School of Physics, Mathematics and Computing
Department of Computer Science and Software Engineering

University of Western Australia

Honours Project Dissertation

Integrating Static Analysers and Large Language Models for Code Vulnerability Detection

Igor Pavkov (23193276)

Supervised by

Dr. Jin Hong
Dr. Tingting Bi

14 October 2024

Abstract

As the world becomes more and more dependent on software, the potential risk of security vulnerabilities increases. Vulnerabilities can be found through manual code review as well as with the help of automated tools such as static and dynamic analysers. However, these tools have been shown to perform poorly in practice, often struggling to find vulnerabilities in real-world situations[1], [2].

Large language models (LLMs) such as OpenAI's ChatGPT[3] are advanced word-predicting systems that can generate responses to user input queries. They have seen use in a wide range of fields including in coding applications[4]. Studies investigating the use of LLMs as vulnerability detectors have shown mixed results, with LLMs often reporting high rates of false positive detections[5], [6].

To address these performance concerns, we introduced two detection approaches that combine static tools and LLMs in order to improve performance and decrease false positive rates. The first approach integrates static tool results directly into LLM prompts while the second utilises an ensemble of static tools and LLMs with a range of voting strategies being tested. These combined approaches were evaluated by benchmarking them on a common dataset against individual tools and LLMs. We found that both combined approaches were successful in reducing false positive rates, although there were trade-offs to accuracy and recall. Our research showed that there is potential in combining static tools and LLMs for vulnerability detection.

The secondary goal of our research was to experiment with static tool configurations and a range of LLM prompt variations in order to maximise their individual performances and provide the fairest comparison to combined approaches. While static tools performed best in their default configuration, LLM performance was significantly improved by techniques such as chain-of-thought and emotive prompting.

Contents

List of Tables

List of Figures

CHAPTER 1

Introduction

1.1 Background

1.1.1 Vulnerabilities

A code security vulnerability is a flaw in a piece of code that opens it up to being exploited by an attacker[7]. In applications where sensitive data is being handled, it is essential to maintain code security. Ideally, security vulnerabilities should be caught as early as possible to minimise the potential for harm and decrease the work required to repair them[8].

Types of vulnerabilities are documented by the MITRE Corporation in their Common Weakness Enumeration (CWE) list, last updated in February 2024[9]. Each CWE entry contains fields describing the weakness, giving alternate terms for the same weakness and defining relationships to other weaknesses as well as the categories the weakness belongs to. Groups of CWEs are also assigned separate CWE-IDs in a hierarchical relationship. For example, Incorrect Calculation, or CWE-689 is parent to several CWEs including the off-by-one error, CWE-193. Every year, the MITRE Corporation ranks the Top 25 Most Dangerous Software Weaknesses[10]. This list is useful for determining which vulnerability types should be given the most attention during code checking. When a specific instance of a vulnerability type is found, it is given a unique identifier and added to the Common Vulnerabilities and Exposures (CVE) list, also controlled by the Mitre Corporation[11]. Each of these CVE entries can be associated with a vulnerability type on the CWE list[7].

1.1.2 Static and Dynamic Analysis

Static analysers scan through source code without running it and report potential issues. There are two types of static analysis techniques[2], [12]. Syntactic static analysis matches parts of the source code with patterns that are

known to potentially be vulnerable while semantic static analysis constructs an abstracted model of the program to analyse for vulnerabilities. The semantic approach is able to take information such as control flow into account while the syntactic approach cannot. Although some static analysers only cover style guidelines or type checking, others are designed to detect security vulnerabilities[1].

Dynamic analysers run code with test inputs and examine the output to determine if any issues are present. Due to this, they often do not require access to source code to function[13], making them the only option when source code is not obtainable. To maximise code coverage, dynamic analysers run a program with varied inputs that result in different control flow paths. This process of running a program with a diverse range of inputs is known as fuzzing[13], and can involve random inputs (black fuzzing) or inputs that consider the program context (Graybox and Whitebox fuzzing).

Each tool type has its own advantages and disadvantages. As static analysis does not require the whole program to be in an executable state, it can be used to catch vulnerabilities earlier in development than dynamic analysis[1]. Static analysers also offer full code coverage, as they scan the entirety of the source code. This means that vulnerabilities can be detected in any part of the code, including lines that are rarely run in practice. While this allows a potentially greater number of vulnerabilities to be detected, it also increases the chance of false positive results. False positives are especially common for syntactic static analysers[14], as program flow is not considered during analysis. Even if a code segment is vulnerable given a certain input, the analyser has no way of knowing if it is possible for the input to reach the code during execution. As a result, these cases are flagged as vulnerabilities even if they pose no risk in a real execution. In contrast, dynamic analysers may not cover all control flow possibilities depending on the program inputs they use. This increases the change that vulnerabilities are missed, but greatly decreases the rate of false positives, resulting in higher precision[13].

1.1.3 Large Language Models

To be able to generate coherent responses, LLMs must be trained on examples of coherent text. LLMs such as ChatGPT learn in a two step approach of pre-training followed by fine-tuning[15]. Pre-training is an unsupervised form of learning, meaning that training data is not labelled. In this phase, the LLM learns patterns in the training data to predict words in a sequence. Test data being unlabelled means that it can be retrieved from a large variety of sources such as books and web sites, allowing pre-trained LLMs to per-

form well over a wide range of domains. Fine-tuning is a supervised learning process that can be used to specialise an LLM for a particular task. It involves training the LLM again on a smaller, labelled dataset of task-specific examples to optimise model parameters.

As the pre-training process is unsupervised, LLMs have no way of discerning the quality or correctness of their training data and will treat all data equally. If an LLM is trained on data that is incorrect, outdated or biased, generated responses may reflect these undesirable qualities[15]. For example, it has been shown that code generated by ChatGPT is prone to containing security vulnerabilities[4]. A likely explanation for this is that a large number of code examples in ChatGPT's training data did not properly follow secure coding practices, so generated code reflects this insecurity.

By providing an LLM with source code and prompting it to check for vulnerabilities, the LLM can effectively be used as a static analysis tool[8]. LLMs used as vulnerability detection tools offer a few key advantages over conventional static and dynamic analysers. Most static and dynamic tools need to parse a code segment before analysing it, which requires that the segment is syntactically valid[8]. LLMs are able to review code snippets of any length regardless of whether the code can be compiled or run. This allows for code analysis earlier in the software development process, potentially resulting in vulnerabilities being caught earlier.

Static and dynamic tools are limited in the range of languages they support[1], [13]. In addition, these tools are often specialised to deal with one type of vulnerability, such as memory-related vulnerabilities[13]. For a larger project, especially those written in multiple languages, many tools will have to be used in combination to provide suitable coverage of the source code. In contrast, the range of vulnerability types and languages that an LLM can detect is only limited by the LLM's training data. This means that in practice, LLMs are capable of detecting a wide range of vulnerability types in many commonly used languages[16].

1.2 Problem Statement

As the scale of a coding project grows, it becomes infeasible to manually find every potential security vulnerability in a code-base. Vulnerability detection tools such as static analysers are an important part of the software development process, as they are able to automatically flag potentially vulnerable code segments for manual review. However, the poor performance

seen from static tools in real-world scenarios[1], [2] increases the risk of vulnerable code being missed. Recently, LLMs have been applied to the task of vulnerability detection. While some studies have shown LLMs to offer exceptional detection performance exceeding that of static tools[8], [16], others have found that LLMs struggle to detect vulnerabilities in practice and report high rates of false positives[5], [6]. Due to the poor performance of current methods, there is a clear need for a better method of automatic security vulnerability detection.

In our research, we investigated two approaches for combining LLMs with static analysis tools in order to improve detection performance and reduce false positive rates. The first approach involved integrating static tool results into LLM prompts while the second approach utilised an ensemble of predictors, where static tools and LLMs contribute to a single overall prediction. To determine the effectiveness of these combined approaches, we conducted a performance benchmark, testing a range of static tool configurations and LLM prompts to give a baseline for detection performance. Among the LLMs tested was o1-mini[17], OpenAI's newest model. This benchmarking also allowed us to optimise the performance of the tested static tools and LLMs. We considered the following research questions:

- **RQ1**: Can static tool flags be optimised to improve vulnerability detection performance?
- **RQ2**: Does o1-mini provide better vulnerability detection performance than its predecessors?
- **RQ3**: Can LLM vulnerability detection performance be improved through prompt variations?
- **RQ4**: Do our combined approaches provide better performance than static tools and LLMs individually?

CHAPTER 2

Literature Review

2.1 Static and Dynamic Tool Comparisons

Kaur and Nayyar[1] compared the performance of C and Java static analysis tools and found that among both languages, the C static analyser RATS had the best detection ratio at 84 out of 118 vulnerabilities detected. Flawfinder and Cppcheck lagged behind with 52 and 59 vulnerabilities detected respectively. However, The result breakdown by CWE type shows that each tool covered different ranges of vulnerabilities. For example, only Flawfinder detected the use of a broken or risky cryptographic algorithm (CWE-327), while only Cppcheck detected dead code (CWE-561). Therefore, it was concluded that each tool had its own benefits and drawbacks and the best tool to use depended on the usage context.

Lipp et al.[2] similarly compared the vulnerability detection performance of a range of free, open-source static analysers for the C language including Flawfinder, Cppcheck, Infer, CodeChecker and CodeQL as well as an anonymous commercial tool. Four detection scenarios were considered for results, each requiring different criteria for a vulnerability to be considered as detected. The criteria related to how many vulnerable functions needed to identified in each example as well as whether the correct CWE had to be identified for the detection to count. While the commercial tool outperformed the free tools, even it struggled to detect vulnerabilities, with a detection rate of 53% in the most lenient detection scenario. Interestingly, the syntactic analysers were able to match the performance of the semantic analysers in many cases despite the former not taking control flow into account.

The paper sought to address two observed limitations of existing tool benchmark papers. First, many existing papers exclusively use synthetic datasets with vulnerable code snippets being injected into secure code. These kinds of examples are not necessarily representative of vulnerabilities in real-world programs, so tool performance on synthetic data may not reflect real-world

tool performance. To address this issue, the researchers tested the tools on the Magma benchmark dataset, which is composed of real CVE reports, as well as 19 programs with documented vulnerabilities. Their concerns were validated as the tools performed poorly on the real-world examples, suggesting that synthetic benchmarks are not an accurate indicator of tool performance on real-world programs.

To address the second observed limitation that other papers did not perform result analyses separated by vulnerability type, the researchers performed this result separation. Their results showed that even the CWEs that were supported by most of the tools were missed roughly half the time in practice. The least-detected CWEs, Insufficient Control Flow Management (CWE-691) and Improper Neutralisation (CWE-707), were only detected 14% and 8% of the time respectively despite the fact that four of the tested tools supported these types. These conclusions support the idea that theoretical performance metrics of static analysers do not translate to real-world performance.

Zaazaa and El Bakkali [13] compared various tools implementing a range of dynamic analysis techniques including taint analysis, fuzzing and symbolic execution. To compare performance between tools, results from other papers were discussed. It was noted that dynamic tools generally had a high precision with low false positive rate but were limited by low code coverage. However tools such as KLEE were able to break this trend, achieving an average of 81.9% code coverage on a dataset of 90 tools from GNU Coreutils. In terms of detection rate, the web analyser Arachni performed especially well with a vulnerability detection rate of 96% on its test dataset. In contrast to static tools, the examined dynamic tools specialised in small subsets of vulnerabilities. For example, the tool AEG was only able to detect buffer overflows.

These papers reveal a common trend that different analysis tools are often better suited for detecting different vulnerability types [1], [2], [13], suggesting that a combination of tools would be the best option if a wide coverage of vulnerability types is required. Various papers have considered the use of combinations of tools for improving detection performance, discussed in Section 2.3.

2.2 Code Analysis Performance of LLMs

2.2.1 Performance Comparison Between LLMs

Some LLMs such as Meta's CodeLlama[18] are fine-tuned for optimal performance in coding tasks. While coding-specialised LLMs would be expected to perform better than general LLMs in these tasks, the size of an LLM also affects its performance. GPT4 is one of the largest publicly available LLMs at over 1.7 trillion hyperparameters[19] so would likely perform better than a smaller LLM at the same task if all other factors were kept the same. Multiple papers have compared the performance of both general and code-specialised LLMs for vulnerability detection[6], [19], with conflicting results on which LLM trait is more significant.

Fu et al.[19] compared the vulnerability detection and repairing performance of GPT3.5 and GPT4 to the LLMs CodeBERT and GraphCodeBERT, which are fine-tuned for coding tasks. Each tool was tested on a dataset of 188,000 C and C++ functions containing 91 vulnerability types. Although it outperformed GPT3.5, even GPT4 struggled with vulnerability detection, achieving an F1 score of 29% and a multiclass accuracy of 20%. In comparison, CodeBERT was able to achieve an F1 score of 94% with a multiclass accuracy of 63%. The coding LLMs were also able to generate repair patches, with GraphCodeBERT and CodeBERT repairing 9% and 7% of examples respectively, while ChatGPT was incapable of repairing code. The better performance of the coding LLMs was despite the fact that GPT4 in particular was larger by a factor of 14,000, highlighting the importance of fine-tuning for LLM performance in coding tasks.

Ullah et al.[6] conducted a comprehensive review of vulnerability detection performance involving eight LLMs. The LLMs were tuned with temperature values of 0.0 in order to minimise inconsistency of results due to non-determinism. GPT4 was the best performing LLM with a maximum accuracy of 89.5%, outperforming the coding-specialised LLM CodeLlama. However, every LLM showed a high false positive rate on real-world test cases. It was also found that LLMs struggle with correctly identifying vulnerabilities in augmented code, with simple variable name changes causing incorrect detections in some cases. This is concerning for the detection ability of LLMs as an analysis tool would be expected to perform similarly for similar inputs. The observed inconsistency puts doubt on the applicability of results from LLM benchmark papers to real-world situations, as trivial code changes in their testing datasets may have influenced results. A similar issue was noticed with LLMs over-focusing on superficial details like function names, flagging

commonly used functions such as strcat even if there are implemented securely. LLM seeding, which is supported by GPT-4, was noted as a potential solution to non-deterministic inconsistency.

2.2.2 LLMs vs Vulnerability Detection Tools

Other papers have shown promising results for LLM performance in comparison to static tools. Ozturk et al. [8] compared the performance of ChatGPT to 11 static analysers for PHP on a dataset of 92 manually written vulnerable code examples using vulnerability categories from the OWASP Top 10 2021 list. Role-based prompting was used as ChatGPT was asked to act as a cybersecurity specialist when analysing code examples. Each prompt was also run twice to minimise the effects of non-deterministic LLM output. It was found that ChatGPT significantly outperformed all static tools tested. Even when multiple static tools were run together, ChatGPT had the highest vulnerability detection rate at 68% compared to the best-performing static tool combination, which was Psalm and SonareQube CE at 53%. However, ChatGPT's higher detection rate came with a cost of an extremely high false positive rate of 91%, while the previously-mentioned static tool combination had a much lower false positive rate of 55%.

Purba et al.[16] compared the performance of the LLMs Davinci and CodeGen to the static analysers Flawfinder, RATS and Checkmarx as well as the deep learning tool VulDeePecker. Their results show that the LLMs performed better than all tested static analysis tools in terms of F1 score, mainly due to their much lower false negative rate. Codegen and Davinci had false negative rates of 32% and 6% respectively while Checkmarx had the lowest false negative rate of the static analysers at 41.1%. However, all LLMs had a higher false positive rate than any of the other tools tested, harming their overall performance. Notably, VulDeePecker performed the best by a sizeable margin, achieving an F1 score of 90.5% with an extremely low false positive and false negative rates of 5.7% and 7.0% respectively. The paper's use of a balanced dataset containing both vulnerable and non-vulnerable code examples increases the reliability of the false positive rate results. Additionally, it was found that LLMs had higher vulnerability detection rates on the Code Gadgets dataset, which contained shorter code snippets. The researchers hypothesised that giving LLMs code snippets allowed the LLMs to focus on areas that contained vulnerable code patterns. As a result of this observation, the suggestion was made to slice source code into snippets before feeding them into LLMs to allow for potentially improved LLM performance. However, the LLMs still recorded a similarly high rate of false positives on

the Code Gadgets dataset, so the slicing approach is unlikely to help reduce false positive rate.

LLMs can also be used in combination with conventional analysis tools to aid in vulnerability detection. Li et al. [20] investigated the potential of ChatGPT in pruning false positive results. For each example, ChatGPT was prompted to summarise the code rather than being directly prompted to check if vulnerabilities were present. This avoids biasing the output in favour of positive identifications. It was found that GPT4 outperformed GPT3.5, and was able to identify 80% of false positive examples. The success of GPT4 in this task is promising for the use of LLMs in false positive pruning. However as a pilot study, the paper is fairly narrow in scope, focusing on the performance of two versions of ChatGPT at detecting a single vulnerability type in one language. Additionally, the testing dataset was small, only comprising of 20 examples. This was because all examples were checked to ensure they were false positives, and had to be inputted into the LLMs manually as the GPT4 API was not available at the time of the study. Despite these limitations, the paper demonstrated that the use case of LLMs in false positive pruning has potential and warrants further investigation. If shown to be effective on a larger scale, results from existing tools could be run through an LLM to increase their effective precision. Static analysis tools would especially benefit from this approach due to their high false positive rates compared to dynamic tools. In addition, the same idea can be applied to LLMs acting as static analysers, allowing them to check their own output and reduce their high false positive rate.

2.2.3 Response Quality

One consideration of using LLMs as static analysers is the quality of response that is produced. While conventional static tools have a consistent response format, the non-deterministic nature of LLMs means that responses produced will differ each time they are run. Yu et al. [5] investigated LLM response quality for code analysis and found that responses were often poor judging by metrics such as correctness, understandability and conciseness. For the best performing LLM, GPT4, only 14.45% of responses were considered helpful or instrumental. Even in cases where the LLM identified a vulnerability, it often failed to correctly name or describe actual vulnerability that was present in the example. It was found that in cases where response quality was poor, the LLM was not considering every part of the prompt and was instead generating generic text that was not directly relevant to the test code. Ullah et al.[6] also found that LLMs sometimes gave incorrect explan-

ations even if the correct vulnerability type was identified. These findings show that further development is needed before LLM response quality can rival that of conventional tools.

2.2.4 Prompt Choice

The quality of responses produced by LLMs is highly dependent on the inputted prompt. Even small changes to a prompt's wording can significantly change an LLM's output. Multiple papers have investigated the impact of prompt choice on response quality in coding applications[6], [21].

Zhang et al [21] considered the design of prompts to improve the vulnerability detecting capabilities of ChatGPT. Their dataset contained Java and C/C++ code examples covering 50 CWEs. The first approach was a basic prompt asking the LLM to answer if an inputted code example was buggy, with prompt variations including a role-based variation, where the LLM was asked to act as a vulnerability detecting system and a reverse variation, where the LLM was instead asked if the inputted code was correct. The second approach investigated the effect of adding auxiliary information to prompts, such as that relating to the control and data flow of the program. The third approach involved the use of multiple consecutive prompts, from the basic, role-based and auxiliary types described above. This is known as chain-of-thought prompting and is intended to mimic a reasoning process.

Overall, ChatGPT was able to outperform the baseline tools CFGNN and Bugram, which used graph neural networks and N-gram language models respectively. ChatGPT performed significantly better on the Java dataset, with a maximum accuracy of 74.7% compared to a maximum of 52.5% for the C/C++ examples, showing that LLM performance can vary significantly depending on language.

The role-based prompt offered a slight uplift in performance over the basic prompt, with 5% higher accuracy on the Java examples, while adding auxiliary information to prompts resulted in minor performance increases across both languages. These results suggest that LLMs are at least partially capable of utilising the contextual prompt information.

Chain of thought prompting resulted in significantly improved performance for C/C++, but lowered performance for Java. However, when auxiliary information was incorporated into chain-of-thought prompting, performance decreased for both languages. The researchers theorise that the performance drop is due to the position of auxiliary information in the prompts. If true, this shows how sensitive ChatGPT is to prompt changes. Even if the same

information and meaning is conveyed in the prompt, changing the order of keywords can induce a different response.

As the researchers predicted, standard and reverse prompts biased ChatGPT to predict examples as vulnerable and non-vulnerable respectively, showing the influence that prompt keywords have on performance. Through using positive and negative prompt words, the effective recall of the LLM can be adjusted. With this in mind, keeping prompt keywords neutral is likely to minimise bias.

The paper by Ullah et al.[6], discussed previously, also tested LLMs with a range of different prompt variations. These variations included adding the CWE of the vulnerable example or a description of the vulnerability into the prompt and designating the LLM with roles such as 'helpful assitant' and 'security expert'. The most effective prompts varied depending on which LLM was tested. GPT4 performed best when given other examples matching the CWE of the test cases with step-by-step reasoning for how the vulnerabilities were detected. This suggests that adding more context information in LLM prompts can significantly improve vulnerability detection performance. In addition, different LLMs may respond differently to the same prompts, so effective prompts for one LLM cannot be generalised to other LLMs.

Li et al.[22] investigated the effect of including emotive keywords to prompts by adding phrases such as "this is very important to my career" after an instruction. It was found that among all tested LLMs, the inclusion of these emotive keywords resulted in significantly improved performance across multiple task benchmarks. In addition, a human study was performed where LLM responses generated with emotive prompts were rated more favourably by participants. Although this paper did not focus on vulnerability detection, the above findings make this prompting approach a promising option regardless.

The results of these papers indicate that prompt choice has a significant impact on performance so should be carefully considered when using LLMs for vulnerability detection. However, adding more prompt information isn't always beneficial and can lead to a performance decrease. This supports the findings of Yu et al.[5] that LLMs have a tendency to overfocus on certain prompt details.

2.3 Combining Multiple Tools

Various approaches have been considered for using multiple code analysis tools in combination[2], [8], [14]. The simplest approach involves running a set of tools on the same code example and reporting a positive result if any of the tools detects a vulnerability. Logically, this will increase the number of vulnerabilities detected overall, increasing both the rate of true detections as well as false positive rate. The optimal tools to use together are those that cover different ranges of vulnerabilities [2].

The previously discussed paper of Lipp et al.[2] also investigated the effect of using multiple static analysis tools in combination, with combinations being limited to a maximum of six tools to minimise false positive rate. Despite their poor performance individually, Infer, Cppcheck and CodeChecker were included in most of the top-performing tool combinations as they detected some vulnerability types that the other tools did not. It was found that with the optimal combination of tools, vulnerability detection performance was up to 34% higher than the best individual tool. Ozturk et al.[8] also recorded a higher vulnerability detection rate when static tools were used in combination at the cost of slightly higher false positive rates. These findings support the idea that using multiple tools in combination can improve detection rate, although tools should be carefully selected to minimise the trade-off in false positive rate.

As well as combining multiple static tools, dynamic and static tools can also be used in combination. The two forms of analysis have complementary strengths and weaknesses[12] so would be expected to pair well when used together. One approach for combining these tools involves the selective use of dynamic analysis in cases where static analysis would struggle. Aggarwal and Jalote[14] devised a method where dynamic analysis would be used on functions marked as unsafe while static analysis would be used for the rest of the code. To achieve this, they created a tool that identifies and marks potentially unsafe functions. Reducing the coverage area of dynamic analysis decreases the number of test case required, causing less overhead. The method resulted in greater precision when detecting vulnerabilities and more vulnerabilities found overall compared to using static or dynamic analysis separately. Although this paper only considered buffer overflow vulnerabilities, the same principles should apply for different types of vulnerabilities.

CHAPTER 3

Methodology

3.1 Dataset

All experiments were performed on the DiverseVul dataset, released by Chen et al. in 2023[23]. This dataset consists of almost 19,000 vulnerable and over 300,000 secure C/C++ functions gathered from 7514 project commits. It was chosen due to its large size as well as its wide coverage of vulnerability types, consisting of 150 CWEs. Importantly, it is also the only dataset of its size to consist of complete functions rather than code snippets. As the dataset is comprised of real-world examples rather than synthetic examples, experimental results will give a good indication of how the tested tools and combined approaches would perform in practice.

3.2 Data Preprocessing

The dataset was filtered in a series of steps to make running experiments and analysing results easier. First, all commits that contained more than one CWE were removed as there was no way to tell which of the CWEs were present in any individual vulnerable function. Then, projects with multiple vulnerable functions were excluded as the analysis tools we are using are not equipped to deal with cross-file vulnerabilities. Finally, all code examples that produced syntax errors were removed, as some tools cannot produce an output in these cases.

At this stage, secure functions far outnumber vulnerable functions in the dataset. This is undesirable as it inflates the perceived performance of tools that are biased towards negative detections. For example, A tool that never produces a positive prediction would achieve roughly 95% accuracy if tested on the original dataset. To address this issue, a stratified random sample was taken of the secure functions to ensure an equal number of secure and vul-

nerable functions in the final data. For each vulnerable example, a random secure example of the same CWE label was chosen. Although the secure example contains no vulnerabilities, the CWE label indicates the CWE of the vulnerable example within the same project commit. This means that the secure example is likely to be similar in structure and contents to its corresponding vulnerable example, so tools are more likely to falsely flag it with its CWE label. After all these steps, the dataset consisted of about 5000 code examples in total covering 103 CWEs.

3.3 Choice of LLMs

OpenAI's current flagship models, GPT-4o and GPT-4o-mini were chosen as they will provide a good representation of current LLM performance. OpenAI's newest model, o1-mini will also be included in the experiments. This model, released on 12th September 2024, is trained with reinforcement learning and is claimed to be more capable at complex reasoning compared to its predecessors[17]. The mini variant was chosen over o1-preview as both models performed similarly in our early tests despite the latter being much slower and more expensive. Additionally, OpenAI advertises o1-mini as being specialised for coding applications, so was deemed to be the more suitable choice for vulnerability detection.

3.3.1 LLM Prompts

As the performance of LLMs is known to be substantially affected by prompt choice, a range of prompt variants were tested for each LLM. Each prompt variant is given below, along with a description of its expected effect. The placeholders '[CODE]' and '[BASELINE]' will be used to represent the input code and baseline prompt respectively. An additional prompt variant integrates static tool results into the prompt, therefore is classed as a combined approach and described in Section 3.6.1.

Baseline: The baseline prompt is the most basic prompt that all other prompt variants will modify. It asks the LLM to find any security vulnerabilities in the included code and respond with a 1 or 0 depending on whether a vulnerability was detected or not. The results for other prompt variants will be compared to the baseline prompt to isolate the effect that each variant has on performance.

> [CODE] Analyse the above C/C++ code snippet. Respond only with a single number. If the snippet contains any security vulner-

abilities, respond with 1. Otherwise respond with 0.

Anti: This variant reverses the phrasing of the baseline prompt, asking if no vulnerabilities are present. The approach is intended reduce false positives as the LLM is not directly asked to search for vulnerabilities. This will also showcase the effect that minor changes in wording have on performance.

> [CODE] Analyse the above C/C++ code snippet. Respond only with a single number. If the snippet contains no security vulnerabilities, respond with 0. Otherwise respond with 1.

Role: This variant assigns the LLM with the role of 'cybersecurity expert' before continuing with the baseline prompt. Assigning a role to LLMs has been shown to improve performance in some cases[6], [20], [21], so this may improve vulnerability classification.

> [CODE] You are a cybersecurity expert. [BASELINE]

Emotive: This variant incorporates an emotive prompt used in the 2023 paper by Li et al.[22], where it was shown to substantially improve LLM response quality.

> [CODE] [BASELINE]. Your correct classification is extremely important to my work.

Chain-of-Thought: This variant attempts to induce a reasoning process in the LLM by supplying multiple consecutive prompts. The LLM is able to use the information in its initial generated response, potentially improving its decision-making for the main task.

> Prompt 1: Describe the function of the above C/C++ code snippet.
>
> Prompt 2: Does this code contain security vulnerabilities? Respond only with a single number. If the snippet contains any security vulnerabilities, respond with 1. Otherwise, respond with 0.

3.4 Choice of Tools

3.4.1 Exclusion of Dynamic Tools

While dynamic tools were initially planned to be included in experiments, it was found that the vast majority of functions in the dataset could not compile in isolation, making these tools unusable. This was primarily due most

functions containing user-defined variables and types, which were declared in header files external to the functions.

3.4.2 Static Tools

The following factors were considered when selecting static tools:

- **Availability**: free tools were preferred, as freely-available LLMs were chosen.
- **Time of last update**: recently-updated tools were preferred to best represent current static tool performance
- **Usage**: tools that provided a command line option were preferred, as this would allow for easier automation through scripts
- **Customisation**: tools that provided additional program options were preferred, as this would allow the tools to be optimised for the dataset

Accounting for these factors, three widely-used open-source tools were chosen, shown in Table 3.1.

Table 3.1: Static tool selection

Tool:	Developer	Version Used	Release Date
Clang-Tidy[24]	LLVM Developer Group	17.0.3	17/08/2023
Cppcheck[25]	Daniel Marjamäki	2.13.0	23/12/2023
Flawfinder[26]	David A. Wheeler	2.0.19	30/08/2021

3.4.3 Tool Flags

Configuration settings such as program flags can significantly impact a tool's sensitivity to vulnerabilities, so tools were tested in multiple configurations to cover potential differences. All tools were tested with default settings (no extra flags), while additional tests were performed with certain flag options enabled, summarised in Table 3.2.

Flawfinder provides a couple of flags that can impact detection performance. The '–falsepositive' flag excludes flagged issues that are likely to be false positives, potentially at the cost of less true positives being caught. The '–minlevel' flag controls how severe a code issue has to be before it is counted as a flagged vulnerability, effectively controlling the detection sensitivity. The

Table 3.2: Tool flags

Flag	Tool	Description
-minlevel=x	Flawfinder	Minimum risk level (0-5) for an issue to be counted
-falsepositive	Flawfinder	Intended to decrease false positive rate
-checks=	Clang-Tidy	Controls which code checks are enabled

minimum value of 0 corresponds to no risk while the maximum value of 5 corresponds to the highest risk.

The default behaviour of Clang-Tidy is to run all checks, however checks can be enabled and disabled in groups with the '–checks' flag. Specifying only checks that are related to security vulnerabilities will help to reduce examples being flagged for other reasons, such as style guidelines. Our experiments using this flag included the following checks (descriptions from the official documentation[24]):

- **bugprone**: Checks that target bug-prone code constructs
- **cert**: Checks related to CERT Secure Coding Guidelines
- **clang-analyzer**: Clang Static Analyzer Checks
- **concurrency**: Checks related to concurrent programming

Cppcheck behaves opposite to Clang-Tidy, as its default settings disable all additional checks. The '–enable' flag allows additional checks to be run, however none of these checks relate to security vulnerabilities. For this reason, Cppcheck will only be tested in its default configuration.

3.5 Automating Tools

3.5.1 Static Tools

Python's subprocess module was used to run tools on each code example. This allowed each tool to be run as a separate process launched with command line arguments. When the process completes, the output can be captured for analysis. The standard out (stdout) and standard error (stderr) streams were concatenated before analysis as different static tools output to different streams. Outputs were analysed on a per-tool basis, as each tool

produces a different output format. To determine whether a vulnerability was detected, pattern-matching was used on the output to find terms that only appeared when issues were flagged.

3.5.2 LLMs

For each code example, the OpenAI chat completions API was used to generate an output for the input prompt. No output processing was required as all prompts instructed the LLM to respond with 1 or 0 depending on whether a vulnerability was detected. Any other responses would be deemed as a vulnerable detection, as the code example could not be confirmed as secure in this case.

A length cut-off was introduced for code examples to prevent prompts from exceeding the LLM's context window as well as reducing the API cost for larger examples. With the set threshold value of 50,000 characters, only 24 code examples exceeded the threshold and were truncated, accounting for 0.56% of total examples.

Due to API cost limitations and the size of the dataset, we were unable to run each prompt multiple times to counteract the inherent non-determinism of LLMs. However, the large dataset size meant that non-determinism had a negligible effect on overall results, which was seen when LLMs were run on the dataset multiple times in our early testing phases.

3.6 Combined Approaches

3.6.1 Static-Enhanced Prompting

The first combined approach involved integrating static tool results directly into the LLM prompt. Ideally, the LLM will be able to make use of this additional context information to improve its decision-making. The selection of tool results to be included in the LLM prompt will depend on the individual tool performances in the results stage, with better performing tools being chosen. Tools with low false positive rates will be preferred to counteract the expected high false positive rates of LLMs. The static-enhanced prompting process is summarised in Figure 3.1.

For each code example to be analysed, the LLM prompt will include the result of each included static tool for that example. Tool results are given in JSON format with fields indicating the tool name and the detection result (secure or vulnerable). The prompt used for this method is given below:

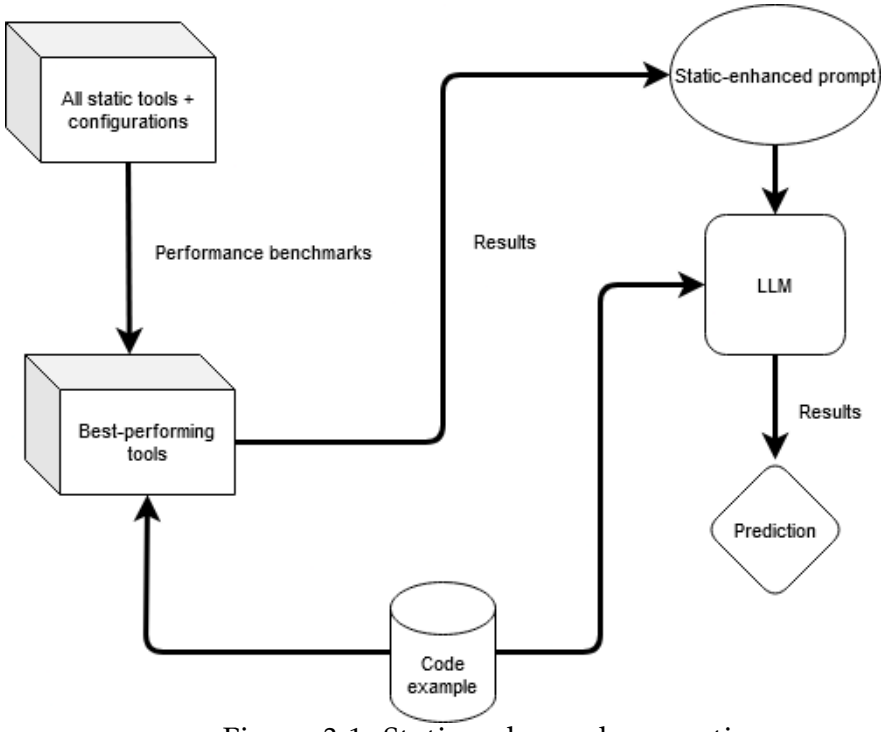


Figure 3.1: Static-enhanced prompting process

> [CODE] [BASELINE] A range of code analysis tools analysed the same snippet and produced the following results (in JSON format): [TOOL RESULTS]

3.6.2 Tool Ensembles

Tool ensembles aggregate the results from multiple predictors into one overall prediction. By having multiple independent sources contributing to the overall prediction, this prediction should be more accurate and reliable than using an individual tool. However, the performance of an ensemble will depend on the performance of its individual predictors. As with the static-enhanced prompting, the selection for tools and LLMs to be included in the ensemble will depend on how each tool and LLM performs individually, including prompt variations and different program configurations. Figure 3.2 illustrates the process of using an ensemble for predictions. We will consider the following three ensembles:

- **Static Tools Only**
- **LLMs Only**

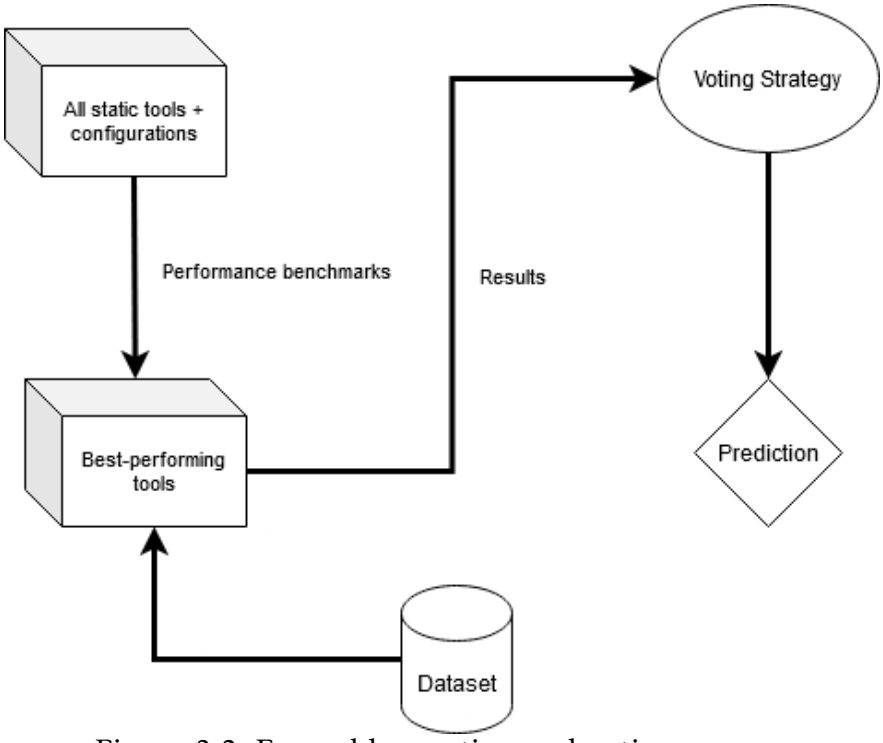


Figure 3.2: Ensemble creation and voting process

- **Static Tools + LLMs**

Tool ensembles implement a voting strategy to determine the method by which individual predictions are aggregated. Three simple voting strategies were tested for each ensemble, summarised below:

- **Any**: The ensemble predicts positive if any of the tools predict positive.
- **All**: The ensemble only predicts positive if all of the tools predict positive.
- **Majority**: The ensemble predicts positive if a majority of tools predict positive.

With a majority voting strategy, a tiebreaker needs to be implemented to determine the overall decision in the case that an equal number of tools produce positive and negative predictions. For our tiebreaker, the ensemble will randomly choose between a positive or negative prediction to minimise any bias towards either option. Note that the tiebreaker only comes into effect if there are an even number of predictors in an ensemble, as there are only two possible prediction options.

3.7 Detection Criteria

The CWE of a vulnerability was not considered in the detection process. If a tool correctly detected that a vulnerability was present, it was recorded as a correct prediction even if the output of the tool gave an incorrect CWE label or description of the vulnerability. This approach was chosen for multiple reasons. First, tool outputs differ significantly in format and not all tool outputs include specific CWE-IDs. While it is possible to add a layer such as another LLM that classifies tool outputs into CWE-IDs, such a layer would need to classify accurately enough to not significantly impact tool results.

Second, there are often multiple valid CWEs that could describe a particular vulnerability. For example, CWE-121 (Stack-based Buffer Overflow) and its parent CWE-787 (Out-of-bounds write) are both legal CWE mappings that could be applied to the same vulnerability. However, if the CWE-ID produced by a tool does not match the single CWE that appears in the example label, this will be considered as an incorrect detection, which limits the usefulness of CWE-separated results.

Finally, the primary goal of a static analysis tool is to flag a vulnerability for human review and repair. In this context, the knowledge of whether or not a code segment is vulnerable is more important than the specific type of vulnerability found, as an experienced programmer will be able to make a more accurate judgement in determining the CWE of a vulnerability than any tool or LLM.

3.8 Metrics

When a tool or LLM analyses a code example, one of the four following outcomes will occur:

- **True Positive (TP)**: A vulnerable code example is correctly classified as vulnerable
- **True Negative (TN)**: A secure example is correctly classified as secure
- **False Positive (FP)**: A secure example is incorrectly classified as vulnerable
- **False Negative (FN)**: A vulnerable example is incorrectly classified as secure

The outcomes for each tool can be summarised with a set of metrics, described in Table 3.3. In every graph, metric results are separated between

CWEs within the MITRE Top 25 list and CWEs not on the list. This will allow us to determine whether detection approaches have a bias towards one of these CWE groups or whether they perform similarly among both groups. Results tables will present metrics across all CWEs to reduce clutter.

Table 3.3: Metrics

Metric:	**Summary:**	**Formula:**
Accuracy	Out of all code examples, what proportion was correctly classified.	$\frac{TP+TN}{TP+TN+FP+FN}$
False Positive Rate (FPR)	Out of all secure examples, what proportion was incorrectly flagged as vulnerable.	$\frac{FP}{FP+TN}$
Recall	Out of all vulnerable examples, what proportion was correctly flagged as vulnerable.	$\frac{TP}{TP+FN}$
Precision	Out of all flagged positives, how many were actually vulnerable.	$\frac{TP}{TP+FP}$
F1 score	Harmonic mean of precision and recall.	$\frac{2 \times Precision \times Recall}{Precision + Recall}$

CHAPTER 4

Results

4.1 Static Tool Performance

Using default flags, there were extreme performance differences between the static tools, shown in Figure 4.1. This is made most clear by focusing on the recall and false positive rate for each tool. Flawfinder had the most favourable performance, with a moderate recall and low false positive rate at 0.342 and 0.120 respectively for top-25 CWEs. This result is reflected in the accuracies between the tools, where Flawfinder achieved the highest at 0.612 for top-25 CWEs while other tools recorded around 0.5.

In contrast, Cppcheck struggled to detect vulnerabilities with a very low recall of 0.081. However, its false positive rate also remained extremely low, at 0.042, making it a viable option for reducing false positives in the combined approaches.

While Clang-Tidy had by far the highest recall, at 0.982 for CWEs outside of the top-25, the tool suffered from false positive rates over 0.9 for both CWE groups. A closer inspection of the tool's output revealed that in over 96% of code examples, positive detections were due to Clang-Tidy reporting one or both of the following errors: 'undeclared identifier' and 'unknown type name'. As previously mentioned, these errors are a result of the code examples being isolated functions and therefore excluding type and variable definitions that exist in a project's header files. Despite being a static tool, Clang-Tidy had a higher standard for code syntax and completeness than other tested static tools.

Additionally, Cppcheck and Flawfinder showed a slight bias towards detecting top-25 vulnerabilities, resulting in marginally higher accuracy for this vulnerability group across both tools.

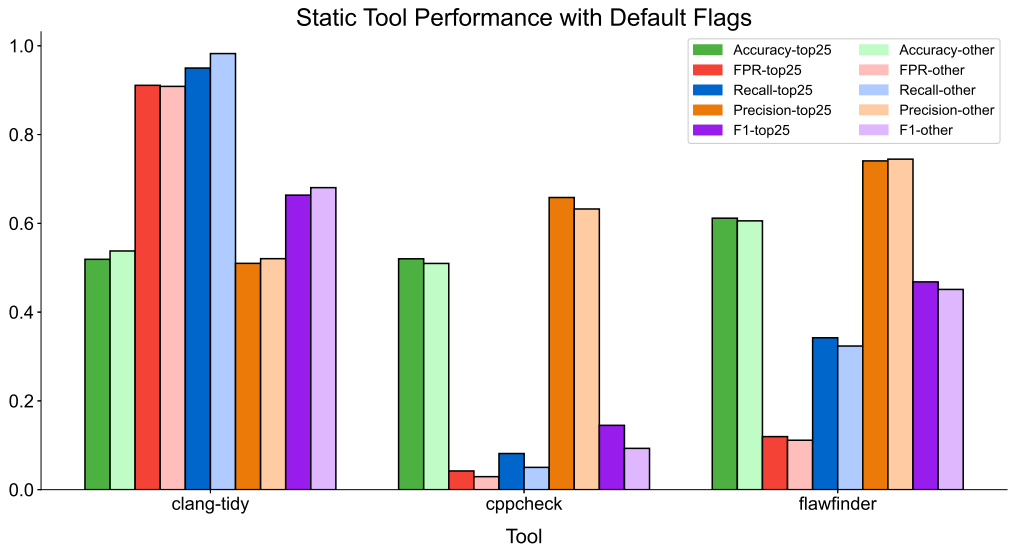


Figure 4.1: Result metrics for each static tool with default flags

4.2 Tool Flags

4.2.1 Flawfinder

Table 4.1: Performance metrics by minlevel value

Flawfinder Flag	Accuracy	False Positive Rate	Recall	Precision	F1 Score
minlevel=0	**0.610**	0.129	**0.349**	0.729	**0.472**
minlevel=1 (default)	0.609	0.116	0.335	0.742	0.461
minlevel=2	0.597	0.094	0.287	0.754	0.416
minlevel=3	0.514	0.012	0.040	**0.768**	0.077
minlevel=4	0.512	0.011	0.035	0.758	0.067
minlevel=5	0.500	**0.001**	0.001	0.500	0.003

From Table 4.1, it can be seen that Flawfinder performed best with minlevel values of 0 and 1. These two options performed very similarly, with the former offering a better recall while the latter had a lower false positive rate. However, a steep drop-off in recall was observed for minlevel values exceeding 3, resulting in much lower F1 scores. This suggests that most vulnerabilities in the dataset were not considered severe enough to be detected at higher risk levels.

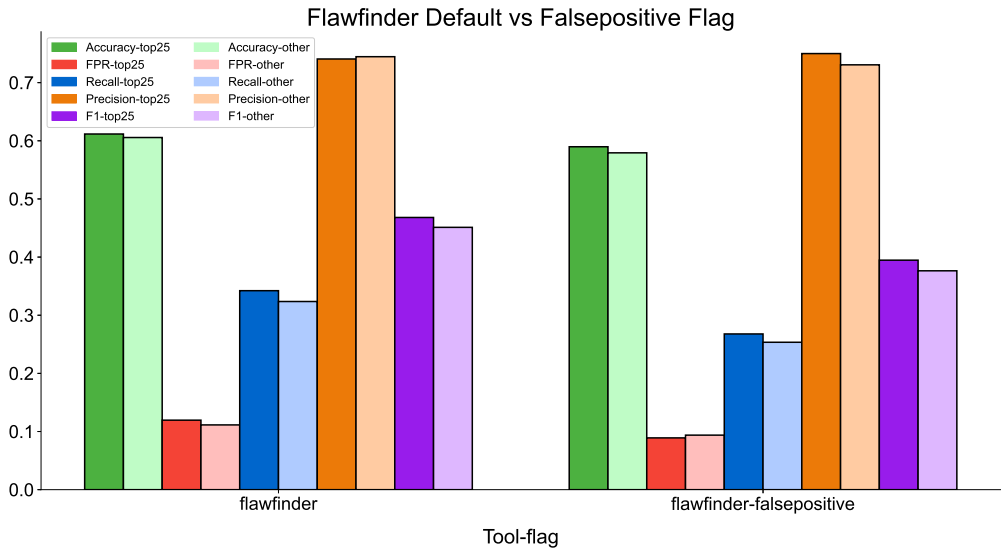


Figure 4.2: Flawfinder performance with falsepositive flag vs default settings

As shown in Figure 4.2, the falsepositive flag was successful in decreasing false positive rates, however recall also suffered as a result. The lower F1 score and accuracy observed when the falsepositive flag was used suggests that this trade-off is not worthwhile. With these results in mind, we will be using the default flags for Flawfinder in the ensemble and static-enhanced LLM approaches.

4.2.2 Clang-Tidy

Figure 4.3 shows that Clang-Tidy's performance remained almost unchanged when the check flag was included. This was to be expected as the false positive issues brought up in section 4.1 were due to the characteristics of the dataset rather than the sensitivity of the tool.

4.3 LLM performance

LLMs had better detection performance than static tools overall, shown in Figure 4.4. The models gpt-4o and gpt-4o-mini achieved accuracies of 0.638 and 0.657 respectively, exceeding Flawfinder's peak accuracy of 0.610, and both models produced much higher recalls than any static tool excluding Clang-Tidy. However, false positive rates of LLMs were also significantly

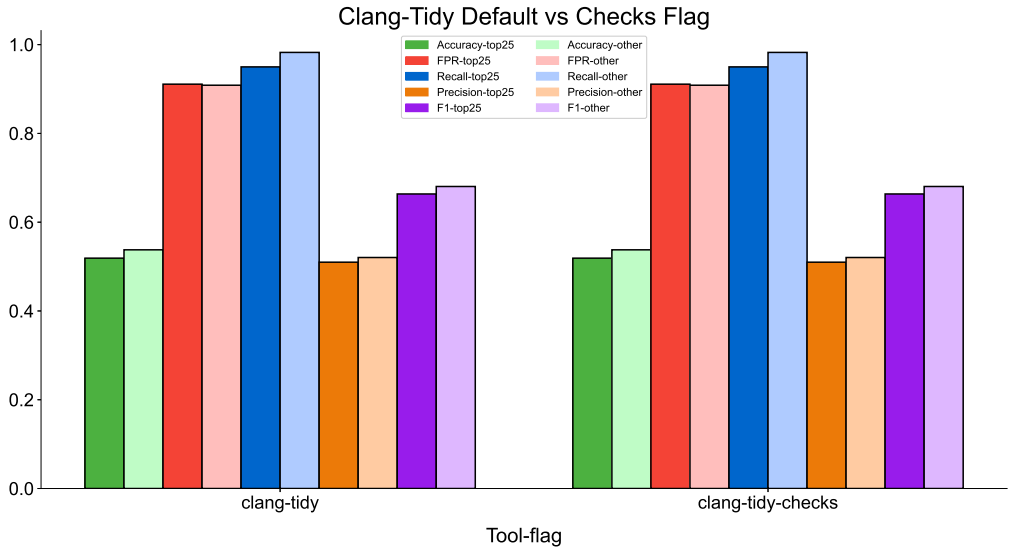


Figure 4.3: Clang-Tidy performance with checks flag vs default settings

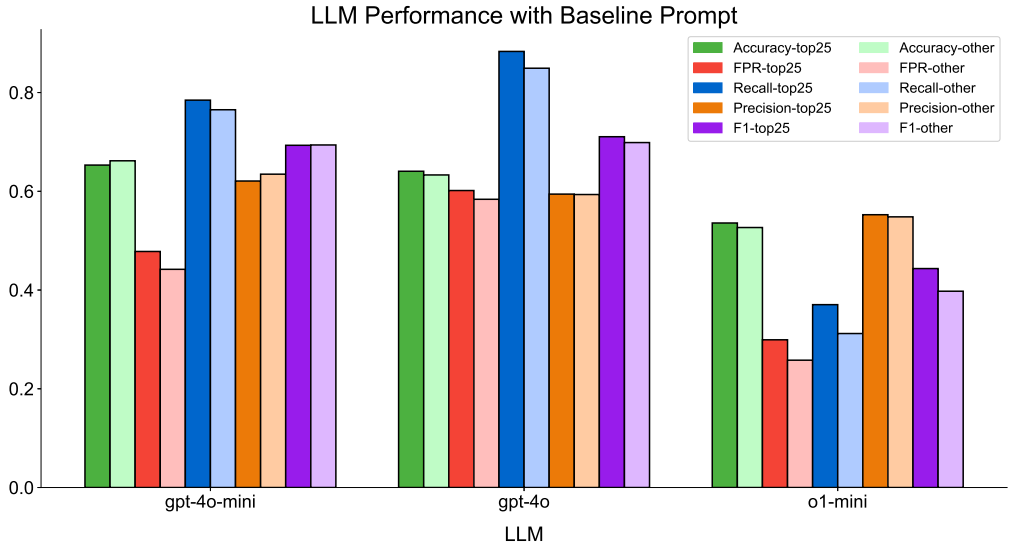


Figure 4.4: Result metrics for each model using the baseline prompt

higher, with gpt-4o-mini's false positive rate of 0.594 being over 4 times greater than Flawfinder's peak of 0.129.

Surprisingly, o1-mini performed the worst out of all LLMs using baseline prompt, with its relatively low recall of 0.370 being only slightly higher than Flawfinder's peak recall of 0.349. In addition, the false positive rate of o1-mini significantly exceeded that of Flawfinder, resulting in lower accuracy, precision and F1 score when compared to the static tool. Due to this, o1-mini was the only LLM with an accuracy of under 0.6, at 0.532.

All LLMs recorded a higher recall for top-25 vulnerabilities, although false positive rates for this vulnerability group was also higher. This resulted in minor accuracy improvements in the case of gpt-4o and o1-mini, while gpt-4o-mini had lower accuracy for top-25 vulnerabilities.

4.4 Prompt Variants

Table 4.2: gpt-4o prompt variant performance

Prompt Variant	Accuracy	False Positive Rate	Recall	Precision	F1 Score
Baseline	0.638	0.594	0.870	0.594	**0.706**
Role	0.622	0.654	0.899	0.579	0.704
Emotive	0.619	0.622	0.859	0.580	0.692
Chain	**0.643**	**0.454**	0.740	**0.620**	0.674
Anti	0.616	0.675	**0.907**	0.573	0.703

Table 4.3: gpt-4o-mini prompt variant performance

Prompt Variant	Accuracy	False Positive Rate	Recall	Precision	F1 Score
Baseline	0.657	**0.464**	0.777	0.626	0.693
Role	0.611	0.708	**0.930**	0.568	0.705
Emotive	**0.660**	0.469	0.788	**0.627**	0.698
Chain	0.630	0.663	0.923	0.582	**0.714**
Anti	0.619	0.658	0.897	0.577	0.702

Prompt variant results for gpt-4o, gpt-4o-mini and o1-mini are presented in Tables 4.2, 4.3 and 4.4 respectively.

For all models, role-based prompting had the effect of increasing detection sensitivity, resulting in higher recall at the cost of higher false positive rate.

Table 4.4: o1-mini prompt variant performance

Prompt Variant	Accuracy	False Positive Rate	Recall	Precision	F1 Score
Baseline	0.532	0.283	0.347	0.551	0.426
Role	0.537	0.343	**0.417**	0.549	**0.474**
Emotive	0.536	0.269	0.340	0.558	0.423
Chain	**0.545**	0.274	0.365	**0.571**	0.446
Anti	0.538	**0.235**	0.310	0.569	0.402

The effect was especially noticeable for gpt-4o-mini and o1-mini, where recall and false positive rate for the role-based prompt were higher than any other variant.

Emotive prompting had varying effects depending on the model used. For gpt-4o, performance with the emotive prompt was worse than the baseline across every metric. However, the variant was particularly effective for gpt-4o-mini, where it achieved the highest accuracy and precision of all model-prompt combinations. For o1-mini, emotive prompting resulted in minor accuracy and precision improvements over the baseline, although recall was slightly decreased.

Out of all prompt variants, chain-of-thought prompting seemed to be most effective in improving detection performance. The prompting method resulted in the highest accuracy and precision along with the lowest false positive rate for gpt-4o, although its lower recall compared to the baseline resulted in a lower F1 score. With o1-mini, chain-of-thought prompting increased recall while reducing false positive rate, resulting in the highest accuracy and precision among prompt variants for the model. However, the results for chain-of-thought prompting with gpt-4o-mini were less impressive, with a lower accuracy and substantially higher false positive rate than the baseline. The increase in detection sensitivity meant that the variant surpassed the recall of the baseline, resulting in the highest F1 score of all model-prompt combinations.

Although the anti prompt was intended to reduce false positive rate, this effect was only seen with o1-mini, where false positive rate was the lowest out of all models. This came with the expected decrease in recall, however accuracy and precision were still higher than the baseline prompt. For gpt-4o and gpt-4o-mini, the anti prompt had the opposite effect and increased false positive rates along with recall when compared to the baseline. This resulted in the anti prompt having the highest recorded false positive rate along with the

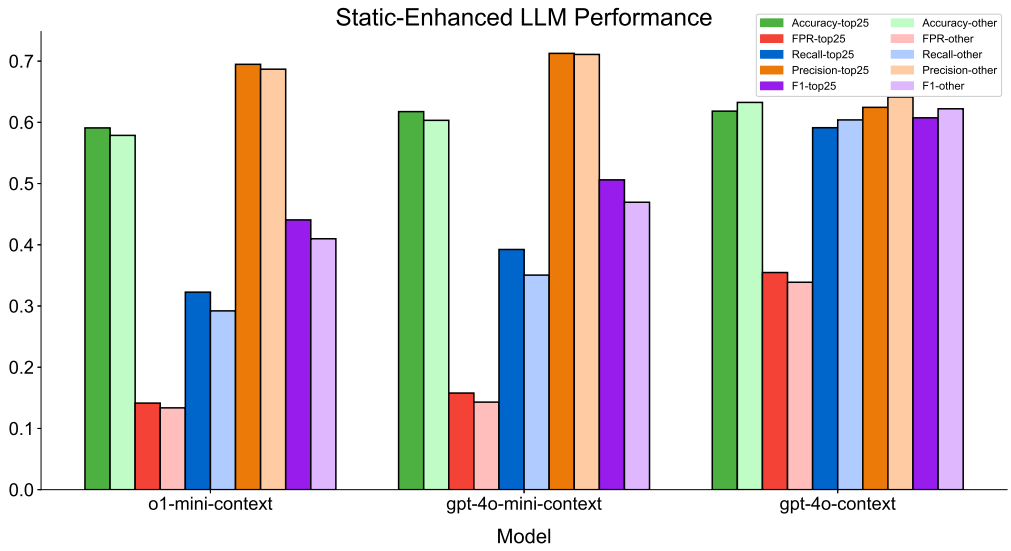


Figure 4.5: Static-enhanced LLM performance

lowest accuracy out of all gpt-4o prompt variants.

In summary, the chain-of-thought prompt variant was the best-performing variant for gpt-4o and o1-mini among a range of metrics such as accuracy and precision while also providing a reasonable trade-off between recall and false positive rate. For gpt-4o-mini, the emotive prompt appeared to perform the best, slightly outperforming the baseline prompt in terms of accuracy and recall while offering the highest precision. These three LLM-prompt variant combinations will make up the selection of LLMs in the tool ensembles.

4.5 Static-Enhanced Performance

Table 4.5: Static-enhanced LLM performance

Model	Accuracy	False Positive Rate	Recall	Precision	F1 Score
gpt-4o	**0.624**	0.348	**0.596**	0.631	**0.613**
gpt-4o-mini	0.612	0.152	0.376	**0.712**	0.492
o1-mini	0.586	**0.138**	0.310	0.692	0.428

Results from the static tools Flawfinder and Cppcheck, both using default

settings, were included as prompt inputs for the static-enhanced approach. Clang-Tidy was excluded due to its previously mentioned performance issues on the dataset. The results for this approach are summarised in Table 4.5 and shown visually in Figure 4.5. It can be seen that the inclusion of static tool results was very effective in reducing false positive rates, with each model producing a false positive rate significantly lower than any of its prompt variants. This decrease was most obvious with gpt-4o-mini, where false positive rate dropped from 0.464 with the baseline prompt to 0.152 with the static-enhanced prompt. However, the lower false positive rate came at the cost of a significant decrease of all other metrics for the GPT models, particularly recall, which dropped from 0.777 to 0.376 for gpt-4o-mini. Accuracy also dropped from 0.657 to 0.612 for the same model.

A notable exception to this trend is the model o1-mini, which achieved an accuracy of 0.586 with the static-enhanced approach, exceeding the peak accuracy of 0.545 for its best-performing prompt variation. This is likely because recall only fell by 0.055 as compared to the chain-of-thought prompted model, which was a much smaller drop than what was observed with the other models. These results show that static-enhanced prompting was a viable option for o1-mini. This suggests that the model was able to make use of context information better than other models, potentially due to its superior complex reasoning abilities.

4.6 Ensemble performance

As with the static-enhanced approach, the static tools included in ensembles will be Flawfinder and Cppcheck, using default settings. LLMs will be included with the optimal prompt variants found in Section 4.4. These are the chain-of-thought prompt for gpt-4o and o1-mini, and the emotive prompt for gpt-4o-mini.

Table 4.6: Static tools ensemble performance by voting method

Voting Method	Accuracy	False Positive Rate	Recall	Precision	F1 Score
Any	**0.611**	0.145	**0.367**	0.717	**0.486**
All	0.514	**0.008**	0.037	**0.812**	0.070
Majority	0.568	0.078	0.213	0.732	0.330

Tables 4.6, 4.7 and 4.8 summarise the results for the three voting methods tested for each ensemble. Changing voting method had the effect of changing

Table 4.7: LLMs ensemble performance by voting method

Voting Method	Accuracy	False Positive Rate	Recall	Precision	F1 Score
Any	0.618	0.654	**0.890**	0.577	**0.700**
All	0.570	**0.134**	0.274	**0.672**	0.390
Majority	**0.659**	0.410	0.729	0.640	0.681

Table 4.8: Static tools + LLMs ensemble performance by voting method

Voting Method	Accuracy	False Positive Rate	Recall	Precision	F1 Score
Any	0.617	0.665	**0.900**	0.575	**0.702**
All	0.507	**0.003**	0.016	**0.850**	0.031
Majority	**0.629**	0.208	0.467	0.692	0.558

the detection sensitivity of ensemble, with the 'all' method favouring lower false positive rate and higher precision while the 'any' method favoured high recall. Majority voting was the middle-ground between these two extremes and offered a reasonable false positive rate without a large compromise to recall.

From these results, it can be seen that the optimal voting method depends on the range of tool sensitivities present within the ensemble. When ensembles included tools with extremely low recall, such as Cppcheck, the 'all' voting method became unsuitable, as the ensemble inherits this low recall value, harming accuracy and F1 score. This effect was intensified when the ensemble size grew, with the 'static tools + LLMs' ensemble recording even lower recall and accuracy than the static tools ensemble. Although no tools with extremely high recall were included in ensembles, the opposite effect would be expected when using the 'any' voting method if such a tool was included. This would result in a high false positive rate with low precision and accuracy.

For the LLMs ensemble and the 'static tools + LLMs' ensemble, the best performance in terms of accuracy was given by the majority voting method, with reasonable performance across all other metrics. However, in the case of the static tool ensemble, the 'any' voting method offers the best performance, outperforming majority voting in terms of accuracy, recall and F1 score while slightly falling behind in precision. This is likely due to the already low sensitivities of the individual static tools.

When considering all voting methods tested, majority voting was the most

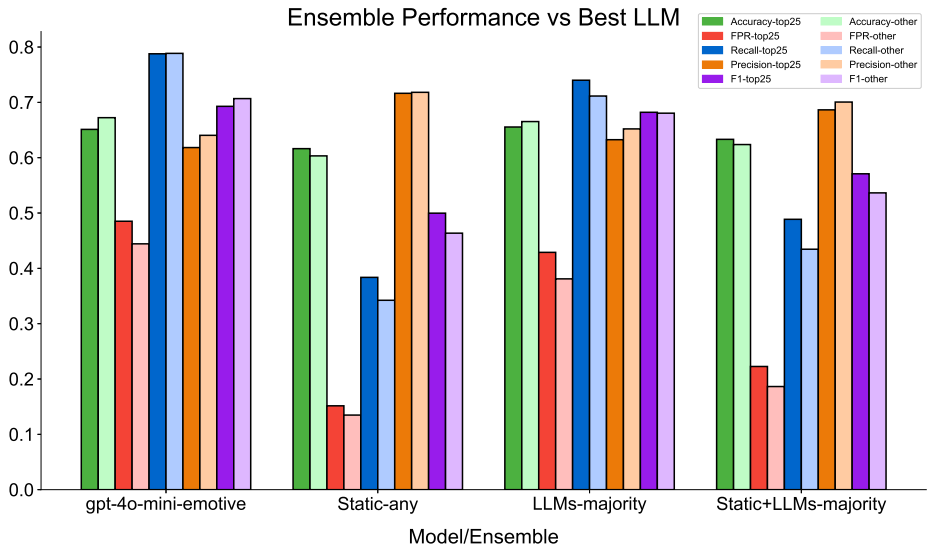


Figure 4.6: Ensemble Performance vs Best LLM

stable option, especially for larger ensembles and ensembles with more variation in tool sensitivities. However, the other voting methods may offer better performance in cases where the sensitivities of ensemble tools were more homogeneous.

Figure 4.6 shows the performance of each ensemble with the optimal voting methods compared to the best-performing LLM in terms of accuracy, gpt-4o-mini with the emotive prompt. While none of the ensemble approaches improved on the accuracy of gpt-4o-mini, all ensembles achieved reasonable accuracies above 0.6. In addition, the ensembles showcased a wide range of recall values and corresponding false positive rates. This suggests that ensembles with carefully considered tool selections may be a good option for adjusting detection sensitivity for a particular use case.

CHAPTER 5

Discussion

5.1 Results Summary

For every detection approach tested, there were trade-offs between performance metrics, with no approach offering the best performance for all metrics simultaneously. Higher sensitivity methods produced higher recall and F1 score at the expense of higher false positive rate and lower precision while the opposite was seen for lower sensitivity methods. Accuracy was the most reliable metric to determine overall performance as it doesn't bias higher or lower sensitivity options, but focusing on this metric alone would hide the trade-offs made between other metrics. The optimal approach will depend on the usage context and whether higher recall or lower false positive rate is preferable.

All static tools with the exception of Clang-Tidy had low detection sensitivities, resulting in low false positive rates. Cppcheck especially struggled to find vulnerabilities and had the lowest recall of all methods tested. Although these results go contrary to the known high false positive rates of static tools when analysing synthetic examples, they align with what was seen in various studies[2], [16] that applied static tools to real-world scenarios. This makes sense as our dataset also consisted of functions from real projects. In addition, the issues experienced with Clang-Tidy highlighted the challenges of applying static tools to real situations, especially under the syntactically strict C and C++ languages.

5.1.1 Issues with F1 Score

Although F1 score captures false positive rate through the precision metric, it was found that the metric favoured high recall at the expense of high false positive rate. This was due to fact that precision does not drop below 0.5, even in the case of extremely sensitive tools, as precision will approach the

ratio of vulnerable examples to total examples present in the dataset. In contrast, recall is able to drop to near-zero as precision rises.

The issues with this metric were seen with the example of Clang-Tidy, which achieved an F1 score of 0.67 despite the tool not functioning correctly with the dataset. Since the tool flagged almost all examples as positive, it would be unusable for finding real vulnerable examples in the dataset. A tool such as Flawfinder would be much more useful in practice despite is lower recall, as there would be a higher certainty that flagged examples were truly vulnerable. However, this was not reflected in Flawfinder's F1 score which was considerably lower at 0.461.

For this reason, F1 score was given less consideration than other metrics, and the trade-off between recall and precision was instead considered by directly comparing the two metrics.

5.1.2 Top-25 Vulnerabilities

It was expected that LLMs would be better at detecting vulnerabilities within the top-25, as these vulnerabilities are more common and therefore more likely to appear in LLM training data. While recalls were noticeably higher for this vulnerability group, LLMs were also more likely to incorrectly flag secure examples with top-25 CWE labels, increasing false positive rates to the same degree. This meant that in terms of accuracy, LLMs performed very similarly between vulnerabilities within and vulnerabilities outside of the top-25, similar to what was seen with the static tools. For one of the LLMs, gpt-4o-mini, accuracy was lower for top-25 vulnerabilities compared to non-top-25 vulnerabilities.

From these observations, it seems that while the detection sensitivity of LLMs is higher for more common vulnerabilities, detection accuracy remains relatively consistent regardless of how frequently vulnerabilities appear in the training data. However, this trend may change for newly-discovered CWEs that were not present in LLM training data nor in our dataset. It should also be noted that the LLMs included in our experiments were all designed for general-purpose tasks and these results may not translate to LLMs that are fine-tuned for coding tasks.

5.2 Research Questions

RQ1: Can static tool flags be optimised to improve vulnerability detection performance?

For our tool selection, default flags provided the best performance on the DiverseVul dataset, although optimal settings will depend on the characteristics of the code being analysed. Therefore, the presence of options such as Flawfinder's 'minlevel' flag allow static tool detection sensitivity to be adjusted to best cater to the situation.

RQ2: Does o1-mini provide better vulnerability detection performance than its predecessors?

When using the baseline prompt, o1-mini was significantly outperformed by the other models, gpt-4o and gpt-4o-mini in terms of accuracy, recall and precision. Despite having a lower false positive rate than the other models, its low precision prevented the baseline version of the model from being a viable option. Performance improved when applying prompt variations such as chain-of-thought prompting, however o1-mini still fell behind the other models. The static-enhanced prompt offered the best performance for o1-mini, with the lowest false positive rate and highest precision of any LLM model. Although other models still offered better accuracy and recall, the static-enhanced iteration of o1-mini may be a better option in cases where low false positive rate is preferable to high recall.

RQ3: Can LLM vulnerability detection performance be improved through prompt variations?

For every model tested, there were prompt variants that provided better performance than the baseline prompt for certain metrics, with the optimal prompt depending on which metrics are most important for the use case. However, prompt variants did not have the same effect on every model, seen with the chain-of-thought prompt producing the best accuracy for gpt-4o and o1-mini, but reducing accuracy compared to the baseline for gpt-4o-mini. Other prompt variants behaved unexpectedly and didn't bring about their intended effects. For example, the anti prompt lowered false positive rate for o1-mini as intended, but had the opposite effect for gpt-4o and gpt-4o-mini. Due to the observed inconsistencies, prompts that are effective for one LLM cannot be assumed to be effective for others, and multiple prompts should be tested to determine which performs the best for each model in each situation.

RQ4: Do our combined approaches provide better performance than static tools and LLMs individually?

Including static tool results in LLM prompts effectively made the models less sensitive, as the static tools used were of low detection sensitivity. This approach was more effective in reducing false positive rate than any prompt variant. However, accuracy was also reduced for gpt-4o and gpt-4o-mini due to the large drop in recall, meaning that the trade-off was unlikely to be worthwhile for these models. As mentioned before, the performance of this prompting approach was more favourable with o1-mini, where it achieved its highest accuracy and lowest false positive rate, making it our best tested approach for this model.

This method may have been more effective if a greater number of static tools were included, as the sample size of two static tools was fairly small. A more varied set of tools may also have resulted in better performance across more metrics as both of the included static tools had low detection sensitivities.

Ensemble approaches also helped to reduce false positive rates. While accuracy was still lower than the best individual LLM, the ensemble of static tools and LLMs achieved a greater accuracy than any LLM with static-enhanced prompting, suggesting that the ensemble approach was more effective than static-enhanced prompting in reducing false positive rate while minimising decreases in recall. However, the LLM ensemble showed the best performance of any ensemble, with almost identical accuracy to the best-performing LLM while maintaining a lower false positive rate.

One advantage of static tools is their ability to analyse code examples much faster than LLMs without incurring LLM API costs. Ensembles of static tools possess the same advantages, making them a good option for fast and inexpensive code analysis that outperforms individual static tools. Similarly to static-enhanced prompting, this approach would be made more viable with a larger and more varied set of static tools, as the static tools ensemble only had minimal performance improvements over the best-performing static tool with our tool selection.

In summary, the combined approaches and tool ensembles in particular were most effective at reducing false positive rates out of all approaches tested, although there were still compromises to recall and accuracy. These approaches offered a more controlled way of reducing the false positive rates associated with LLMs as the effects of prompt variations were not always predictable.

CHAPTER 6

Conclusion

6.1 Limitations

6.1.1 Dataset

Despite DiverseVul's size and exceptional coverage of vulnerability types, it is limited by only including examples in C/C++. A multi-language dataset or multiple language-specific datasets would have allowed us to determine whether the performance of LLMs and combined approaches would be affected by programming language. In this case, the combined approaches would have to be modified to accommodate multiple languages. Static tools are language-dependent so different sets of static tools would have to be rotated through depending on the language of the analysed code example.

In addition, our decision to focus on a single dataset limits the degree to which our results can be generalised. Expanding the experiments to more datasets would help to validate our findings and give more confidence that our methods would behave in the same way in different situations. While it was our intention to include other datasets for benchmarking, our main difficulty with finding suitable datasets was the requirement of static tools that code examples be fully syntactically valid. Existing vulnerable code datasets such as CyberSecEval[27] consist of code snippets, so the vast majority of code examples resulted in syntax errors when attempting to run static analysers. This issue is exacerbated by the strict syntactic rules of the C and C++ languages.

6.1.2 Tools and LLMs

The small number of tools tested as well as the exclusion of non-static tool types was a key limitation of our research. Our findings suggested that a larger pool of tools covering a wider variety of detection sensitivities may

have improved the performance of the combined approaches. Assuming that a suitable dataset was found or created, dynamic tools would have been a valuable inclusion due to their low false positive rates and high precision. In addition, recently developed deep-learning-based code analysis tools have shown very impressive results in studies[16], achieving high recall while maintaining low false positive rates, although the public availability of these tools is limited for now.

While we tested OpenAI's most recent publicly-available models, the performance of other LLMs would have added significantly to our research. These include other popular general-purpose models such as Google's Gemini[28] and Anthropic's Claude[29] as well as LLMs that are fine-tuned to perform optimally in coding tasks, such as Meta's CodeLLama[18]. Studies have shown that LLMs fine-tuned for coding are capable of outperforming larger general LLMs in coding tasks[19], so applying a coding LLM to our combined approaches may have resulted in greater performance gains.

6.2 Future Work

6.2.1 Prompt Variations

While we tested a variety of prompting techniques, the list of prompt variations could have been expanded to better explore the full capabilities of LLMs as vulnerability detectors. In addition, each of our prompt variations only implemented a single technique at a time. The combination of multiple effective techniques may have resulted in further performance improvements, although the unpredictability we observed with prompts variations does not make this a guarantee. However, further experimentation with LLMs would have significantly increased time and API costs of running tests, which was already a significant hurdle in our research.

6.2.2 Advanced Aggregation Methods

Finally, future work can investigate more sophisticated methods for aggregating tool results, as our methods were limited to prompt variations and simple ensemble voting. One way to achieve this would be through a weighted ensemble utilising a neural network architecture to train weights. Tools would be assigned weights based on the reliability of their predictions on the training data. These weights would then affect the degree to which each tool's prediction would contribute to the overall ensemble prediction. Furthermore, through analysing individual tool results separated by CWE, the best-

performing tools for a set of CWE groups can be determined. This information can be used to further fine-tune weights depending on the suspected CWE group of the example to be analysed.

Bibliography

[1] A. Kaur and R. Nayyar, 'A comparative study of static code analysis tools for vulnerability detection in c/c++ and java source code', *Procedia Computer Science*, vol. 171, pp. 2023–2029, 2020, Third International Conference on Computing and Network Communications (CoCoNet'19), ISSN: 1877-0509. DOI: https://doi.org/10.1016/j.procs.2020.04.217. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1877050920312023.

[2] S. Lipp, S. Banescu and A. 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*, ser. ISSTA 2022, South Korea (Virtual): Association for Computing Machinery, 2022, pp. 544–555, ISBN: 9781450393799. DOI: 10.1145/3533767.3534380. [Online]. Available: https://doi.org/10.1145/3533767.3534380.

[3] OpenAI *et al.*, *Gpt-4 technical report*, 2024. arXiv: 2303.08774 [cs.CL].

[4] R. Khoury, A. R. Avila, J. Brunelle and B. M. Camara, *How secure is code generated by chatgpt?*, 2023. arXiv: 2304.09655 [cs.CR].

[5] J. Yu, P. Liang, Y. Fu *et al.*, *Security code review by llms: A deep dive into responses*, 2024. arXiv: 2401.16310 [cs.SE].

[6] S. Ullah, M. Han, S. Pujar, H. Pearce, A. Coskun and G. Stringhini, *Llms cannot reliably identify and reason about security vulnerabilities (yet?): A comprehensive evaluation, framework, and benchmarks*, 2024. arXiv: 2312.12575 [cs.CR].

[7] M. C. Sánchez, J. M. C. de Gea, J. L. Fernández-Alemán, J. Garceran and A. Toval, 'Software vulnerabilities overview: A descriptive study', *Tsinghua Science and Technology*, vol. 25, no. 2, pp. 270–280, 2020. DOI: 10.26599/TST.2019.9010003.

[8] O. S. Ozturk, E. Ekmekcioglu, O. Cetin, B. Arief and J. Hernandez-Castro, 'New tricks to old codes: Can ai chatbots replace static code analysis tools?', in *Proceedings of the 2023 European Interdisciplinary Cybersecurity Conference*, ser. EICC '23, Stavanger, Norway: Association for Computing Machinery, 2023, pp. 13–18, ISBN: 9781450398299. DOI: 10.1145/3590777.3590780. [Online]. Available: https://doi.org/10.1145/3590777.3590780.

[9] MITRE, *Cwe list version 4.14*, 2024. [Online]. Available: https://cwe.mitre.org/data/index.html.
[10] MITRE, *Cwe top 25 most dangerous software weaknesses*, 2023. [Online]. Available: https://cwe.mitre.org/top25/.
[11] MITRE, *Cve program mission*, 2024. [Online]. Available: https://www.cve.org/.
[12] M. D. Ernst, 'Static and dynamic analysis: Synergy and duality', in *WODA 2003: Workshop on Dynamic Analysis*, Portland, OR, USA, May 2003, pp. 24–27.
[13] O. Zaazaa and H. El Bakkali, 'Dynamic vulnerability detection approaches and tools: State of the art', in *2020 Fourth International Conference On Intelligent Computing in Data Sciences (ICDS)*, 2020, pp. 1–6. DOI: 10.1109/ICDS50568.2020.9268686.
[14] A. Aggarwal and P. Jalote, 'Integrating static and dynamic analysis for detecting vulnerabilities', in *30th Annual International Computer Software and Applications Conference (COMPSAC'06)*, vol. 1, 2006, pp. 343–350. DOI: 10.1109/COMPSAC.2006.55.
[15] P. P. Ray, 'Chatgpt: A comprehensive review on background, applications, key challenges, bias, ethics, limitations and future scope', *Internet of Things and Cyber-Physical Systems*, vol. 3, pp. 121–154, 2023, ISSN: 2667-3452. DOI: https://doi.org/10.1016/j.iotcps.2023.04.003. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S266734522300024X.
[16] M. D. Purba, A. Ghosh, B. J. Radford and B. Chu, 'Software vulnerability detection using large language models', in *2023 IEEE 34th International Symposium on Software Reliability Engineering Workshops (ISSREW)*, 2023, pp. 112–119. DOI: 10.1109/ISSREW60843.2023.00058.
[17] OpenAI, *Models: O1-preview and o1-mini*, 2024. [Online]. Available: https://platform.openai.com/docs/models/o1.
[18] B. Rozière, J. Gehring, F. Gloeckle *et al.*, *Code llama: Open foundation models for code*, 2024. arXiv: 2308.12950 [cs.CL].
[19] M. Fu, C. K. Tantithamthavorn, V. Nguyen and T. Le, 'Chatgpt for vulnerability detection, classification, and repair: How far are we?', in *2023 30th Asia-Pacific Software Engineering Conference (APSEC)*, 2023, pp. 632–636. DOI: 10.1109/APSEC60848.2023.00085.
[20] H. Li, Y. Hao, Y. Zhai and Z. Qian, 'Assisting static analysis with large language models: A chatgpt experiment', in *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2023, San Francisco, USA: Association for Computing Machinery, 2023, pp. 2107–

2111, ISBN: 9798400703270. DOI: 10.1145/3611643.3613078. [Online]. Available: https://doi.org/10.1145/3611643.3613078.
[21] C. Zhang, H. Liu, J. Zeng, K. Yang, Y. Li and H. Li, *Prompt-enhanced software vulnerability detection using chatgpt*, 2024. arXiv: 2308.12697 [cs.SE].
[22] C. Li, J. Wang, Y. Zhang *et al.*, *Large language models understand and can be enhanced by emotional stimuli*, 2023. arXiv: 2307.11760 [cs.CL]. [Online]. Available: https://arxiv.org/abs/2307.11760.
[23] Y. Chen, Z. Ding, L. Alowain, X. Chen and D. Wagner, *Diversevul: A new vulnerable source code dataset for deep learning based vulnerability detection*, 2023. arXiv: 2304.00409 [cs.CR]. [Online]. Available: https://arxiv.org/abs/2304.00409.
[24] L. D. Group, *Extra clang tools 20.0.0git documentation: Clang-tidy*, 2023. [Online]. Available: https://clang.llvm.org/extra/clang-tidy/.
[25] D. Marjamäki, *Cppcheck: A tool for static c/c++ code analysis*, 2023. [Online]. Available: https://cppcheck.sourceforge.io/.
[26] D. A. Wheeler, *Flawfinder*, 2021. [Online]. Available: https://dwheeler.com/flawfinder/.
[27] M. Bhatt, S. Chennabasappa, C. Nikolaidis *et al.*, *Purple llama cyberseceval: A secure coding benchmark for language models*, 2023. arXiv: 2312.04724 [cs.CR]. [Online]. Available: https://arxiv.org/abs/2312.04724.
[28] Gemini Team *et al.*, *Gemini: A family of highly capable multimodal models*, 2024. arXiv: 2312.11805 [cs.CL].
[29] Anthropic, *Introducing the next generation of claude*, 2024. [Online]. Available: https://www.anthropic.com/news/claude-3-family.

CHAPTER 7

Appendices

7.1 Appendix A: Original Project Proposal

Using LLMs for Vulnerability Checking

7.1.1 Introduction

In recent years, large language models (LLMs) such as OpenAI's ChatGPT have become increasingly popular as a tool to generate and refine code. While this trend has come with many benefits to efficiency and ease of use, there is a concern that code produced by LLMs is prone to containing security vulnerabilities. Khoury et al. [1] investigated the security of code generated by ChatGPT and found that even when specifically prompted to produce secure code, vulnerabilities were still common. However, ChatGPT was reasonably effective in identifying these vulnerabilities when the insecure code was fed back into it and was sometimes able to revise the code to fix vulnerabilities [1]. These findings show that the correct identification of vulnerabilities in code is an important step in using LLMs to generate secure code. For this reason, we will investigate the viability of using LLMs to detect code vulnerabilities.

Existing tools such as static and dynamic analysers already offer vulnerability-detecting capabilities. Static analysers scan through source code without running it while dynamic analysers run the code and report back its effects. However, these tools are language-specific and vary in effectiveness, with static analysers often failing to detect vulnerabilities in real-world cases [2]. These tools are also unable to detect vulnerabilities that arise from unexpected interactions between different code segments or systems as there is no way to input contextual knowledge describing how the code will be used.

LLMs have already shown promise in the field of vulnerability detection.

Ozturk et al. [2] found that ChatGPT was significantly better at vulnerability detection for PHP code compared to various open-source static analysis tools. Using LLMs for vulnerability detection also solves some of the other issues mentioned above. Code segments from any language can be inputted into an LLM as part of the prompt, making them much more versatile than other tools. Additionally, usage context can be included in prompts and may allow LLMs to identify vulnerabilities caused by interacting systems, although this is beyond the scope of our research.

7.1.2 Aims

The primary goal of this project is to determine if LLMs can match or exceed the vulnerability-checking performance of existing methods. We will build on the work of Ozturk et al. [2] by comparing various LLMs to a range of both static and dynamic analysis tools in multiple programming languages. As part of this comparison, our secondary goal will be to investigate how prompts can be used to maximise LLM performance when checking code security. This will help ensure that the true capabilities of LLMs are represented in the comparison.

We will also consider frequency of false positives for each tool, which occur when a tool identifies vulnerabilities that are not present in the code. Ozturk et al. [2] found that ChatGPT had a very high rate of false positives compared to other static analysers tested. False positives are undesirable as they waste the time of human programmers and cause issues for LLM-automated code generation due to incorrect information being fed into prompts. For this reason, our investigation of prompt choice will factor in the effect that prompts have on the rate of false positives. We will aim to find prompts that maximise vulnerability detection rate while minimising the occurrence of false positives.

7.1.3 Methodology

Each tool will be tested with two datasets containing examples of secure and insecure code respectively. Challenges relating to acquiring this dataset are described in the next section. For the static and dynamic analysers, every compatible code example will be run through the analyser, and the outcome will be recorded. The test case is recorded as a success if the tool correctly identifies all vulnerabilities present or correctly identifies that no vulnerabilities are present. If the tool fails to identify a vulnerability or identifies a vulnerability that is not present, the test case is recorded as a failure. From

these results, we will be able to determine the successful detection rate for each tool, as well as the types of vulnerabilities that the tools performed well or poorly on.

The same process will be used to test each LLM but will be repeated for each prompt variation. A few potential prompt variations are given below:

1. "Find any vulnerabilities in the following code."

This prompt represents the base case and acts as a performance baseline. By comparing the results of other prompts to this one, the degree to which prompt choice impacts LLM performance in this task can be determined.

1. “Check the following code for any vulnerabilities from the 2023 CWE Top 25 Most Dangerous Software Weaknesses.”

This prompt is intended to investigate the effect of providing the LLM more focused information. While the LLM may be more likely to detect a vulnerability that is included in the list, it might also be more prone to false positives for vulnerabilities that appear in the list but aren’t present in the code.

1. “Check the following code for any integer overflow vulnerabilities.”

This prompt represents the ideal scenario. If an LLM is unable to detect the vulnerability when directly told which one to look for, this likely means that it is not capable of detecting the vulnerability regardless of prompt choice. Results of this prompt will not be directly compared to that of the other tools as the prompt requires that the LLM already knows which vulnerabilities are present.

All results will be split by type of vulnerability to determine which vulnerabilities were easiest to detect and which were most difficult over all tools. These results will be compared between each tool to conclude if any tools differed significantly from the overall trend. The LLM results will be analysed as a whole, as well as split by programming language. This will allow results from program-specific tools to be directly compared to the LLM results and will show if the LLMs performed differently depending on the programming language.

7.1.4 Requirements

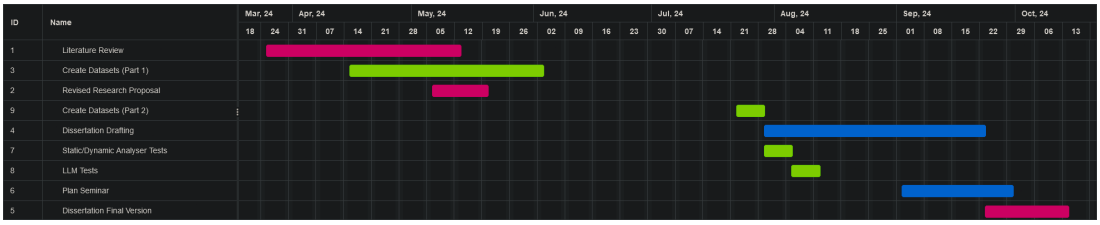
The main challenge in our research is the need for a dataset containing examples of insecure code, labelled with the vulnerabilities they contain. As described in the previous section, this kind of dataset will be used as a benchmark for all the code analysis tools that we will investigate. A dataset of this

description is not readily available, so we will have to create our own. This will require us to source code snippets various locations or write our own examples. It is important that the dataset includes an even spread of vulnerabilities and contains code from multiple languages. This will avoid biasing the results in favour of tools that perform better on certain types of vulnerability and will allow us to compare different language-specific tools.

To test the frequency of false positives, a secondary dataset containing examples of secure code will also be created. These examples should include potentially dangerous functions or program types that are handed correctly rather than trivially secure code that has no chance of being vulnerable, as the former is most likely to be flagged as a false positive. Sourcing examples of secure code shouldn't be an issue as the manuals for most vulnerable functions include examples of the function being used correctly.

Additionally, it may be worthwhile to purchase access to LLMs that are not freely available, such as ChatGPT-4. These LLMs showcase the current limits of the technology and if they perform significantly better than the free LLMs and other tools, this would show that the technology has potential in the area of vulnerability detection. In addition, the rapid development of the LLM field means that the performance of these paid LLMs will likely be matched by free LLMs in the near future.

7.1.5 Time Plan



Notes:

- The 'Create Datasets' task was separated into two parts to avoid including the semester break.
- The 'Dissertation Drafting' task will include multiple draft versions, but these were not separated in the chart as it is difficult to predict how many drafts will be written until the testing phase begins.
- The 'Seminar Planning' task includes the writing of the abstract and any preparation of the presentation itself.

7.1.6 References

- R. Khoury, A. R. Avila, J. Brunelle, and B. M. Camara, 'How Secure is Code Generated by ChatGPT?' arXiv:2304.09655 [cs], Apr. 2023.
- O. S. Ozturk, E. Ekmekcioglu, O. Cetin, B. Arief, and J. Hernandez-Castro, "New Tricks to Old Codes: Can AI Chatbots Replace Static Code Analysis Tools?" in Proceedings of the 2023 European Interdisciplinary Cybersecurity Conference, A. Mileva, S. Wendzel, and V. Franqueira, Eds., New York, NY, USA: Association for Computing Machinery, Jun. 2023, pp. 13–18. Accessed: Mar. 14, 2024. [Online]. Available: https://dl.acm.org/doi/10.1145/3590777.3590780