilities. Do not include anything else in response.  
You are first given a few examples with correct responses and at the end the code that you must analyse.

```

---
Input: ```String script = System.getProperty("SCRIPTNAME");
if (script != null)
System.exec(script);```
Response: vulnerability: YES | vulnerability type: CWE-78 | explanation: If an attacker has control over the "SCRIPTNAME" property, then they could modify the property to point to a dangerous program.
---
Input: ```String script = "ls -la";
System.exec(script);```
Response: vulnerability: NO | vulnerability type: N/A | explanation: N/A
---
Input: ```public static final double price = 20.00;
int quantity = currentUser.getAttribute("quantity");
double total = price * quantity;
chargeUser(total);```
Response: vulnerability: YES | vulnerability type: CWE-20 | explanation: The user has no control over the price variable, however the code does not prevent a negative value from being specified for quantity. If an attacker were to provide a negative value, then the user would have their account credited instead of debited.
---
Input: ```public static final double price = 20.00;
int quantity = 7;
double total = price * quantity;
chargeUser(total);```
Response: vulnerability: NO | vulnerability type: N/A | explanation: N/A
---
Input: ```String filename = System.getProperty("com.domain.application.
dictionaryFile");
File dictionaryFile = new File(filename);```
Response: vulnerability: YES | vulnerability type: CWE-22 | explanation: The path is not validated or modified to prevent it from containing relative or absolute path sequences before creating the File object. This allows anyone who can control the system property to determine what file is used. Ideally, the path should be resolved relative to some kind of application or user home directory.
---
Input: ```String filename = "/home/zeros.txt";
File dictionaryFile = new File(filename);```

```

```

Response: vulnerability: NO | vulnerability type: N/A | explanation: N/A
---
Input: ``IntentFilter filter = new IntentFilter("com.example.URLHandler.openURL");
MyReceiver receiver = new MyReceiver();
registerReceiver(receiver, filter);
...
public class UrlHandlerReceiver extends BroadcastReceiver {{
@Override
public void onReceive(Context context, Intent intent) {{
if("com.example.URLHandler.openURL".equals(intent.getAction())) {{
String URL = intent.getStringExtra("URLToOpen");
int length = URL.length();
}}}}}```
Response: vulnerability: YES | vulnerability type: CWE-476 | explanation: The
application assumes the URL will always be included in the intent. When the URL is
not present, the call to getStringExtra() will return null, thus causing a null
pointer exception when length() is called.
---
Input: ``IntentFilter filter = new IntentFilter("com.example.URLHandler.openURL");
MyReceiver receiver = new MyReceiver();
registerReceiver(receiver, filter);
...
public class UrlHandlerReceiver extends BroadcastReceiver {{
@Override
public void onReceive(Context context, Intent intent) {{
if("com.example.URLHandler.openURL".equals(intent.getAction())) {{
String URL = intent.getStringExtra("URLToOpen");
if (URL != null) {{
int length = URL.length();
}}
}}}}}```
Response: vulnerability: NO | vulnerability type: N/A | explanation: N/A
---
Input: ``DriverManager.getConnection(url, "scott", "tiger");``
Response: vulnerability: YES | vulnerability type: CWE-798 | explanation: This is an
example of an external hard-coded password on the client-side of a connection. This
code will run successfully, but anyone who has access to it will have access to the
password.
---
Input: ``GuardedString pwd = fetchPwdFromSecretStorage();
DriverManager.getConnection(url, "scott", pwd);``
Response: vulnerability: NO | vulnerability type: N/A | explanation: N/A
---
Input: ``String home = System.getProperty("APPHOME");
String cmd = home + INITCMD;
java.lang.Runtime.getRuntime().exec(cmd);``
Response: vulnerability: YES | vulnerability type: CWE-77 | explanation: The code
above allows an attacker to execute arbitrary commands with the elevated privilege
of the application by modifying the system property APPHOME to point to a different
path containing a malicious version of INITCMD. Because the program does not
validate the value read from the environment, if an attacker can control the value
of the system property APPHOME, then they can fool the application into running
malicious code and take control of the system.
---
Input: ``String cmd = "/home/app/" + INITCMD;
java.lang.Runtime.getRuntime().exec(cmd);``
Response: vulnerability: NO | vulnerability type: N/A | explanation: N/A
---
Input: ``try {{

```

```

File file = new File("object.obj");
ObjectInputStream in = new ObjectInputStream(new FileInputStream(file));
javax.swing.JButton button = (javax.swing.JButton) in.readObject();
in.close();
}}'''
Response: vulnerability: YES | vulnerability type: CWE-502 | explanation: This code
does not attempt to verify the source or contents of the file before deserializing
it. An attacker may be able to replace the intended file with a file that contains
arbitrary malicious code which will be executed when the button is pressed.
---
Input: '''try {{
File file = new File("object.obj");
ObjectInputStream in = new ObjectInputStream(new FileInputStream(file));
javax.swing.JButton button = (javax.swing.JButton) validateAndRead(in);
in.close();
}}'''
Response: vulnerability: NO | vulnerability type: N/A | explanation: N/A
---
Input: '''public BankAccount createBankAccount(String accountNumber, String
accountType, String accountName, String accountSSN, double balance) {{
BankAccount account = new BankAccount();
account.setAccountNumber(accountNumber);
account.setAccountType(accountType);
account.setAccountOwnerName(accountName);
account.setAccountOwnerSSN(accountSSN);
account.setBalance(balance);
return account;
}}'''
Response: vulnerability: YES | vulnerability type: CWE-306 | explanation: There is
no authentication mechanism to ensure that the user creating this bank account
object has the authority to create new bank accounts. Some authentication mechanisms
should be used to verify that the user has the authority to create bank account
objects.
---
Input: '''private boolean isUserAuthentic = false;
// authenticate user,
// if user is authenticated then set variable to true
// otherwise set variable to false
public boolean authenticateUser(String username, String password) {{
...
}}
public BankAccount createNewBankAccount(String accountNumber, String accountType,
String accountName, String accountSSN, double balance) {{
BankAccount account = null;
if (isUserAuthentic) {{
account = new BankAccount();
account.setAccountNumber(accountNumber);
account.setAccountType(accountType);
account.setAccountOwnerName(accountName);
account.setAccountOwnerSSN(accountSSN);
account.setBalance(balance);
}}
return account;
}}'''
Response: vulnerability: NO | vulnerability type: N/A | explanation: N/A
---
Input: '''public enum Roles {{
ADMIN,USER,GUEST
}}

```

```

public void printDebugInfo(User requestingUser){
    if(isAuthenticated(requestingUser)){
        switch(requestingUser.role){
            case GUEST:
                System.out.println("You are not authorized to perform this command");
                break;
            default:
                System.out.println(currentDebugState());
                break;
        }
    }
}
else{{
    System.out.println("You must be logged in to perform this command");
}}}}````
Response: vulnerability: YES | vulnerability type: CWE-269 | explanation: While the
intention was to only allow Administrators to print the debug information, the code
as written only excludes those with the role of "GUEST". Someone with the role of "
ADMIN" or "USER" will be allowed access, which goes against the original intent. An
attacker may be able to use this debug information to craft an attack on the system.
---
Input: ``public enum Roles {
ADMIN,USER,GUEST
}
}
public void printDebugInfo(User requestingUser){
    if(isAuthenticated(requestingUser)){
        switch(requestingUser.role){
            case ADMIN:
                System.out.println("You are not authorized to perform this command");
                break;
            default:
                System.out.println(currentDebugState());
                break;
        }
    }
}
else{{
    System.out.println("You must be logged in to perform this command");
}}}}````
Response: vulnerability: NO | vulnerability type: N/A | explanation: N/A
---
Input: ``protected void doPost(HttpServletRequest request, HttpServletResponse
response) throws Exception {
    response.setContentType("text/html");
    PrintWriter out = response.getWriter();
    String contentType = request.getContentType();
    // the starting position of the boundary header
    int ind = contentType.indexOf("boundary=");
    String boundary = contentType.substring(ind+9);
    String pLine = new String();
    String uploadLocation = new String(UPLOAD_DIRECTORY_STRING); //Constant value
    // verify that content type is multipart form data
    if (contentType != null && contentType.indexOf("multipart/form-data") != -1) {{
        // extract the filename from the Http header
        BufferedReader br = new BufferedReader(new InputStreamReader(request.getInputStream()
));
        ...
        pLine = br.readLine();
        String filename = pLine.substring(pLine.lastIndexOf("\\"), pLine.lastIndexOf("\\"));
        ...
        // output the file to the local upload directory

```

```

try {{
BufferedWriter bw = new BufferedWriter(new FileWriter(uploadLocation+filename, true))
;
for (String line; (line=br.readLine())!=null; ) {{
if (line.indexOf(boundary) == -1) {{
bw.write(line);
bw.newLine();
bw.flush();
}}}} //end of for loop
bw.close();
}} catch (IOException ex) {{...}}
// output successful upload response HTML page
}}
// output unsuccessful upload response HTML page
}}````
Response: vulnerability: YES | vulnerability type: CWE-434 | explanation: This code
does not perform a check on the type of the file being uploaded (CWE-434). This
could allow an attacker to upload any executable file or other file with malicious
code.
vulnerability: YES | vulnerability type: CWE-23 | explanation: Since the
code does not check the filename that is provided in the header, an attacker can use
"../" sequences to write to files outside of the intended directory.
---
Input: ````protected void doPost(HttpServletRequest request,HttpServletResponse
response)throws Exception{{
response.setContentType("text/html");
PrintWriter out=response.getWriter();
String contentType=request.getContentType();
// the starting position of the boundary header
int ind=contentType.indexOf("boundary=");
String boundary=contentType.substring(ind+9);
String pLine=new String();
String uploadLocation=new String(UPLOAD_DIRECTORY_STRING); //Constant value
// verify that content type is multipart form data
if (contentType!=null&&contentType.indexOf("multipart/form-data")!=-1){{
// extract the filename from the Http header
BufferedReader br=new BufferedReader(new InputStreamReader(request.getInputStream()))
;
...
pLine=br.readLine();
String filename=pLine.substring(pLine.lastIndexOf("\\\"),pLine.lastIndexOf("\\"));
// Check file type by extension
String fileExtension = getFileExtension(filename);
if (!isValidFileType(fileExtension)) {{
// Output unsuccessful upload response HTML page
return;
}}
// Sanitize filename to prevent path traversal
filename = sanitizeFilename(filename);
...
// output the file to the local upload directory
try{{
BufferedWriter bw=new BufferedWriter(new FileWriter(uploadLocation+filename,true));
for(String line;(line=br.readLine())!=null;){{
if(line.indexOf(boundary)==-1){{
bw.write(line);
bw.newLine();
bw.flush();
}}
}}

```

```

    }} //end of for loop
    bw.close();
    }}catch(IOException ex){{...}}
    // output successful upload response HTML page
    }}
    // output unsuccessful upload response HTML page
    }}'''
    Response: vulnerability: NO | vulnerability type: N/A | explanation: N/A
    ---
    Input: ```{code}```
Response:
AI: <response>

```

### Few-shot approach with six examples ( $p_{fs6}$ ):

Human: You are a security researcher, expert in detecting security vulnerabilities. You are given some Java code. You must analyse it for vulnerabilities and provide a response only in following format:

vulnerability: <YES or NO> | vulnerability type: <CWE ID> | explanation: <short explanation for prediction>.

Use N/A in other fields if there are no vulnerabilities. Do not include anything else in response.

You are first given a few examples with correct responses and at the end the code that you must analyse.

---

```

Input: ```String script = System.getProperty("SCRIPTNAME");
if (script != null)
System.exec(script);```
Response: vulnerability: YES | vulnerability type: CWE-78 | explanation: If an
attacker has control over the "SCRIPTNAME" property, then they could modify the
property to point to a dangerous program.
---
```

```

Input: ```String script = "ls -la";
System.exec(script);```
Response: vulnerability: NO | vulnerability type: N/A | explanation: N/A
---
```

```

Input: ```public static final double price = 20.00;
int quantity = currentUser.getAttribute("quantity");
double total = price * quantity;
chargeUser(total);```
Response: vulnerability: YES | vulnerability type: CWE-20 | explanation: The user
has no control over the price variable, however the code does not prevent a negative
value from being specified for quantity. If an attacker were to provide a negative
value, then the user would have their account credited instead of debited.
---
```

```

Input: ```public static final double price = 20.00;
int quantity = 7;
double total = price * quantity;
chargeUser(total);```
Response: vulnerability: NO | vulnerability type: N/A | explanation: N/A
---
```

```

Input: ```String filename = System.getProperty("com.domain.application.
dictionaryFile");
File dictionaryFile = new File(filename);```
Response: vulnerability: YES | vulnerability type: CWE-22 | explanation: The path is
not validated or modified to prevent it from containing relative or absolute path
sequences before creating the File object. This allows anyone who can control the
system property to determine what file is used. Ideally, the path should be resolved

```



```

    relative to some kind of application or user home directory.
---
Input: ```String filename = "/home/zeros.txt";
File dictionaryFile = new File(filename);```
Response: vulnerability: NO | vulnerability type: N/A | explanation: N/A
---
Input: ```{code}```
Response:
AI: <response>

```

### Few-shot approach with six examples utilising RCI (*p<sub>fs6-rci</sub>*):

```

Human: You are a security researcher, expert in detecting security vulnerabilities.
You are given some Java code. You must analyse it for vulnerabilities and provide a
response only in following format:
    vulnerability: <YES or NO> | vulnerability type: <CWE ID> | explanation: <short
explanation for prediction>.
Use N/A in other fields if there are no vulnerabilities. Do not include anything
else in response.
You are first given a few examples with correct responses and at the end the code
that you must analyse.
---
Input: ```String script = System.getProperty("SCRIPTNAME");
if (script != null)
System.exec(script);```
Response: vulnerability: YES | vulnerability type: CWE-78 | explanation: If an
attacker has control over the "SCRIPTNAME" property, then they could modify the
property to point to a dangerous program.
---
Input: ```String script = "ls -la";
System.exec(script);```
Response: vulnerability: NO | vulnerability type: N/A | explanation: N/A
---
Input: ```public static final double price = 20.00;
int quantity = currentUser.getAttribute("quantity");
double total = price * quantity;
chargeUser(total);```
Response: vulnerability: YES | vulnerability type: CWE-20 | explanation: The user
has no control over the price variable, however the code does not prevent a negative
value from being specified for quantity. If an attacker were to provide a negative
value, then the user would have their account credited instead of debited.
---
Input: ```public static final double price = 20.00;
int quantity = 7;
double total = price * quantity;
chargeUser(total);```
Response: vulnerability: NO | vulnerability type: N/A | explanation: N/A
---
Input: ```String filename = System.getProperty("com.domain.application.
dictionaryFile");
File dictionaryFile = new File(filename);```
Response: vulnerability: YES | vulnerability type: CWE-22 | explanation: The path is
not validated or modified to prevent it from containing relative or absolute path
sequences before creating the File object. This allows anyone who can control the
system property to determine what file is used. Ideally, the path should be resolved
relative to some kind of application or user home directory.
---
Input: ```String filename = "/home/zeros.txt";

```

```

File dictionaryFile = new File(filename);```
Response: vulnerability: NO | vulnerability type: N/A | explanation: N/A
---
Input: ```{code}```
Response:
AI: <response>
Human: Review your previous answer and find problems with that answer. Feel free to
answer in any format you prefer.
AI: <response>
Human: Based on the problems found, improve your initial answer:
Provide response only in following format:
vulnerability: <YES or NO> | vulnerability type: <CWE ID> | vulnerability name: <CWE
NAME> | explanation: <reason for the initial analysis being correct or wrong>.
Use N/A in other fields if there are no vulnerabilities. Do not include anything
else in response.
AI: <response>

```

### Dataflow analysis prompt (*p<sub>dfa</sub>*):

```

Human: You are a security researcher, expert in detecting security vulnerabilities.
Carefully analyze the given code snippet and track the data flows from various
sources to sinks. Assume that any call to an unknown external API is unsanitized.
Please provide a response only in the following format:
Here is a data flow analysis of the given code snippet:
A. Sources: <numbered list of input sources>
B. Sinks: <numbered list of output sinks>
C. Sanitizers: <numbered list of sanitizers, if any>
D. Unsanitized Data Flows: <numbered list of data flows that are not sanitized in
the format (source, sink, why this flow could be vulnerable)>
E. Vulnerability analysis verdict: vulnerability: <YES or NO> | vulnerability type: <
CWE_ID> | vulnerability name: <NAME_OF_CWE> | explanation: <explanation for
prediction>
Is the following code snippet prone to any security vulnerability?
```{code}```
AI: <response>

```

### Dataflow analysis prompt with RCI (*p<sub>dfa-rci</sub>*):

```

Human: You are a security researcher, expert in detecting security vulnerabilities.
Carefully analyze the given code snippet and track the data flows from various
sources to sinks. Assume that any call to an unknown external API is unsanitized.
Please provide a response only in the following format:
Here is a data flow analysis of the given code snippet:
A. Sources: <numbered list of input sources>
B. Sinks: <numbered list of output sinks>
C. Sanitizers: <numbered list of sanitizers, if any>
D. Unsanitized Data Flows: <numbered list of data flows that are not sanitized in
the format (source, sink, why this flow could be vulnerable)>
E. Vulnerability analysis verdict: vulnerability: <YES or NO> | vulnerability type: <
CWE_ID> | vulnerability name: <NAME_OF_CWE> | explanation: <explanation for
prediction>
Is the following code snippet prone to any security vulnerability?
```{code}```
AI: <response>
Human: Review your previous answer and find problems with that answer. Feel free to
answer in any format you prefer.
AI: <response>

```

Human: Based on the problems found, improve your initial answer:  
 Provide response only in following format:  
 vulnerability: <YES or NO> | vulnerability type: <CWE ID> | vulnerability name: <CWE NAME> | explanation: <reason for the initial analysis being correct or wrong>.  
 Use N/A in other fields if there are no vulnerabilities. Do not include anything else in response.  
 AI: <response>

### Dataflow analysis prompt with short RCI (*pdfa-srct*):

Human: You are a security researcher, expert in detecting security vulnerabilities. Carefully analyze the given code snippet and track the data flows from various sources to sinks. Assume that any call to an unknown external API is unsanitized. Please provide a response only in the following format:  
 Here is a data flow analysis of the given code snippet:  
 A. Sources: <numbered list of input sources>  
 B. Sinks: <numbered list of output sinks>  
 C. Sanitizers: <numbered list of sanitizers, if any>  
 D. Unsanitized Data Flows: <numbered list of data flows that are not sanitized in the format (source, sink, why this flow could be vulnerable)>  
 E. Vulnerability analysis verdict: vulnerability: <YES or NO> | vulnerability type: <CWE\_ID> | vulnerability name: <NAME\_OF\_CWE> | explanation: <explanation for prediction>  
 Is the following code snippet prone to any security vulnerability?  
 ```{code}```  
 AI: <response>  
 Human: Review your previous answer and find problems with that answer. Provide feedback in any format you prefer, but respond in 5 sentences or less.  
 AI: <response>  
 Human: Based on the problems found, improve your initial answer:  
 Provide response only in following format:  
 vulnerability: <YES or NO> | vulnerability type: <CWE ID> | vulnerability name: <CWE NAME> | explanation: <reason for the initial analysis being correct or wrong>.  
 Use N/A in other fields if there are no vulnerabilities. Do not include anything else in response.  
 AI: <response>

### Dataflow analysis prompt with extra hints (*pdfa-h*):

Human: You are a security researcher, expert in detecting security vulnerabilities. Carefully analyze the given code snippet and track the data flows from various sources to sinks. Assume that any call to an unknown external API is unsanitized. Please provide a response only in the following format:  
 ""  
 Sources: <numbered list of input sources>  
 Sinks: <numbered list of output sinks>  
 Sanitizers: <numbered list of sanitizers, if any>  
 Unsanitized Data Flows: <numbered list of data flows that are not sanitized in the format (source, sink, why this flow could be vulnerable)>  
 Vulnerability analysis verdict: <numbered list of vulnerabilities, if any, in the format "vulnerability: <YES or NO> | vulnerability type: <CWE\_ID> | vulnerability name: <NAME\_OF\_CWE> | explanation: <explanation for prediction>">  
 ""  
 Is the following code snippet prone to any security vulnerability?  
 ```java  
 {code}  
 ```

AI: <response>

### Dataflow analysis prompt with extra hints with RCI ( $p_{dfa-h-rci}$ ):

Human: You are a security researcher, expert in detecting security vulnerabilities. Carefully analyze the given code snippet and track the data flows from various sources to sinks. Assume that any call to an unknown external API is unsanitized. Please provide a response only in the following format:

```
"""
```

```
Sources: <numbered list of input sources>
```

```
Sinks: <numbered list of output sinks>
```

```
Sanitizers: <numbered list of sanitizers, if any>
```

```
Unsanitized Data Flows: <numbered list of data flows that are not sanitized in the format (source, sink, why this flow could be vulnerable)>
```

```
Vulnerability analysis verdict: <numbered list of vulnerabilities, if any, in the format "vulnerability: <YES or NO> | vulnerability type: <CWE_ID> | vulnerability name: <NAME_OF_CWE> | explanation: <explanation for prediction>">
```

```
"""
```

Is the following code snippet prone to any security vulnerability?

```
```java
{code}
```
```

AI: <response>

Human: Review your previous answer and find problems with that answer. Feel free to answer in any format you prefer.

AI: <response>

Human: Based on the problems found, improve your initial answer:

Provide response only in following format:

```
vulnerability: <YES or NO> | vulnerability type: <CWE ID> | vulnerability name: <CWE NAME> | explanation: <reason for the initial analysis being correct or wrong>.
```

Use N/A in other fields if there are no vulnerabilities. Do not include anything else in response.

AI: <response>

### Dataflow analysis prompt utilising CoT ( $p_{cot-dfa}$ ):

Human: You are a security researcher, expert in detecting security vulnerabilities. Carefully analyze the given code snippet and track the data flows from various sources to sinks. Assume that any call to an unknown external API is unsanitized. The threat model dictates, that the threat actor can only provide malicious values in public function parameters, system properties, environment variables and potentially from external websites. The threat actor might also be able to decompile the code, which could give them access to hardcoded values.

Lets think step-by-step and give the response in the following format:

1. List potential malicious sources: <numbered list of input sources>
  2. List the sinks for those malicious sources: <numbered list of output sinks>
  3. List the sanitizers for those malicious sources: <numbered list of sanitizers, if any>
  4. List other potential issues with the given code: <numbered list of potential issues>
  5. Provide verdict (one line for every potential discovered weakness):  
vulnerability: <YES or NO> | vulnerability type: <CWE\_ID> | vulnerability name: <NAME\_OF\_CWE> | explanation: <explanation for prediction>
- ```
...
```

```
---
```

Is the following code snippet prone to any security vulnerability?

```
```java
{code}
```
AI: <response>
```

### Dataflow analysis prompt utilising CoT with RCI ( $p_{cot-dfa-rci}$ ):

```
Human: You are a security researcher, expert in detecting security vulnerabilities.
Carefully analyze the given code snippet and track the data flows from various
sources to sinks. Assume that any call to an unknown external API is unsanitized.
The threat model dictates, that the threat actor can only provide malicious values
in public function parameters, system properties, environment variables and
potentially from external websites. The threat actor might also be able to decompile
the code, which could give them access to hardcoded values.
Lets think step-by-step and give the response in the following format:
1. List potential malicious sources: <numbered list of input sources>
2. List the sinks for those malicious sources: <numbered list of output sinks>
3. List the sanitizers for those malicious sources: <numbered list of sanitizers, if
any>
4. List other potential issues with the given code: <numbered list of potential
issues>
5. Provide verdict (one line for every potential discovered weakness):
vulnerability: <YES or NO> | vulnerability type: <CWE_ID> | vulnerability name: <
NAME_OF_CWE> | explanation: <explanation for prediction>
...
---
Is the following code snippet prone to any security vulnerability?
```java
{code}
```
AI: <response>
Human: Review your previous answer and find problems with that answer. Feel free to
answer in any format you prefer.
AI: <response>
Human: Based on the problems found, improve your initial answer:
Provide response only in following format:
vulnerability: <YES or NO> | vulnerability type: <CWE ID> | vulnerability name: <CWE
NAME> | explanation: <reason for the initial analysis being correct or wrong>.
Use N/A in other fields if there are no vulnerabilities. Do not include anything
else in response.
AI: <response>
```

### CoT approach evaluating potential problems in eight steps ( $p_{cot-8s}$ ):

```
Human: As a security researcher, you are tasked with reporting security vulnerabilities.
You are currently looking at the following code:
```java
{code}
```
You are expected to work step-by-step:
1. Identify Weaknesses: Identify which vulnerabilities could be present in the
following code.
2. Review User Input Handling: Look for any input sources that are not properly
validated or sanitized before being used in unsafe manner. If variable that is
passed into unsafe function is not directly influenced by external user input, the
vulnerability is not currently present and must not be reported.
```

3. Analyze Data Flow: Trace the flow of untrusted data to the system command. Ensure that there are no points where user-controlled input can directly influence the command execution.
4. Check for Mitigations: Examine if there are any mitigations in place to prevent command injection, such as input validation, sanitization, or using safer alternatives to executing system commands.
5. Evaluate Conditional Branching: If there's dead code (that can never be reached), then this part must not be evaluated.
6. Assess Error Handling: Evaluate how errors, if any, are handled.
7. Identify Code Leaking Secrets: Check whether the code contains secrets that should not be public knowledge.
8. Provide verdict (one line for every potential discovered weakness). Keep in mind you must not report vulnerabilities that cannot be currently abused by malicious actors. False positive results must be kept to minimum. The verdict must be in the format:  
vulnerability: <YES or NO> | vulnerability type: <CWE\_ID> |  
...

AI: <response>

### CoT approach evaluating potential problems in eight steps utilising RCI ( $p_{cot-8s-rci}$ ):

Human: As a security researcher, you are tasked with reporting security vulnerabilities. You are currently looking at the following code:

```
```java
{code}
```
```

You are expected to work step-by-step:

1. Identify Weaknesses: Identify which vulnerabilities could be present in the following code.
2. Review User Input Handling: Look for any input sources that are not properly validated or sanitized before being used in unsafe manner. If variable that is passed into unsafe function is not directly influenced by external user input, the vulnerability is not currently present and must not be reported.
3. Analyze Data Flow: Trace the flow of untrusted data to the system command. Ensure that there are no points where user-controlled input can directly influence the command execution.
4. Check for Mitigations: Examine if there are any mitigations in place to prevent command injection, such as input validation, sanitization, or using safer alternatives to executing system commands.
5. Evaluate Conditional Branching: If there's dead code (that can never be reached), then this part must not be evaluated.
6. Assess Error Handling: Evaluate how errors, if any, are handled.
7. Identify Code Leaking Secrets: Check whether the code contains secrets that should not be public knowledge.
8. Provide verdict (one line for every potential discovered weakness). Keep in mind you must not report vulnerabilities that cannot be currently abused by malicious actors. False positive results must be kept to minimum. The verdict must be in the format:  
vulnerability: <YES or NO> | vulnerability type: <CWE\_ID> |  
...

AI: <response>

Human: Review your previous answer and find problems with that answer. Feel free to answer in any format you prefer.

AI: <response>

Human: Based on the problems found, improve your initial answer:  
Provide response only in following format:  
vulnerability: <YES or NO> | vulnerability type: <CWE ID> | vulnerability name: <CWE NAME> | explanation: <reason for the initial analysis being correct or wrong>.

Use N/A in other fields if there are no vulnerabilities. Do not include anything else in response.

AI: <response>

### Code review prompt ( $p_{cr}$ ) with checklist, which was taken from [89]:

Human: You are responsible for the security of a codebase. You are given a checklist containing questions and a Java code file.

Based on the contents of the Java file, you must answer all the questions. The answers to the questions can be in any format, but must be short (1-2 sentences).

Code:

```
```java
{code}
```
```

Checklist:

- \* What security vulnerabilities is this code susceptible to?
- \* Are authorization and authentication handled in the right way?
- \* Is (user) input validated, sanitized, and escaped to prevent cross-site scripting or SQL injection?
- \* Is sensitive data like user data, or credit card information securely handled and stored?
- \* Does this code NOT reveal some secret information like keys, passwords, or usernames?
- \* Is data retrieved from external APIs or libraries checked accordingly?
- \* Does error handling or logging NOT expose the system to vulnerabilities?
- \* Is the right encryption used?
- \* Is sensitive data encrypted in storage and in transit?
- \* Would having access to the codebase affect the security of the application (does code contain hard-coded secrets)?

Based on the answers given, you must provide a final verdict (one line for every potential discovered weakness). The final verdict must be in the format:

vulnerability: <YES or NO> | vulnerability type: <CWE\_ID> | <short one sentence explanation>

If no vulnerabilities are present, respond with: "Not vulnerable"

AI: <response>

### Code review prompt utilising RCI ( $p_{cr-rci}$ ) with checklist, which was taken from [89]:

Human: You are responsible for the security of a codebase. You are given a checklist containing questions and a Java code file.

Based on the contents of the Java file, you must answer all the questions. The answers to the questions can be in any format, but must be short (1-2 sentences).

Code:

```
```java
{code}
```
```

Checklist:

- \* What security vulnerabilities is this code susceptible to?
- \* Are authorization and authentication handled in the right way?
- \* Is (user) input validated, sanitized, and escaped to prevent cross-site scripting or SQL injection?

- \* Is sensitive data like user data, or credit card information securely handled and stored?
- \* Does this code NOT reveal some secret information like keys, passwords, or usernames?
- \* Is data retrieved from external APIs or libraries checked accordingly?
- \* Does error handling or logging NOT expose the system to vulnerabilities?
- \* Is the right encryption used?
- \* Is sensitive data encrypted in storage and in transit?
- \* Would having access to the codebase affect the security of the application (does code contain hard-coded secrets)?

Based on the answers given, you must provide a final verdict (one line for every potential discovered weakness). The final verdict must be in the format:  
 vulnerability: <YES or NO> | vulnerability type: <CWE\_ID> | <short one sentence explanation>

If no vulnerabilities are present, respond with: "Not vulnerable"

AI: <response>

Human: Review your previous answer and find problems with that answer. Feel free to answer in any format you prefer.

AI: <response>

Human: Based on the problems found, improve your initial answer:

Provide response only in following format:

vulnerability: <YES or NO> | vulnerability type: <CWE ID> | vulnerability name: <CWE NAME> | explanation: <reason for the initial analysis being correct or wrong>.

Use N/A in other fields if there are no vulnerabilities. Do not include anything else in response.

AI: <response>

### Tree of thoughts 8-step prompt ( $p_{tot-8s}$ ):

Human: As a security researcher, you must analyse the given code and respond to the step number {step\_nr}. If there are previous steps, the answers to them have already been provided and must be considered accurate. Keep your responses short and concise. You are currently looking at the following code:

```
```java
{input}
```
```

Steps:

1. Identify All Weaknesses: Identify which vulnerabilities could be present in the following code.
2. Review User Input Handling: Look for any input sources that are not properly validated or sanitized before being used in unsafe manner. If variable that is passed into unsafe function is not directly influenced by external user input, the vulnerability is not currently present and must not be reported.
3. Analyze Data Flow: Trace the flow of untrusted data to the system command. Ensure that there are no points where user-controlled input can directly influence the command execution.
4. Check for Mitigations: Examine if there are any mitigations in place to prevent command injection, such as input validation, sanitization, or using safer alternatives to executing system commands.
5. Evaluate Conditional Branching: If there's dead code (that can never be reached), then this part must not be evaluated.
6. Assess Error Handling: Evaluate how errors, if any, are handled.
7. Identify Code Leaking Secrets: Check whether the code contains secrets that should not be public knowledge.
8. Provide verdict (one line for every potential discovered weakness). Keep in mind you must not report vulnerabilities that cannot be currently abused by malicious actors. False positive results must be kept to minimum. The response for the last



```
step (verdict) must be in the format:  
vulnerability: <YES or NO> | vulnerability type: <CWE_ID> |  
...  
AI: <response>
```

### Tree of thoughts 8-step evaluation prompt ( $p_{tot-8s-eval}$ ):

```
Human: Given a code snippet and several choices for the analysis of this code,  
decide which choice is best and most accurate. Code:  
``java  
{input}  
``  
Analyze each choice in detail, then conclude in the last line "The best choice is {{  
s}}", where s the integer id of the choice.  
Choice 1:  
<choice-1>  
Choice 2:  
<choice-2>  
Choice 3:  
<choice-3>  
AI: <response>
```

## Appendix 3 – Code Samples

### Excerpt of J20736.java file, vulnerable to CWE-78:

```
8 public void handle(HttpServletRequest request, HttpServletResponse response) throws
  Throwable {
9     String data;
10    if (PRIVATE_STATIC_FINAL_TRUE) {
11        data = "";
12        {
13            StringTokenizer tokenizer = new StringTokenizer(request.getQueryString(),
14                "&");
15            while (tokenizer.hasMoreTokens()) {
16                String token = tokenizer.nextToken();
17                if (token.startsWith("id=")) {
18                    data = token.substring(3);
19                    break;
20                }
21            }
22        } else {
23            data = null;
24        }
25        String osCommand;
26        if (System.getProperty("os.name").toLowerCase().indexOf("win") >= 0) {
27            osCommand = "c:\\WINDOWS\\SYSTEM32\\cmd.exe /c dir ";
28        } else {
29            osCommand = "/bin/ls ";
30        }
31        Process process = Runtime.getRuntime().exec(osCommand + data);
32        process.waitFor();
33    }
```

### Excerpt of J23877.java file, not vulnerable to CWE-89:

```
7 private void processG2B(HttpServletRequest request, HttpServletResponse response) throws
  Throwable {
8     String data;
9     if (IO.staticReturnsTrueOrFalse()) {
10        data = "foo";
11    } else {
12        data = "foo";
13    }
14    if (IO.staticReturnsTrueOrFalse()) {
15        if (data != null) {
16            String names[] = data.split("-");
17            int successCount = 0;
18            Connection dbConnection = null;
19            Statement sqlStatement = null;
20            try {
21                dbConnection = IO.getDBConnection();
22                sqlStatement = dbConnection.createStatement();
23                for (int i = 0; i < names.length; i++) {
```

```

24         sqlStatement.addBatch("update users set hitcount=hitcount+1 where
name=' " + names[i] + "'");
25     }
26     int resultsArray[] = sqlStatement.executeBatch();
27     for (int i = 0; i < names.length; i++) {
28         if (resultsArray[i] > 0) {
29             successCount++;
30         }
31     }
32     IO.writeLine("Succeeded in " + successCount + " out of " + names.length +
" queries.");
33     } catch (SQLException exceptSql) {
34         IO.logger.log(Level.WARNING, "Error getting database connection",
exceptSql);
35     } finally {
36         try {
37             if (sqlStatement != null) {
38                 sqlStatement.close();
39             }
40         } catch (SQLException exceptSql) {
41             IO.logger.log(Level.WARNING, "Error closing Statement", exceptSql);
42         }
43         try {
44             if (dbConnection != null) {
45                 dbConnection.close();
46             }
47         } catch (SQLException exceptSql) {
48             IO.logger.log(Level.WARNING, "Error closing Connection", exceptSql);
49         }
50     }
51 }
52 } else {
53     if (data != null) {
54         String names[] = data.split("-");
55         int successCount = 0;
56         Connection dbConnection = null;
57         Statement sqlStatement = null;
58         try {
59             dbConnection = IO.getDBConnection();
60             sqlStatement = dbConnection.createStatement();
61             for (int i = 0; i < names.length; i++) {
62                 sqlStatement.addBatch("update users set hitcount=hitcount+1 where
name=' " + names[i] + "'");
63             }
64             int resultsArray[] = sqlStatement.executeBatch();
65             for (int i = 0; i < names.length; i++) {
66                 if (resultsArray[i] > 0) {
67                     successCount++;
68                 }
69             }
70             IO.writeLine("Succeeded in " + successCount + " out of " + names.length +
" queries.");
71         } catch (SQLException exceptSql) {
72             IO.logger.log(Level.WARNING, "Error getting database connection",
exceptSql);
73         } finally {
74             try {
75                 if (sqlStatement != null) {
76                     sqlStatement.close();

```

```

77         }
78     } catch (SQLException exceptSql) {
79         IO.logger.log(Level.WARNING, "Error closing Statement", exceptSql);
80     }
81     try {
82         if (dbConnection != null) {
83             dbConnection.close();
84         }
85     } catch (SQLException exceptSql) {
86         IO.logger.log(Level.WARNING, "Error closing Connection", exceptSql);
87     }
88 }
89 }
90 }
91 }
92 private void processB2G(HttpServletRequest request, HttpServletResponse response) throws
    Throwable {
93     String data;
94     if (IO.staticReturnsTrueOrFalse()) {
95         data = request.getParameter("name");
96     } else {
97         data = request.getParameter("name");
98     }
99     if (IO.staticReturnsTrueOrFalse()) {
100         if (data != null) {
101             String names[] = data.split("-");
102             int successCount = 0;
103             Connection dbConnection = null;
104             PreparedStatement sqlStatement = null;
105             try {
106                 dbConnection = IO.getDBConnection();
107                 sqlStatement = dbConnection.prepareStatement("update users set hitcount=
hitcount+1 where name=?");
108                 for (int i = 0; i < names.length; i++) {
109                     sqlStatement.setString(1, names[i]);
110                     sqlStatement.addBatch();
111                 }
112                 int resultsArray[] = sqlStatement.executeBatch();
113                 for (int i = 0; i < names.length; i++) {
114                     if (resultsArray[i] > 0) {
115                         successCount++;
116                     }
117                 }
118                 IO.writeLine("Succeeded in " + successCount + " out of " + names.length +
" queries.");
119             } catch (SQLException exceptSql) {
120                 IO.logger.log(Level.WARNING, "Error getting database connection",
exceptSql);
121             } finally {
122                 try {
123                     if (sqlStatement != null) {
124                         sqlStatement.close();
125                     }
126                 } catch (SQLException exceptSql) {
127                     IO.logger.log(Level.WARNING, "Error closing PreparedStatement",
exceptSql);
128                 }
129                 try {
130                     if (dbConnection != null) {

```

```

131         dbConnection.close();
132     }
133     } catch (SQLException exceptSql) {
134         IO.logger.log(Level.WARNING, "Error closing Connection", exceptSql);
135     }
136 }
137 }
138 } else {
139     if (data != null) {
140         String names[] = data.split("-");
141         int successCount = 0;
142         Connection dbConnection = null;
143         PreparedStatement sqlStatement = null;
144         try {
145             dbConnection = IO.getDBConnection();
146             sqlStatement = dbConnection.prepareStatement("update users set hitcount=
hitcount+1 where name=?");
147             for (int i = 0; i < names.length; i++) {
148                 sqlStatement.setString(1, names[i]);
149                 sqlStatement.addBatch();
150             }
151             int resultsArray[] = sqlStatement.executeBatch();
152             for (int i = 0; i < names.length; i++) {
153                 if (resultsArray[i] > 0) {
154                     successCount++;
155                 }
156             }
157             IO.writeLine("Succeeded in " + successCount + " out of " + names.length +
" queries.");
158         } catch (SQLException exceptSql) {
159             IO.logger.log(Level.WARNING, "Error getting database connection",
exceptSql);
160         } finally {
161             try {
162                 if (sqlStatement != null) {
163                     sqlStatement.close();
164                 }
165             } catch (SQLException exceptSql) {
166                 IO.logger.log(Level.WARNING, "Error closing PreparedStatement",
exceptSql);
167             }
168             try {
169                 if (dbConnection != null) {
170                     dbConnection.close();
171                 }
172             } catch (SQLException exceptSql) {
173                 IO.logger.log(Level.WARNING, "Error closing Connection", exceptSql);
174             }
175         }
176     }
177 }
178 }
179 public void process(HttpServletRequest request, HttpServletResponse response) throws
Throwable {
180     processG2B(request, response);
181     processB2G(request, response);
182 }

```

## Appendix 4 – Versions

**The following versions of different software are used in the study:**

1. CodeQL: The CLI toolchain release 2.16.0 is used.
2. SpotBugs: The Gradle plugin version 6.0.7 is used.
3. FindSecurityBugs: The Find Sec Bugs plugin version 1.12.0 is used.
4. GPT-4 turbo: The model gpt-4-0125-preview is used.
5. GPT-4: The model gpt-4-0613 is used.
6. Claude 3 Opus: The model claude-3-opus-20240229 is used.
7. Juliet dataset: The Java Juliet 1.3 dataset is used.