Benchmarking Large Language Models for Vulnerability Detection: Comparing Local and Cloud LLMs

Karl Müller-Uri & Alexandra Pykälistö
karl.mueller.uri@gmail.com & pykalistoalexandra@gmail.com

Department of Electrical and Information Technology
Lund University

Supervisor LTH: Christian Gehrmann
Supervisors Advenica: Marcus Kindberg & Victor Arvidsson
Examiner: Thomas Johansson

30th May 2026

Abstract

The number and complexity of modern software systems have increased substantially since their inception. This unrelenting growth is making manual code review more difficult year by year, and the gap between the amount of code and security verification creates a critical need for automated tools that can assist developers in detecting vulnerabilities. A possible solution to this is large language model driven vulnerability detection.

This thesis investigates the possibility of utilizing locally fine-tuned LLMs in order to discover and flag memory related security flaws in C/C++ code. Five locally fine-tuned models have been examined and compared to each other, their non-fine-tuned versions, as well as proprietary cloud models. The models were fed functions taken from C/C++ projects, and were asked to determine whether the function in question was vulnerable. Two different prompting methods were used during the evaluation, zero-shot prompting and few-shot prompting. After each evaluation, performance metrics such as accuracy and F_1-score were calculated.

While fine-tuning enhanced the performances of the local models with respect to F_1-score, their ability to detect vulnerabilities remained unsatisfactory. The highest performing model, CodeLlama 7B, achieved a F_1-score of only 0.12. Cloud models are orders of magnitude larger in parameter size and have had more extensive pre-training. However, as the cloud models did not outperform this, it indicates that the methods utilized in the thesis were sub-optimal. The two prompting methods did not significantly impact the results of any model. Further research to improve model performance may include Chain-of-Thought prompting, Retrieval-Augmented Generation, or fine-tuning of the cloud models.

Popular Science Summary

Can Locally Fine-Tuned LLMs Win Against Big Cloud Models in the Race to Find Software Vulnerabilities?

With the rising usage of software in the world, the attack surface has increased in tandem. Therefore, the importance of secure software has grown. This thesis investigates how locally fine-tuned large language models (LLMs) compare to cloud-based models for vulnerability detection in C/C++ code. Can a local, fine-tuned LLM specialized in vulnerability detection outperform a cloud-based LLM?

In an era where software governs everything from your social media to national security, a single coding error can have catastrophic consequences. From the infamous Heartbleed Bug to recent global IT disruptions, the message is clear: we need better ways to find and fix vulnerabilities before they are exploited. Traditionally, this required intensive manual code review, but a new generation of artificial intelligence is emerging.

This study investigated an important topic for the future of cybersecurity. It compared larger cloud LLMs such as ChatGPT against smaller, open-source models such as Mistral. While cloud models have immense processing power, local models offer a significant advantage: data privacy. For a cybersecurity firm or a government agency, sending sensitive, proprietary code to a third-party cloud provider is often a deal-breaker. But in order to match the capabilities of the enormous cloud models, the local LLMs need to be specialized into advanced vulnerability detection models. This was achieved through a powerful tool: Fine-tuning.

What is Fine-Tuning?

Large language models can have different sizes, which is usually measured in the parameter count of the model. If one wants to give the relatively small, local, billion parameter-LLMs a fighting chance against the trillion parameter-cloud LLMs, the local models need to be specialized. This is achieved by way of presenting the local models with example tasks, and gradually updating the parameters of said models to enhance their performance. This process is known as fine-tuning.

The Surprising Results

The outcome of the battle was unexpected: one of the locally fine-tuned models actually outperformed the cloud giants. The local LLM CodeLlama 7B emerged as the top performer, achieving the highest F_1-score of all models. The cloud models, despite their size, struggled with the tasks presented before them. Furthermore, fine-tuning provided the local LLMs with essential knowledge of the formatting of the desired output, as the local models were unable to structure their answers correctly without it.

While locally fine-tuned models won, the study found that they are not quite ready for real world application within cybersecurity. Their overall score remained low, and they showed a bias toward assuming code was safe.

Abbreviations

AI	Artificial Intelligence
API	Application Programming Interface
CoT	Chain-of-Thought
CSV	Comma-Separated Values
CWE	Common Weakness Enumeration
DAPT	Domain Adaptive Continual Pre-Training
FFT	Full Fine-Tuning
FN	False Negative
FNR	False Negative Rate
FP	False Positive
FPR	False Positive Rate
IFT	Instruction Fine-Tuning
JSON	JavaScript Object Notation
LLM	Large Language Model
LoRA	Low-Rank Adaptation
NLP	Natural Language Processing
PEFT	Parameter Efficient Fine-Tuning
PPV	Positive Predictive Value
QLoRA	Quantized Low-Rank Adaptation
RAG	Retrieval Augmented Generation
SFT	Supervised Fine-Tuning
TN	True Negative
TNR	True Negative Rate
TP	True Positive
TPR	True Positive Rate

Contents

List of Figures

List of Tables

Chapter 1

Introduction

This chapter outlines the motivation, purpose and research questions for this thesis. It explores the importance of the subject and context of the field. Additionally, it details the division of labor during the work process and provides background information regarding Advenica.

1.1 Advenica

This thesis is performed at Advenica in close collaboration with LTH Faculty of Engineering at Lund University [48]. Advenica is a Swedish cybersecurity company specializing in customized solutions for cybersecurity. Their products are primarily designed and manufactured in-house at their office in Malmö, Sweden. Advenica's main focus is providing governments and corporations with secure cybersecurity products within the fields of cryptography and network segmentation [2].

Secure code is of utmost importance for a cybersecurity company trusted with Sweden's national security. This thesis explores the potential of using artificial intelligence (AI) [76] as a tool for detecting security vulnerabilities in code, with local LLMs being a high priority, due to the risks of relying on cloud based LLMs for vulnerability detection on sensitive or classified code.

1.2 Motivation

Vulnerability detection plays an important role in the development and maintenance of software [19]. As the presence of software increases, the attack

surface increases in turn, and therefore also the risk of considerable damage being done [17]. The complexity of modern software systems has grown substantially, making manual code review increasingly difficult. The gap between the amount of code and security verification creates a critical need for automated tools that can assist developers in more effectively and reliably detecting vulnerabilities [84].

Large language models (LLMs), such as GPT [63] and Gemini [78], are advanced types of AI which have been trained on a large amount of data to understand, generate, and process human language [45]. The development of LLMs has advanced significantly in recent years [51]. A particularly interesting application of this technology is the automated detection of security vulnerabilities within code.

A widely publicized case that highlights the necessity of such tools is the Heartbleed Bug from 2014. This vulnerability allowed sensitive data to be exfiltrated from protected servers due to a fundamental coding error [16] [38]. While the specific bug was patched in April 2014, it remains an important example of the catastrophic impact a small error can have.

In July 2024, a decade following Heartbleed, the CrowdStrike incident highlighted a different type of threat. This was not a threat caused by malicious actors exploiting an underlying vulnerability, but rather a coding error introduced by a faulty configuration update affecting Microsoft Windows [59]. The CrowdStrike outage is considered one of the largest IT disruptions in history, as it caused surgeries and flights to be canceled while simultaneously disrupting global banking operations [58].

Since security risks like these remain a constant problem, there is a critical need for a robust vulnerability detection and the minimization of human error in software development. This thesis aims to investigate the use of LLMs of varying sizes for identifying security flaws, with the purpose of keeping sensitive data secure.

1.3 Thesis Purpose

The purpose of this thesis is to compare larger cloud LLMs against smaller, local LLMs for vulnerability detection, after fine-tuning of the smaller models on specific common weakness enumerations (CWEs) [25]. This thesis focuses on memory related vulnerabilities within C/C++ code. Evaluation is performed using the prompting methods zero-shot prompting and few-

shot prompting. Prompting is the process of guiding an AI toward a specific desired outcome and a strategic prompt structure can significantly enhance model performance [9]. Therefore, prompt engineering is a core component of this study for evaluating the LLMs.

Zero-shot prompting is the most simple prompting method where the user provides a task without any context, whereas few-shot prompting provides standalone examples of the wanted output from the given input [14] [55].

1.4 Research Questions

The research questions that are examined throughout this thesis are listed below. The focus is on fine-tuning local models to compare against cloud models.

- **RQ1**: How do locally fine-tuned LLMs compare to cloud LLMs for vulnerability detection in terms of accuracy, F_1 score, precision and recall?
- **RQ2:** How do the different locally fine-tuned LLMs compare to each other for vulnerability detection in terms of accuracy, F_1-score, precision and recall?
- **RQ3:** Does fine-tuning increase the performance of local LLMs relative to their original pre-trained version for vulnerability detection in terms of accuracy, F_1 score, precision and recall?
- **RQ4**: How do the different prompting methods (zero-shot and few-shot) compare in terms of speed, resource usage and F_1 score?

These research questions were developed during the planning phase of this thesis in collaboration with the supervisors at Advenica. While the primary objective is summarized in the first research question, the rest emerged during the implementation of the methodology to provide critical perspectives for further investigation. The significance of this research lies in evaluating whether local models can achieve a performance standard that justifies their use over cloud-based models for vulnerability detection.

Thus, several locally fine-tuned models are compared against one another, their respective base versions, as well as cloud-based models. This comparison aims to determine whether local models can meet the performance standards of cloud-based solutions and to identify which specific architectures and fine-tuning strategies yield the most effective results.

1.5 Division of Labor

Activity	Müller-Uri (%)	Pykälistö (%)
Literature Review	40	60
Development	60	40
Testing	50	50
Evaluation	50	50
Report	40	60
Presentation	50	50
Popular Science Summary	60	40

Table 1.1: Division of labor between Müller-Uri and Pykälistö.

Table 1.1 summarizes the division of labor between the two Master Thesis students Karl Müller-Uri and Alexandra Pykälistö.

Chapter 2

Technical Background

This chapter presents the technical background for the thesis, specifically examining large language models and various prompting methods. It also defines the evaluation metrics used to measure performance.

2.1 Large Language Models (LLM)

Modern language models trace back to Shannon's 1951 study [72], which used n-gram statistics to demonstrate that natural language is highly predictable. He showed that the subsequent letter of a given n-gram of letters was highly predictable. The goal was to predict and model natural language, which is the precursor to the modern field of natural language processing (NLP) [18] and the development of LLMs. While early language models in the 1990s showed limited results, the field has since evolved into large language models by scaling up training datasets and increasing the number of parameters [45]. Additionally, there has also been a structural change within the models since then [81]. This has allowed for specialized applications across various fields.

The focus in this thesis is on two different ways to run AI: cloud-hosted via an API, and local models. The benefits of local models include greater customization and data privacy [7]. However, local models suffer from limitations in performance compared to cloud LLMs when not fine-tuned for a specific task [50].

During this thesis a comparison is therefore made between smaller open-source LLMs that can run locally and be fine-tuned, and larger proprietary LLMs that are run in the cloud.

2.1.1 Local LLMs

Local LLMs are locally hosted models. For this thesis, these models are open-source with a manageable number of parameters, which enables the LLM to be controlled and edited locally in a reasonable amount of time [12]. Locally hosted LLMs are advantageous when a secure environment is required and non-local alternatives cannot be trusted. These models also offer replicability [8], which is of importance in this thesis. To improve the performance of locally hosted models, fine-tuning is usually implemented [8].

This thesis evaluates three distinct local models, which are not specialized for code knowledge: Llama 3.2-3B [46], Mistral 7B [52] and Qwen3-1.7B [66]. For this thesis, each model is selected based on the requirement that its number of parameters could not exceed 40 billion, due to computational limits of the provided hardware. In practice, however, the fine-tuning process limits the parameter count of the models to considerably less than 40 billion.

Llama 3.2 is the newest member of the Llama family that fits the above criterion, with 3 billion parameters. It can be customized, which makes it a candidate for this thesis [46]. Advantages stated with the 2024 release include fast responses and maintaining privacy since there is no connection to the cloud [46].

Mistral 7B is intended for fine-tuning on any task. Mistral contains 7 billion parameters [52].

Qwen3-1.7B has 1.7 billion parameters. This model was developed for local applications through sites as Ollama [60]. A larger model with 4 billion parameters was initially chosen for fine-tuning but the used hardware is not able to evaluate it.

Two other local models trained for code are taken into consideration to compare with the first three general LLMs, CodeLlama-2B [41] and DeepSeek-Coder-6.7B [27]. CodeLlama-2B has 2 billion parameters while DeepSeek-Coder-6.7B has 6.7 billion parameters.

For all local models used in this thesis, they can be accessed through HuggingFace [40] and incorporated into the same fine-tuning process. Larger parameter sizes would have been preferred for this thesis, however due to hardware limitations it is not possible to go above the chosen models.

2.1.2 Cloud LLMs

Larger, closed-source LLMs are restricted, as users cannot download the full model or its weights [55] [64]. As a result of this, these LLMs have to be hosted in a cloud environment [30]. Instead, access is managed through an application programming interface (API) [55] [64]. Since cloud models are not public their structure is unrevealed [55].

This thesis investigates two prominent cloud based models: ChatGPT-5.4 [42] and Claude Opus 4.7 [6]. Both are, as of writing this thesis, the latest releases from their respective developers and among the leading large language model providers [3] [47] [71]. Neither Claude Opus 4.7 nor GPT-5.4 have released the number of parameters of their models.

GPT-5.4, the most recent iteration of the GPT family, was released in December 2025, and is accessible via OpenAI's API [82]. According to OpenAI [61], this version significantly outperforms its predecessors, particularly in agent coding which is a development cited as a major leap for the field. Additionally, OpenAI claims that the model has shown robust performance in long context analysis and extended reasoning tasks [82].

Claude Opus 4.6 was released in early February 2026 by Anthropic [4], making it a highly relevant candidate for this evaluation. Recent benchmarks provided by Anthropic indicate that Claude Opus 4.6 maintains an unmatched position in LLM-based vulnerability detection [15]. However, during implementation in the middle of April 2026, the newer version Claude Opus 4.7 was released [6]. Consequently, the focus switched to the newer version, as this would not affect the implementation time nor research questions.

At the time of writing, reviews of GPT-5.4 and Claude Opus 4.7 have yet to be published due to their recent release dates. As such, the available sources may contain biases.

2.2 Fine-Tuning

Fine-tuning involves an additional training of an already pre-trained model on a smaller, task-specific dataset to enhance its performance on a target application, in this case vulnerability detection in C/C++ code. A primary advantage of this approach is the achievement of high performance while significantly reducing the amount of training data required in comparison to

training from scratch [64]. A notable limitation is the need for a unique dataset for every specialized task the LLM is intended to perform [14].

The fine-tuning process updates the parameter weights of the pre-trained LLM, and by manually updating and altering specific configurations known as hyperparameters between fine-tuning rounds, different results may be achieved. Optimizing these parameters is essential for maximizing the models performance. While this often begins with a trial and error approach, it eventually moves toward a more systematic optimization [65].

While fine-tuning is typically implemented on local models, similar techniques exist for cloud based LLMs as well. For instance, OpenAI offers supervised fine-tuning (SFT), a process that allows users to develop highly customized and specialized versions of the model [77]. Fine-tuning of cloud based LLMs will not be implemented during this thesis, since the focus is to compare the original cloud based models to the fine-tuned local models.

Table 2.1 reproduced from Parthasarathy et al. [64] summarizes the differences between pre-trained models and applying fine-tuning.

2.3 Low-Rank Adaptation (LoRA)

During the fine-tuning process of LLMs, a technique known as Low-Rank Adaptation (LoRA) is often implemented. LoRA is a form of parameter efficient fine-tuning (PEFT) [33]. In short, LoRA freezes the pre-trained model weights and introduces a separate set of weights via lower-rank matrices. This approach significantly reduces the number of trainable parameters, as the original weights remain unchanged while only the lower-dimensional adapters are updated. Consequently, implementing LoRA accelerates the fine-tuning process and substantially reduces the memory required for training [39] [64].

Quantized LoRA (QLoRA) is an extension of the LoRA technique that further reduces memory requirements. While standard models typically utilizes 32-bit precision, QLoRA quantizes the pre-trained model weights to 4-bit precision [64]. This allows for fine-tuning of significantly larger models on the hardware with minimal impact on performance. By combining 4-bit quantization with LoRA adapters, QLoRA substantially lowers the VRAM usage compared to standard LoRA, making the fine-tuning process more accessible and resource efficient [28].

Aspect	Pre-training	Fine-tuning
Definition	Training on a vast amount of unlabelled text data	Adapting a pre-trained model to specific tasks
Data Requirement	Extensive and diverse unlabelled text data	Smaller, task-specific labelled data
Objective	Build general linguistic knowledge	Specialise model for specific tasks
Process	Data collection, training on large dataset, predict next word/sequence	Task-specific data collection, modify last layer for task, train on new dataset, generate output based on tasks
Model Modification	Entire model trained	Last layers adapted for new task
Computational Cost	High (large dataset, complex model)	Lower (smaller dataset, fine-tuning layers)
Training Duration	Weeks to months	Days to weeks
Purpose	General language understanding	Task-specific performance improvement
Examples	GPT, LLaMA 3	Fine-tuning LLaMA 3 for summarisation

Table 2.1: Comparison between pre-training and fine-tuning processes in LLMs reproduced from Parthasarathy et al. [64].

2.4 Prompting Techniques

Prompting is the process of directing a large language model toward a specific output by providing structured inputs or instructions [11] [55]. Prompting techniques have been used as an alternative to fine-tuning for larger models, since fine-tuning can be time consuming. Prompting requires significantly less time [11] [83]. The two prompting techniques that are utilized in this thesis are zero-shot prompting and few-shot prompting.

2.4.1 Zero-Shot Prompting

Zero-shot prompting is the most basic form of instructing an LLM. The user provides a task without any examples or demonstration of the desired result [14] [55]. Achieving high quality outcomes with this technique can be challenging, as there is no indication of how the output should be structured or presented beyond the initial task description [14].

2.4.2 Few-Shot Prompting

Few-shot prompting is similar to zero-shot prompting in that it defines a specific task for the LLM. However, the primary difference is that few-shot includes a few examples of the intended output. These examples serve as demonstrations of the desired output format and style [14] [55].

One advantage of few-shot prompting is the reduced amount of specific data that would have been needed to fine-tune the LLM into a similar task. The trade-off however, is that few-shot performance generally does not reach the results of a fully fine-tuned LLM [14].

2.5 Evaluation Metrics

This section will explain the metrics used during evaluation to compare the models.

2.5.1 Confusion Matrix

The confusion matrix is essential for evaluating large language models. It is a square matrix of size $N \times N$, where N is the number of output classes. The matrix provides insight into the number of correct and incorrect classifications, where one axis represents the actual class (ground-truth), and the other axis represents the predicted class. The predicted values can either be correct or incorrect. This is later used to direct the course of action to enhance the performance of the model [70].

Table 2.2 is an example of a confusion matrix that will be used. In this thesis, a vulnerable code snippet is considered to be in the positive class, and the benign code is considered negative.

A true positive is a positive example that has been correctly classified as positive, while a false positive is a negative example that has been incorrectly

		Predicted Values	
		Positive	**Negative**
Actual Values	**Positive**	**T**rue **P**ositive	**F**alse **N**egative
	Negative	**F**alse **P**ositive	**T**rue **N**egative

Table 2.2: Confusion Matrix.

classified as positive. Similarly, a true negative is a negative example that has been correctly classified as negative, while a false negative is a positive example that has been incorrectly classified as negative [35] [70].

In the confusion matrix, TP is the number of true positives, FP is the number of false positives, TN the number of true negatives, and FN the number of false negatives [35] [70].

True positive rate (TPR) measures a model's ability to correctly identify all positives, also known as recall. TPR is shown in Equation 2.1 [35].

$$\text{True Positive Rate (TPR)} = \frac{TP}{TP + FN} \tag{2.1}$$

False positive rate (FPR) represents the proportion of how actual negative cases has been marked as positive. FPR is seen in Equation 2.2 [35] [70].

$$\text{False Positive Rate (FPR)} = \frac{FP}{FP + TN} \tag{2.2}$$

The true negative rate (TNR), also known as specificity, is the proportion of actual negatives that have also been correctly identified as such. TNR is shown in Equation 2.3 [35].

$$\text{True Negative Rate (TNR)} = \frac{TN}{TN + FP} \tag{2.3}$$

False negative rate (FNR) helps measure how many real vulnerabilities are identified. A lower FNR is expected when the model can correctly mark positives as such. The formula for FNR is listed in Equation 2.4 [31].

$$\text{False Negative Rate (FNR)} = \frac{FN}{FN + TP} \tag{2.4}$$

2.5.2 Accuracy

Accuracy refers to the fraction of correct predictions, both positives and negatives, out of all predictions [70]. A high accuracy is desirable when evaluating a model. It is represented in Equation 2.5.

$$\text{Accuracy} = \frac{TP + TN}{TP + TN + FP + FN} \tag{2.5}$$

This is beneficial to use when classes are balanced in the dataset that is being used, which means that there is approximately equal amounts of positives and negatives [70].

However, limitations with accuracy occur for imbalanced datasets. A high accuracy can then be achieved by only predicting the majority class, which is not a desired result [70].

2.5.3 Precision

Precision is useful when it is important to reduce false positives. Precision represents the fraction of the correctly predicted positives, which can be seen in Equation 2.6.

$$\text{Precision} = \frac{TP}{TP + FP} \tag{2.6}$$

A limitation with precision is that it does not evaluate performance in regards to the negative cases, since precision does not take false negative into account [70]. Precision is also known as Positive Predictive Value (PPV) [53].

2.5.4 Recall

Recall is another name for TPR and measures how accurately the model is able to predict positive cases. It is presented in Equation 2.7 [70].

$$\text{Recall} = \frac{TP}{TP + FN} \tag{2.7}$$

Recall is prioritized when it is essential to minimize the number of false negatives, as when recall is low, the model is more prone to missing positive cases [70].

2.5.5 F_1-score

The F_1-score is favorable when the precision and recall are equally important. This is illustrated in Equation 2.8, where it can be calculated as the harmonic mean of precision and recall. Typically, a trade-off exists between recall and precision, whereby an increase in one of them usually leads to a decrease in the other [70].

$$F_1 = 2\,\frac{\text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}} = \frac{2TP}{2TP + FP + FN} \tag{2.8}$$

Since it is a combination of precision and recall, F_1-score takes both false negatives and false positives into account. Therefore, it is preferable when the datasets are imbalanced [70].

The F_1-score can be viewed as a special case of the F_β-score for when $\beta = 1$, where the F_β-score is calculated as

$$F_\beta = \frac{\beta^2 + 1}{\left(\beta^2 \cdot \text{Recall}^{-1}\right) + \text{Precision}^{-1}} = \frac{(\beta^2 + 1) \cdot \text{Precision} \cdot \text{Recall}}{(\beta^2 \cdot \text{Precision}) + \text{Recall}}. \tag{2.9}$$

The F_β-score values the importance of recall as β times greater than the importance of precision [69].

2.6 Previous Research

This section reviews the current state of the field within the subject of AI for vulnerability detection in cybersecurity, different prompting methods for open and closed-source models, and the significance of the choice of dataset.

2.6.1 Vulnerability Detection

The importance of LLMs for vulnerability detection has long been a topic of interest. This part reviews different studies focused on vulnerability detection.

In a study from 2024, Bhusal et al. [12] present a new framework for cybersecurity scenarios by the name of SECURE, an acronym for security, extraction, understanding, reasoning and evaluation. SECURE describes a method of testing LLMs in realistic cybersecurity applications, specifically as assistants for security analysts of a company. By utilizing OpenAI's ChatGPT-4o, six benchmark datasets were created, each with the goal of testing a specific skill (e.g. knowledge extraction, understanding) of the LLM being benchmarked. Although this specific benchmark is not used during this thesis, a particular interest was found in the seven LLMs which were evaluated: ChatGPT-4o, ChatGPT-3.5, Llama3-70B, Llama3-8B, Gemini-Pro, Mistral-7B and Mixtral-8x7B. For Llama3-8B, RAG and fine-tuning were implemented for performance evaluation. Bhusal et al. revealed that closed-source models such as ChatGPT-4o and Gemini-Pro stood out in comparison to the other LLMs. ChatGPT-4o had the highest results in 4 out of 6 tasks, with Gemini-Pro close behind, except on one task where it outperformed ChatGPT-4o. From the open-source models, Llama3-70B had the greatest performance, even though it did not reach the results of the closed-source models in all but one task [12].

In an article by Sheng et al. in 2025 [73], LLMs were evaluated for their effectiveness in vulnerability detection. A literature review was conducted on 80 papers selected from an initial 500 papers published after 2019. The datasets reviewed indicated a majority of C/C++ datasets within the field. Furthermore, the review established that LLM-based vulnerability detection focuses primarily on C/C++, Java and Solidity. For C/C++, the focus was mainly on memory related vulnerabilities, as these are of the highest importance within these languages, due to the lack of automatic memory management. In accordance with this, memory related vulnerabilities is the main focus during this thesis.

Due to their larger model sizes, GPT and CodeLlama are the most popular models for these applications. The models in the review were categorized into encoder-only, encoder-decoder and decoder-only architectures. The ten most utilized LLMs in vulnerability detection were GPT-4o, GPT-3.5,

BERT, CodeBERT, CodeLlama, LLaMA, StarCoder, CodeT5, Mistral and GraphCodeBERT, with a majority of them being decoder-only categorized. The study also discusses the importance of detecting vulnerability in every commit, noting that the CVEfixes and Pan2023 datasets specifically contain commit level data. However, there is a lack of datasets containing repository level data which would better reflect realistic vulnerability code. As the surrounding context usually plays a part in what makes a piece of code vulnerable, its absence puts a bound on the efficacy of LLM-assisted vulnerability detection. Finally, accuracy and precision are mostly used for the evaluation of the models [73].

Target Language	**FT Method**	**Main Dataset**	**Model**	**F1-Score**
Solidity	PEFT	VulSmart	GPT-4o-mini	0.99
C/C++	FFT&PEFT	Devign	CodeLlama-7B	0.97
C/C++	DAPT	NVD	MPT-7B	0.93
C/C++	PEFT+IFT	SeVC	CodeLlama-13B	0.92
Solidity	PEFT	Ma2024	CodeLlama-13B	0.91
C/C++	PEFT	Liu2023	Gemma-7B	0.90
C/C++/Java	PEFT	Mixed Dataset	GPT-4	0.90
C/C++	PEFT	Zhang2024	Llama-7B	0.85
C/C++/Solidity	FFT	Big-Vul	CodeLlama-7B	0.82
HTML/JavaScript	FFT	Sakaolu	DistilRoBERTa	0.82
C/C++	FFT	Code Gadget	Davinci	0.73
C/C++	FFT+IFT	Devign	CodeLlama	0.71
Java	PEFT	CVEFixes	WizardCoder	0.71
C/C++	Model Innovation	QEMU	UnixCoder	0.71
C/C++	FFT	DiverseVul	BERT	0.69
C/C++	FFT	QEMU	UnixCoder	0.65
PHP	PEFT	RealVul	GPT-4	0.58
C/C++	FFT&PEFT	CVEFixes	NatGen	0.53
C/C++	FFT	Big-Vul	DeepSeek-Coder-6.7B	0.27
C/C++	FFT	PrimeVul	UnixCoder	0.21

Table 2.3: Comparison of fine-tuning methods and model performance from Sheng et al. [73] literature review.

Table 2.3 lists the results from the paper by Sheng et al. [73], ranked after the highest F1-score. Additionally, the programming language, fine-tuning method, dataset and model is in the table. The different fine-tuning methods were parameter efficient fine-tuning (PEFT), full fine-tuning (FFT), domain adaptive continual pre-training (DAPT) and instruction fine-tuning (IFT). Table 2.3 also provides insight into several datasets, some of which were considered for this thesis.

A paper by Shestov et al. [74] focused on fine-tuning LLMs for vulnerability detection in Java code using three StarCoder family models, StarCoderBase, StarCoder and WizardCoder. The authors found no significant performance difference between StarCoderBase and StarCoder, nor between WizardCoder and StarCoder. Therefore, the focus switched to the model WizardCoder, since no difference was found and WizardCoder is theoretically the better model. To fine-tune WizardCoder, LoRA was applied, which is a method from the Hugging Face library designed to reduce trainable parameters and memory requirements. Specifically, LORA reduced the number of trainable parameters from 13 billion to 25 million. The study utilized the CVEfixes, Manually-Curated and VCMatch datasets, with results evaluated using F_1 score and ROC AUC, which is a metric common for classifiers [35]. Fine-tuning improved both the ROC AUC and F_1 scores. These improvements were likely driven by WizardCoder's superior model size and its specific pre-training [74].

Key methodology have been adopted from Shestov et al., which includes the use of Low-Rank Adaptation (LoRA) [64] to reduce the number of trainable parameters. Additionally, their approach to fine-tune for vulnerability detection provides valuable insights, despite being originally applied to different programming languages and models than those examined in this thesis.

In a study by Widyasari et al. [84], a novel approach to vulnerability detection was employed, in which a court-like design was implemented when reviewing whether vulnerabilities are present in the code. The method, called VulTrial, included four agents as members of the court: (1) the Security Researcher Agent, which acted as a prosecutor, searching for vulnerabilities in the code, establishing facts and building a case for every vulnerability it finds; (2) the Code Author Agent, which acted as a defense attorney, presenting counter-arguments for the Code Author's claims; (3) the Moderator Agent, taking the role of the judge, whose primary role was ensuring fairness, avoiding any bias, and focusing on the facts presented to summarize the arguments made; and lastly, (4) the Review Board, acting as jury, taking everything into account by the other agents, before delivering the final verdict on whether the alleged vulnerability is present. VulTrial was tested on GPT-3.5 and GPT-4o, and showed an increased performance compared to the single-agent baseline, using techniques such as role-specific instruction tuning of the LLMs. Furthermore, the effectiveness of including a request to articulate the LLMs reasoning in the prompts [84].

Widyasari et al.'s findings on improved performance influence the choice to use longer, more descriptive evaluation prompts, even though this thesis does not replicate their exact experimental setup.

2.6.2 Prompting Techniques

Prompting techniques are critical to the evaluation of LLMs. Consequently, this thesis assesses the efficacy of different prompting methods, similarly to the following articles.

In a study by Nazi et al. [55] in 2025, different prompting methods, such as zero-shot, few-shot and CoT, were applied to both closed- and open-source LLMs. The evaluation focused on how these models handled low-resource languages across the four tasks: news classification, sentiment analysis, question answering and summarization [55]. Low-resource languages refers to spoken languages which are less studied and less commonly taught [49]. The closed-source models included GPT-3.5, and GPT-4o, while the open-source models were Aya 101, BLOOM 7b1 and LLaMA3 8B. F_1 score revealed that the open-source models performed better at classification, summarization and question answering than the closed models, while the closed-source models were better at reasoning. Performance in low-resource languages proved sensitive to the number of shots. Overall, GPT-4o was the best performing model, while LLaMA showed the most promise among open-source options when also fine-tuned. Regarding prompting, few-shots with three shots performed better than zero-shot. However, while CoT was superior in some cases, for others it led to the models reaching false conclusions based on faulty reasoning. The study included a suggestion for future work, namely, applying these three prompting methods to high-resource languages like English and Chinese [55].

The relevance of Nazi et al. is not in the low-resource languages, but rather in how the models were evaluated with the three different prompting methods, since two of them are of interest during this thesis.

Yildiz et al. [85], similarly applied four different prompting methods in 2025, to a new benchmarking dataset called JitVul. Researchers pinpointed the importance of having real-world examples of vulnerabilities, such as null pointer dereference, where it frequently emerges from complex sequences rather than a standalone function. To detect these, an understanding of the context is required. Just in time (JIT) vulnerability detection stands

out since it focuses exclusively on modified functions within a commit, while still taking the context into account. In the JitVul dataset, each commit that introduces a vulnerability is paired to a commit that fixes the vulnerability. It contains 879 CVEs with 91 different vulnerability types from the PrimeVul dataset. JitVul pairs the vulnerable function with the non-vulnerable function that has been fixed, and therefore has in total 1 758 paired commits. This study examines three vulnerability detection methods: Plain LLM, Dependency-Augmented LLM and ReAct Agent. Each method is evaluated using the four prompting strategies zero-shot, few-shot, CoT and a combination of CoT and few-shot. Using GPT-4o and GPT-4o-mini as underlying models, the evaluation revealed that GPT-4o-mini achieved its highest F_1 score when combined with dependency augmentation and CoT, while the highest pairwise accuracy score was a combination of ReAct Agent with few-shot prompting. In contrast, GPT-4o performed best in F_1 score using plain LLM with zero-shot prompting, whereas its top pairwise accuracy was achieved via the ReAct Agent with CoT. Pairwise accuracy measures the proportion of correctly labeled pairs [85].

JitVul is under consideration as a dataset used during this study. However, the main relevance from the article by Yildiz et al. lies in the prompting methods for evaluation.

2.6.3 Datasets with Vulnerabilities

When selecting the right dataset, several factors must be taken into account. In addition to this, there are often limitation in the datasets available for vulnerability detection.

Jensen et al. [43] examines the differences between datasets that are artificially generated and those built from code containing real-world or constructed vulnerabilities. Through a literature study of 19 different datasets and six generation tools, Jensen et al. evaluated their respective utility. They found that only five datasets, including PrimeVul [31], had been updated since July 2024, while the majority had remained unattended since their implementation. Furthermore, the study identified several limitations, such as incorrect labeling, class imbalance and issues regarding the complexity and quality of the code. There remains a significant inconsistency between vulnerability representation in these datasets compared to real-world scenarios, with technical issues like duplicate entries or files missing code entirely. A substantial challenge in generating vulnerable code is the shortage of available

tools, and the fact that existing options focus primarily on C/C++. These generators function by inserting vulnerabilities into existing code. However, there is limited knowledge regarding their overall effectiveness. Nevertheless, it is crucial to continue researching these generation systems. Given the documented limitations of existing datasets, automated generation may offer a solution. Jensen et al. conclude that further research is required to determine the trustworthiness of these tools before they can be fully adopted as a standard solution [43].

This article highlights the challenges of identifying a reliable dataset, a topic which is explored in greater detail in Chapter 3: Data Description.

2.6.4 Newly Released Models

During the course of this research, new cloud models have entered the market. This thesis specifically decided to evaluate Claude Opus 4.6 as a primary cloud model option in the early stages, given its specialized utility in cybersecurity contexts from its release.

Claude Opus 4.6. was released in early February 2026. In an article by Carlini et al. [15] published on Anthropic's website [68], the rapid pace of AI development for cybersecurity applications is discussed. Their methodology tested Claude in out-of-box scenarios without specialized prompting methods. Combined with this release, a new security measure was implemented to prevent data breaches by using Claude. Activation probes were used for the new security measure to estimate Claude's internal activity when generating a response to prevent harm related to cybersecurity. Claude Opus 4.6 has shown promise in reasoning similarly to human researchers by analyzing commit histories and utilizing pattern recognition. The model successfully identified 500 vulnerabilities, several of which had previously gone unnoticed. As Carlini et al. state, Claude Opus 4.6 provides superior vulnerability detection compared to prior LLMs [15].

During the implementation phase of this thesis, Claude Opus 4.7 was released [6]. Consequently, version 4.7 is selected as the primary cloud-based model from the Claude family over version 4.6. This shift exemplifies the rapid evolution of the field. Although Claude Opus 4.6 was Anthropic's most recent release in early February 2026, it was succeeded by the newer version only two months later, in mid-April 2026 [6].

Chapter 3

Data Description

This chapter details the evaluation and selection process of datasets used for fine-tuning and benchmarking. It also details the specific software vulnerabilities in C/C++ code used for the evaluation.

3.1 Vulnerabilities and Weaknesses

A common way of classifying weaknesses is using the common weakness enumeration (CWE) terminology [23]. Vulnerabilities are categorized under vulnerability types, or weaknesses, which in general terms classify the causes and underlying mechanisms of vulnerabilities in code [22]. The weaknesses are of the form CWE-X, where X is a number, together with a name, e.g. "CWE-125: Out-of-bounds Read". The CWEs can be found on the CWE website [25].

3.1.1 Common Weakness Enumerations (CWEs)

The focus of this thesis is the detection of memory related vulnerabilities in C/C++ code. To find relevant CWEs, the CWE website's list of Top 25 Most Dangerous Software Weaknesses from 2025 [21] was scanned for memory related weaknesses, and the following list was produced:

- **CWE-121**: Stack-based Buffer Overflow;
- **CWE-122**: Heap-based Buffer Overflow;
- **CWE-125**: Out-of-bounds Read;
- **CWE-476**: NULL Pointer Dereference; and
- **CWE-787**: Out-of-bounds Write.

In addition to these, two closely related CWEs are included: **CWE-119**: Improper Restriction of Operations within the Bounds of a Memory Buffer; and **CWE-120**: Buffer Copy without Checking Size of Input.

These CWEs can be largely categorized into two types, namely buffer overflow (CWE-119, CWE-120, CWE-121, CWE-122, CWE-125, and CWE-787) and null pointer dereference (CWE-476) [24] [25]. Buffer overflow occurs when operations are performed on a memory buffer, but reads or writes are issued outside of said buffer [24]. This is dangerous due to an unexpected ability to view or alter parts of memory which should normally be inaccessible. Null pointer dereference, in turn, occurs when an unexpectedly NULL pointer is dereferenced [25]. This can be used in denial of service attacks, since if a null pointer dereference occurs, the program usually stops or crashes entirely.

3.2 Datasets

This thesis considers different datasets, including SVEN [37], JitVul [85], Devign [87], CrossVul [56], Vulnerability Fix Dataset (VFD) [13], CVEfixes [10], PrimeVul [31], Big-Vul [34] and DiverseVul [17]. Various statistical quantities and comments of these datasets can be seen in Table 3.1.

To ensure a fair comparison between the local and cloud models, the benchmarking is to be performed on the same datasets, irrespective of model. Additionally, to minimize the risk of time travel or duplicate examples, as described by Ding et. al. [31], the local models should be fine-tuned on the same dataset on which the benchmarking occurs, with a chronological train-validation-test split of 80:10:10.

The Vulnerability Fix Dataset is disqualified as a candidate due to only containing Java code, contrary to what is stated in its documentation [13].

Dataset	Size*	# of CWEs**
SVEN [37]	1 606 (1 606)	9
JitVul [85]	1 758 (879)	91
Devign [87]	27 318 (12 460)	-
VFD [13]	70 000 (35 000)	-
CrossVul [56]	134 126 (6 884)	107
CVEfixes† [10]	20 060	127
PrimeVul [31]	235 768 (6 968)	140
Big-Vul [34]	264 919 (11 823)	91
DiverseVul [17]	330 492 (18 945)	150

*The number in parentheses is the number of vulnerable functions.
**A dash (-) indicates that the article did not clearly provide the number of CWEs.
†Figures include only the C/C++ functions and CWEs. The number of vulnerable functions was not clearly stated.

Table 3.1: Datasets evaluated during the project.

Though the SVEN [37] dataset is a human labeled dataset, which is highly desirable, it is also excluded due to only containing vulnerable code, which is not suitable for fine-tuning.

BigVul [34], CrossVul [56], and CVEfixes [10] are constructed from the records of the National Vulnerability Database, NVD, which is maintained by the National Institute of Technology, or NIST [57]. Examples are generated by crawling the database for commits, investigating when the vulnerability was patched, labeling the pre-commit version of the function as vulnerable, and the post-commit as fixed. DiverseVul [17] instead collects data from two bug-detection websites [75] [67], and aggregates these by a similar process as above. PrimeVul, in turn, was created from compiling BigVul [34], CrossVul [56], CVEFixes [10] and DiverseVul [17] into one database, after which normalization of formatting differences, de-duplication, and re-labeling of the functions followed. Because of this, the use of PrimeVul and any of the above mentioned datasets needs to be mutually exclusive, in order to prevent data leakage. Regarding JitVul, it was constructed from PrimeVul's dataset which also limits the choice to at most one of these two datasets [85].

Furthermore, as mentioned by Chen et. al. [17], the estimated positive predictive value (PPV) of Big-Vul and CrossVul are 25.0% and 47.8%. That is, given that an example is labeled vulnerable, the likelihood of said example being an actual vulnerability is less than 50%. These low values could pos-

sibly be caused by differing labeling methods and criteria for a function to be considered a vulnerability between studies. However, as data quality is of high importance, in this case to prevent an abundance of false positives, these datasets are disqualified. Similarly, the DiverseVul and Big-Vul datasets are also removed, as their PPVs were 51.7% and 60.0%.

The Devign [87] dataset has an estimated PPV of 80.0% according to Croft et. al. [20]. However, Ding et. al. [31] estimates the PPV of the dataset to be 25.0%, despite the dataset being manually labeled. Due to this, the Devign dataset is rejected.

Thus, JitVul and PrimeVul emerges as the remaining candidates. As PrimeVul is two orders of magnitude greater in size than JitVul, it is selected as the prime candidate, due to its size, its high estimated positive predictive value (up to 92.0%), and more sophisticated de-duplication process, resulting in an estimated 0.0% duplicate entries [31].

3.3 PrimeVul

As mentioned in the previous section, the PrimeVul [31] dataset is constructed from the four datasets BigVul [34], CrossVul [56], CVEFixes [10] and DiverseVul [17]. It consists of 235 768 examples in the form of code functions, of which 228 800 are labeled benign and 6 968 are labeled vulnerable. Due to incomplete data in the original four datasets, the number of examples supplied with CWE-information is slightly less: 224 530 examples, of which 218 526 are labeled benign and 6 004 labeled vulnerable. The dataset is split into train-validation-test sets with ratios 80:10:10. This split is chronological to prevent data leakage in the form of time traveling, which occurs when a model inadvertently trains on future data to predict past events.

3.3.1 Data Cleaning

Before the dataset can be used, it is cleaned of duplicates, incomplete functions, and functions with character lengths more than three standard deviations from the mean. Irrelevant CWEs are also removed, leaving only those specified in the list above. The distribution of the remaining CWEs is shown in Figure 3.1.

Many non-vulnerable functions in PrimeVul have been incorrectly assigned CWEs. To ensure internal consistency, all non-vulnerable functions are assigned the CWE "Benign" to better distinguish them from the vulnerable functions.

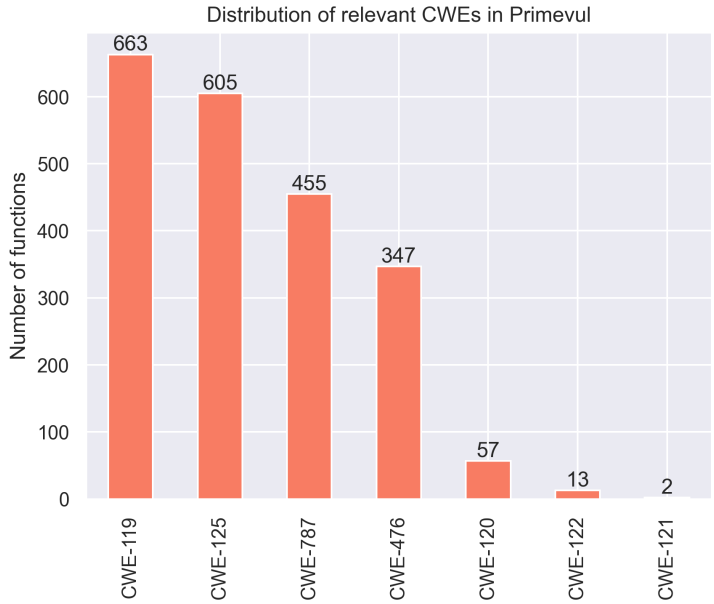
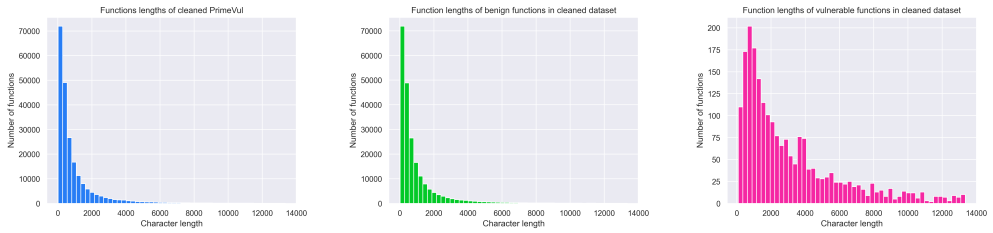


Figure 3.1: Distribution of relevant CWEs in PrimeVul after data cleaning.

The character lengths of the functions in the dataset are of high importance to consider, since the LLMs have a limited context window. As is shown in figure 3.2, if a function is vulnerable, it has a greater probability of being longer, and thus a limited context window disproportionately affects vulnerable functions.

3.3.2 Downsampling

After data cleaning, the PrimeVuls dataset exhibits a 101:1 ratio of non-vulnerable to vulnerable functions. This significant class imbalance risks teaching the model to classify all functions as non-vulnerable. To mitigate this, downsampling is applied. Downsampling reduces the number of examples of the majority class, decreasing the ratio between the majority and minority classes to create a more balanced distribution [29]. Consequently,



(a) All functions. (b) Benign functions. (c) Vulnerable functions.

Figure 3.2: Distributions of function lengths of different subsets of cleaned PrimeVul. Outliers have been removed. Note the difference in scales of the vertical axes.

the model is more frequently exposed to vulnerable examples during the fine-tuning process, which aids it in recognizing vulnerabilities [26].

For this thesis, downsampling is performed by randomly selecting non-vulnerable functions from the training dataset. The ratio between non-vulnerable and vulnerable functions was initially set to 10:1, but is decreased to 3:1 for the final model to optimize performance and reduce fine-tuning time.

Stage	**CWEs**	**Functions**	**Vulnerable**	**Ratio**
Initial	119	175 797	4 862	35:1
After Data Cleaning	7	171 296	1 683	101:1
After Downsampling	7	6 732	1 683	3:1

Table 3.2: PrimeVul training set metrics across different processing stages. Ratio refers to the ratio of the number of benign to the number of vulnerable functions during the different stages.

Table 3.2 shows the amount of functions, vulnerable functions and CWEs in PrimeVul in the different stages described above.

Chapter 4

Methodology

This chapter details the methodology used for fine-tuning local LLMs for C/C++ vulnerability detection. It covers hyperparameter selection, the training loop, and evaluation. Additionally, it outlines the API integration process utilized for the cloud based models. Fine-tuning and evaluation is conducted on a remote desktop with an NVIDIA GeForce RTX 4070 Ti SUPER, which has 16GB of VRAM.

4.1 Fine-Tuning Process

This section describes the fine-tuning process, including code implementation in Python and hyperparameter selection.

4.1.1 Hugging Face

Hugging Face [40] is a prominent open-source platform that develops computation tools for building applications using machine learning. It provides pre-trained models with their weights, configuration and documentation [44]. For this thesis, Hugging Face is selected as the primary resource for the local models used during fine-tuning. This is due to its ease of use and simple implementation with the other programs being used. Hugging Face provides access to the Qwen3, Mistral, Llama 3.2, CodeLlama and DeepSeek Coder.

4.1.2 Unsloth

Unsloth is an open-source Python library designed for fine-tuning LLMs [79]. This library is utilized during the thesis to develop the fine-tuning framework. Unsloth's extensive collection of pre-existing code and published Colab notebooks [80] serves as a technical reference for developing the custom notebooks used in the methodology.

4.1.3 Hyperparameters

Hyperparameters significantly influence model performance during the fine-tuning process, and it is therefore essential to evaluate various combinations of these to determine the optimal settings for a given task. Two common approaches for selecting hyperparameters include trial and error, and adhering to established recommendations from previous research [65]. The hyperparameters which are of interest are batch size, number of epochs, learning rate, seed, LoRA rank, and degree of quantization. The used values of the hyperparameters are inspired by previous research and Unsloth's notebooks with pre-set hyperparameters.

Effective Batch Size

The effective batch size is set to 2, after testing sizes of 1 and 2. A higher batch size can not be employed due to memory constraints. The effective batch size is achieved by using a batch size of 1 together with a gradient accumulation of 2, as their product yields the effective batch size.

Batch size defines the number of examples used in one iteration. A smaller batch size can give the advantage of a more accurate gradient estimation with more frequent updates [1]. A larger batch size however, uses more memory but provides more stable updates [64]. Gradient accumulation on the other hand, simulates a larger effective batch size with a reduced memory usage. It accumulates gradients over several smaller batches before updating the model parameters [32].

Epoch

An epoch represents one complete pass of the entire dataset [54]. The number of epochs for the fine-tuning process is set to 2. This limit prevents the model from overfitting the dataset and ensures the training remains within

the VRAM capacity of the available hardware. Overfitting refers to the model memorizing examples from the training dataset which would result in a lower loss during training, while its performance on other datasets would decline dramatically in comparison [86].

Learning Rate

The learning rate defines the speed with which the model adjusts to the task. A smaller learning rate increases the amount of required training, while a larger rate results in faster adaptation at the cost of less stability [64]. In Unsloth's notebooks, the learning rate was set to either $2 \cdot 10^{-4}$ or $2 \cdot 10^{-5}$, depending on the desired length of the training [80]. As the fine-tuning would only run for 2 epochs, the learning rate is set to $2 \cdot 10^{-4}$.

Seed

A seed is a numerical starting value used to initialize a random number generator. The seed is set to 3407 to provide reproducibility of the results. The number is arbitrary, but is the standard seed in Unsloth's open notebooks and was therefore chosen [80]. Specifying the seed is useful, as it can simplify debugging, as well as make it possible to further development from any previous results [64]. Using the same seed should result in the exact same outcome.

LoRA Rank

The LoRA rank determines the number of trainable parameters. A lower rank utilizes less memory, while a higher rank increases the model's capacity to approximate the original weight updates [39]. In this study, a rank of 32 is utilized, based on both previous research and common practices in open-source fine-tuning.

QLoRa

QLoRA is implemented and set to its 4-bit quantization to reduce the memory usage during fine-tuning. By implementing QLoRA, a larger number of the local models' parameters are able to be stored on the hardware.

Summary

Hyperparameter	Value
Batch size	1
Gradient accumulation	2
Effective batch size	2
Epochs	2
Learning rate	$2 \cdot 10^{-4}$
Seed	3407
LoRA rank	32
QLoRA	4-bit

Table 4.1: Hyperparameters used during fine-tuning of the local models.

Table 4.1 summarizes the final hyperparameters chosen for fine-tuning of the local models.

4.1.4 Fine-Tuning

Data Preparation

Data preparation, which precedes the fine-tuning process, includes cleaning and downsampling the PrimeVul dataset to ensure high data quality prior to training, evaluation and testing. The ratio between non-vulnerable and vulnerable functions is 3:1 after downsampling. These preparations are described in further detail in Section 3.3: PrimeVul.

Model Selection

The models selected for fine-tuning are Qwen3 [66], Llama 3.2 [46], Mistral [52], CodeLlama [41] and DeepSeek Coder [27]. These are chosen due to their accessibility via Hugging Face [40], and are then processed through the

same fine-tuning pipeline to evaluate the performance of code-specific models against general purpose LLMs.

Given limited computational resources, models with smaller parameter counts are prioritized to avoid hardware instability and to minimize training time. Each model is fine-tuned to provide a comparison both between the different local models and against their respective base (non-fine-tuned) version. Additionally, a comparison with the cloud models can also be made.

Fine-Tuning Loop

The fine-tuning pipeline follows an iterative empirical approach. Each cycle begins with fine-tuning a selected local model on the cleaned and down-sampled PrimeVul training set. Following the training loop, the model is evaluated using the validation dataset. Based on these evaluation results, hyperparameters are adjusted and the process is then repeated. This iterative optimization continues until the hyperparameters are set to the values summarized in Table 4.1 to facilitate a direct comparison across all models. Once the hyperparameters are established, the fine-tuned model undergo final testing on the previously unseen test dataset. The data collected from this final stage is presented in Chapter 5: Results. This pipeline is illustrated in Figure 4.1.

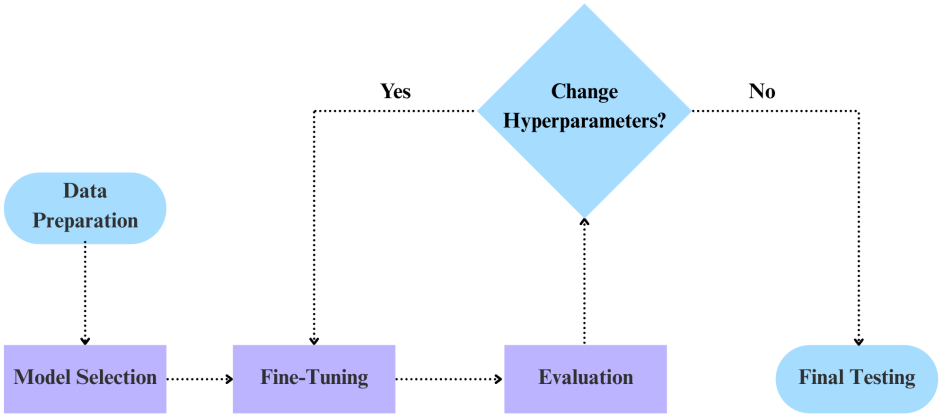


Figure 4.1: Pipeline of the fine-tuning process.

4.2 API Access

This section details the acquisition of API access for the cloud-based LLMs.

The models selected for this thesis are ChatGPT-5.4 [62] and Claude Opus 4.7 [5], accessed by paying their respective developer platforms. Access is provided through a token-based subscription, allowing for the creation of project specific API keys for the duration of the study. No fine-tuning is performed on these models; instead they are integrated into the same evaluation loop used for the local models to ensure a consistent and fair comparison. The evaluation is committed through the API in batch mode, reducing costs with the trade-off of an asynchronous, longer-taking evaluation. Consequently, the evaluation could not record processing time for the individual functions, due to this asynchronicity.

4.3 Evaluation

This section describes the evaluation that is performed for the local fine-tuned models, the local base models, as well as for the cloud based LLMs. The different prompting techniques used during the evaluation are also detailed.

The datasets used during evaluation and final testing are PrimeVul's validation and test sets, respectively. The evaluation and final testing are performed with the same Jupyter Notebook, by exchanging the validation set for the test set when the final testing is to take place. For each evaluation, the accuracy, recall, precision and F_1-score is calculated for each LLM.

4.3.1 Prompting Techniques

During evaluation, two different prompting methods are implemented. These are presented with their respective prompts here. Both prompts begin by instructing the model that it is an advanced vulnerability detection model, followed by the specific task at hand. For the few-shot prompt, a pair of examples is included. At the end of the prompt, the model is provided with a function, and asked whether it is vulnerable or not.

Zero-Shot Prompting

The zero-shot prompt used during evaluation is presented below in Figure 4.2. After the zero-shot prompt, the LLMs are presented with the function in question.

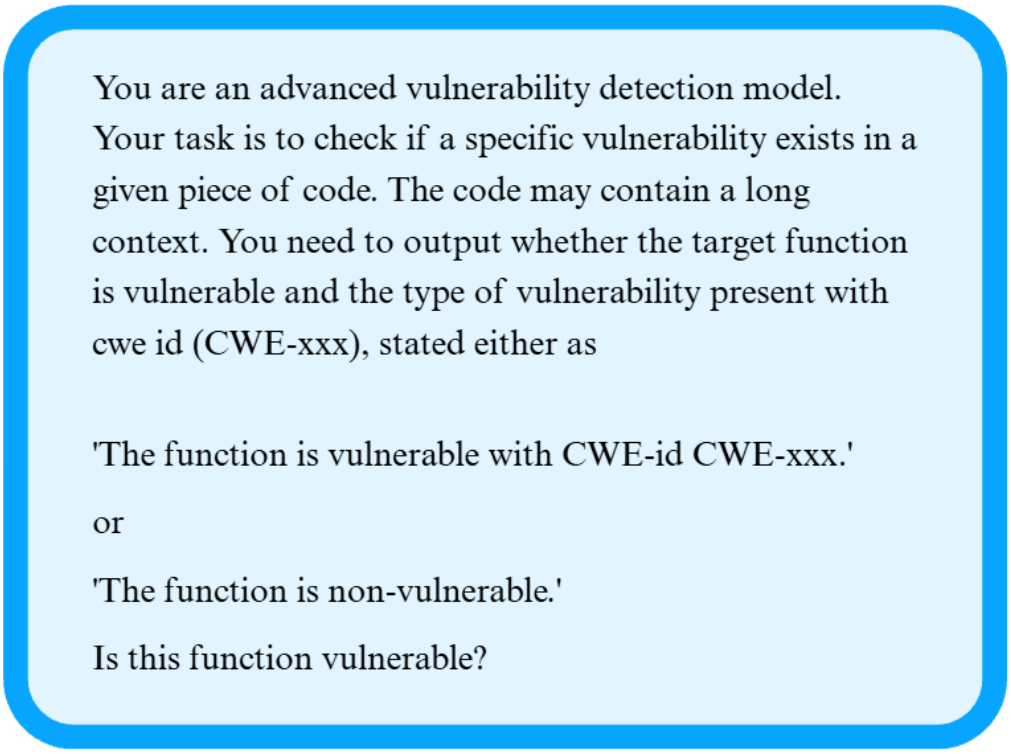


Figure 4.2: Zero-shot prompt which was used during evaluation for the LLMs. After this prompt the LLMs were presented with the function in question.

Few-Shot Prompting

The few-shot prompt used during the evaluation is divided into two parts. First the model is presented with a prompt similar to the zero-shot prompt, without the final question at the end. This is shown in Figure 4.3. Following this, the model is provided with two example functions, one vulnerable and one non-vulnerable. The 2-shot prompt is presented in Figure 4.4.

These two specific examples given to the LLM are taken and removed from the training dataset before being used in the evaluation step. These are

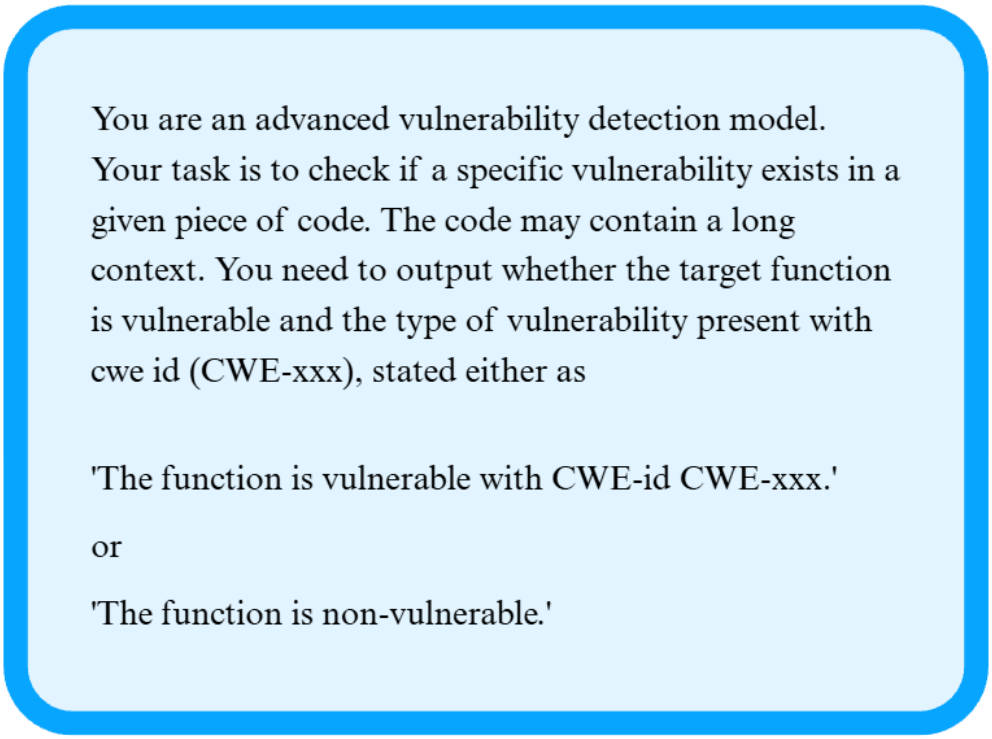


Figure 4.3: The first part of the prompt used in few-shot prompting for evaluation with the instructions on how the model should behave.

chosen at random, with the only restriction being the number of tokens in order to limit input token usage.

4.3.2 Handling of Invalid Outputs

When evaluating the models, invalid outputs that do not follow the formatting specified within the prompts above need to be managed. In order to avoid inflating the scores of models that tend to return invalid responses, the examples are not simply removed from the total when the metrics are calculated. Instead, whenever a model fails in answering with a valid response, this is counted as an incorrect answer. This punishes models which are

Here are two examples of functions with the desired response:

Function 1:
get_next_file(FILE *VFile, char *ptr) { char *ret; ret = fgets(ptr, PATH_MAX, VFile); if (!ret) return NULL; if (ptr[strlen(ptr) - 1] == '\n') ptr[strlen(ptr) - 1] = '\0'; return ret; }
Response 1:
The function is vulnerable with CWE-id CWE-120.

Function 2:
static void invalidate_stack_devices (i_ctx_t *i_ctx_p) { os_ptr op = osbot; while (op != ostop) { if (r_has_type(op, t_device)) op>value.pdevice = 0; op++; }}
Response 2:
The function is non-vulnerable.

Is this function vulnerable?

Figure 4.4: The second part of the prompt used in few-shot prompting for evaluation, including its two examples of functions.

unable to return coherent and concise outputs, and renders the comparison of metrics between models more meaningful. Additionally, the number of invalid responses for each evaluation run is counted, yielding another point of comparison.

Chapter 5

Results

In this chapter, the results for the four different research questions are presented in several tables.

5.1 Research Questions

RQ1 - How do locally fine-tuned LLMs compare to cloud LLMs for vulnerability detection?

This section presents the results for research question 1.

Cloud model	Acc.	Prec.	Rec.	F_1
				Zero-shot
ChatGPT-5.4	0.667	0.037	0.739	0.071
Claude Opus 4.7	0.855	0.039	0.668	0.073
				Few-shot
ChatGPT-5.4	0.771	0.024	0.646	0.046
Claude Opus 4.7	0.855	0.039	0.668	0.073

Table 5.1: The results from the cloud-based models ChatGPT-5.4 and Claude Opus 4.7, evaluated on zero-shot prompting and few-shot prompting with 2-shot examples. The model with the highest F_1-score is highlighted.

Table 5.1 presents the results for the two cloud-based models, ChatGPT-5.4 and Claude Opus 4.7, evaluated with zero-shot prompting and few-shot prompting. The few shot-prompting is a 2-shot prompting. The model with

the highest F_1-score is highlighted, which in this case is Claude Opus 4.7 for zero-shot and few-shot prompting.

Zero-shot	**Acc.**	**Prec.**	**Rec.**	**F_1**
ChatGPT-5.4	0.667	0.037	0.739	0.071
Claude Opus 4.7	0.855	0.039	0.668	0.073
CodeLlama 7B	0.947	0.072	0.431	0.124
DeepSeek Coder 6.7B	0.935	0.052	0.378	0.091
Llama 3.2 3B	0.989	0.000	0.000	0.000
Mistral 7B	0.965	0.060	0.206	0.093
Qwen3-1.7B	0.932	0.056	0.435	0.100

Table 5.2: Comparison between the cloud models and the fine-tuned local models, evaluated using zero-shot prompting. The model with the highest F_1-score is highlighted.

In Table 5.2 a comparison is made between the cloud LLMs and fine-tuned local LLMs when zero-shot prompting is employed. The highlighted row shows the LLM with the highest F_1-score, CodeLlama 7B.

Few-shot	**Acc.**	**Prec.**	**Rec.**	**F_1**
ChatGPT-5.4	0.771	0.024	0.646	0.046
Claude Opus 4.7	0.855	0.039	0.668	0.073
CodeLlama 7B	0.943	0.069	0.450	0.120
DeepSeek Coder 6.7B	0.910	0.040	0.412	0.073
Llama 3.2 3B	0.988	0.023	0.010	0.013
Mistral 7B	0.957	0.050	0.220	0.081
Qwen3-1.7B	0.930	0.055	0.440	0.098

Table 5.3: Comparison between the cloud models and the fine-tuned local models, evaluated using few-shot prompting. The model with the highest F_1-score is highlighted.

From Table 5.3 the cloud models and locally fine-tuned models are compared when few-shot prompting is used. The highlighted row shows which model performs best with respect to F_1-score, which is CodeLlama 7B.

RQ2 - How do the different fine-tuned LLMs compare to each other for vulnerability detection?

The results for research question 2 are included in this section.

Fine-tuned model	Acc.	Prec.	Rec.	F_1
				Zero-shot
CodeLlama 7B	0.947	0.072	0.431	0.124
DeepSeek Coder 6.7B	0.935	0.052	0.378	0.091
Llama 3.2 3B	0.989	0.000	0.000	0.000
Mistral 7B	0.965	0.060	0.206	0.093
Qwen3-1.7B	0.932	0.056	0.435	0.100
				Few-shot
CodeLlama 7B	0.943	0.069	0.450	0.120
DeepSeek Coder 6.7B	0.910	0.040	0.412	0.073
Llama 3.2 3B	0.988	0.023	0.010	0.013
Mistral 7B	0.957	0.050	0.220	0.081
Qwen3-1.7B	0.930	0.055	0.440	0.098

Table 5.4: The results from the fine-tuned local models evaluated on zero-shot prompting and few-shot prompting with 2-shot examples. The model with the highest F_1-score is highlighted.

Table 5.4 presents the results for the five fine-tuned models: CodeLlama 7B, DeepSeek Coder 6.7B, Llama 3.2 3B, Mistral 7B and Qwen3-1.7B. The models are evaluated with zero-shot prompting and few-shot prompting. The model with the highest F_1-score is highlighted for each prompting method, and for both cases this is CodeLlama 7B.

RQ3 - Does fine-tuning increase the performance of local LLMs relative to their original pre-trained version for vulnerability detection?

Tables presented in this section are in respect to research question 3.

In Table 5.5 the results from the evaluation of the base versions of the local models are presented. These are not fine-tuned, but are evaluated with the same zero-shot prompting and few-shot prompting as their fine-tuned versions. The highest F_1-score is highlighted. The base version of Mistral 7B has the highest performance for the zero-shot prompting. CodeLlama

Base model	Acc.	Prec.	Rec.	F_1
				Zero-shot
CodeLlama 7B	0.010	0.008	0.919	0.016
DeepSeek Coder 6.7B	0.000	0.000	0.000	0.000
Llama 3.2 3B	0.507	0.006	0.345	0.012
Mistral 7B	0.124	0.009	0.943	0.018
Qwen3-1.7B	0.212	0.004	0.335	0.007
				Few-shot
CodeLlama 7B	0.050	0.009	0.986	0.018
DeepSeek Coder 6.7B	0.000	0.000	0.000	0.000
Llama 3.2 3B	0.460	0.006	0.392	0.012
Mistral 7B	0.011	0.009	0.928	0.018
Qwen3-1.7B	0.909	0.002	0.019	0.004

Table 5.5: The results from the local base models without fine-tuning evaluated on zero-shot prompting and few-shot prompting with 2-shot examples. The highest performing model is marked.

7B and Mistral 7B have the highest performance for base versions evaluated with few-shot prompting.

In Figure 5.1 the number of invalid responses of the base models can be seen. An output is regarded as invalid if it is not formatted according to the instruction in the prompts. In the case of fine-tuned models, only one invalid output is ever produced, by Mistral 7B during the few-shot run of the evaluation.

RQ4 - How do the different prompting methods compare?

The results in reference to research question 4 are presented in this section.

Table 5.6 presents the results considered relevant to research question 4. It compares the results of the zero-shot and few-shot evaluations, and includes the five different local models, both their original version and their fine-tuned, as well as the two cloud models. The F_1-scores of the local models which are not fine-tuned are written in parentheses. All LLMs are evaluated with zero-shot prompting and few-shot prompting. The highest performance LLM in reference to F_1-score, is highlighted for each model. For Claude Opus 4.7 both models achieve equal results, therefore none of the prompting methods are highlighted.

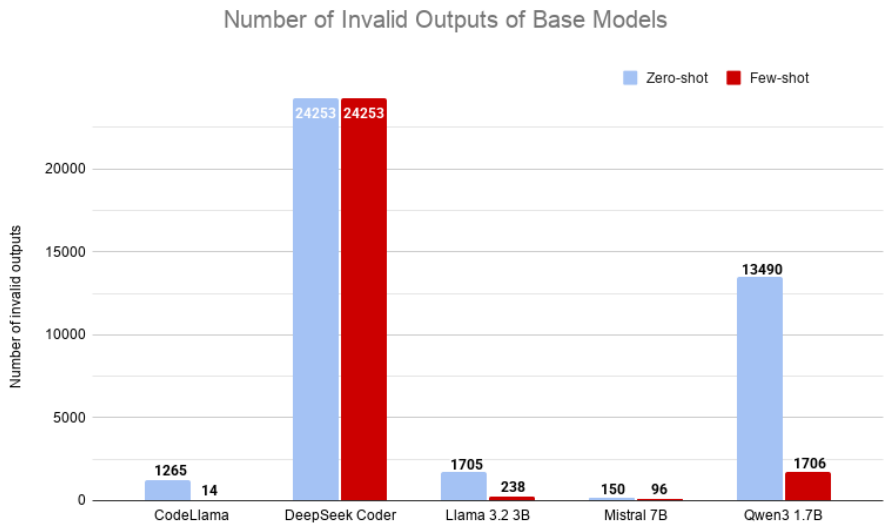
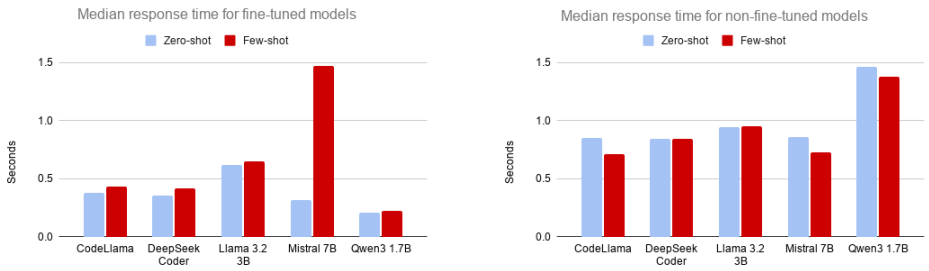


Figure 5.1: The number of invalid outputs generated during testing for the non-fine-tuned base models.

Figure 5.2 shows the median response times for both the fine-tuned LLMs and their base models.



(a) Fine-tuned models.

(b) Base models.

Figure 5.2: Median response times for the fine-tuned and base models, broken up by zero- and few-shot.

Prompting	Acc.	Prec.	Rec.	F_1
				ChatGPT-5.4
Zero-shot	0.667	0.037	0.739	0.071
Few-shot	0.771	0.024	0.646	0.046
				Claude Opus 4.7
Zero-shot	0.855	0.039	0.668	0.073
Few-shot	0.855	0.039	0.668	0.073
				CodeLlama 7B
Zero-shot	0.947 (0.010)	0.072 (0.008)	0.431 (0.919)	0.124 (0.016)
Few-shot	0.943 (0.050)	0.069 (0.009)	0.450 (0.986)	0.120 (0.018)
				DeepSeek Coder 6.7B
Zero-shot	0.935 (0.000)	0.052 (0.000)	0.378 (0.000)	0.091 (0.000)
Few-shot	0.910 (0.000)	0.040 (0.000)	0.412 (0.000)	0.073 (0.000)
				Llama 3.2 3B
Zero-shot	0.989 (0.507)	0.000 (0.006)	0.000 (0.345)	0.000 (0.012)
Few-shot	0.988 (0.460)	0.023 (0.006)	0.010 (0.392)	0.013 (0.012)
				Mistral 7B
Zero-shot	0.965 (0.124)	0.060 (0.009)	0.206 (0.943)	0.093 (0.018)
Few-shot	0.957 (0.011)	0.050 (0.009)	0.220 (0.928)	0.081 (0.018)
				Qwen3-1.7B
Zero-shot	0.932 (0.212)	0.056 (0.004)	0.435 (0.335)	0.100 (0.007)
Few-shot	0.930 (0.909)	0.055 (0.002)	0.440 (0.019)	0.098 (0.004)

Table 5.6: Comparison between the locally fine-tuned models, their base versions in parenthesis, and the cloud models. The model which achieves the highest F_1-score is highlighted.

Chapter 6

Discussion

This chapter includes the discussion regarding the results for each research question. Additionally, limitations, ethical considerations and environmental considerations will be presented.

6.1 Research Questions

RQ1 - How do locally fine-tuned LLMs compare to cloud LLMs for vulnerability detection?

The results can be seen in Table 5.1, 5.2 and 5.3.

The cloud-based models did not achieve the expected performance levels. Higher results were anticipated due to these models being larger and pre-trained on much more extensive datasets. As shown in Table 5.1, the F_1-score reached only 5% for ChatGPT-5.4 and 7% for Claude Opus 4.7 for the few-shot prompting. Slightly better results were achieved for ChatGPT-5.4 when evaluated with zero-shot, as it reached an F_1-score of 7%. The lower few-shot performance suggests that instead of helping, the examples may have misled the model into treating specific characteristics of those samples as universal rules for the entire dataset. Interestingly, Claude Opus 4.7 produced identical results irrespective of prompting technique. While Claude Opus 4.7 produced better results than ChatGPT-5.4 under few-shot prompting, the overall scores reflect the restrictive nature of the evaluation, which limited the models reasoning capabilities by enforcing a strict output limit. Despite the extensive pre-training of cloud models, which led to higher performance expectations compared to locally fine-tuned versions, CodeLlama 7B achieved

the highest performance in both the zero-shot and few-shot comparisons (see Figures 5.2 and 5.3).

The cloud models are trained to reason and think before giving an output. When limiting the output tokens to 30, the models are not able to utilize their full reasoning capabilities. A locally fine-tuned model is prepared for the question and output since it has been specifically trained on similar cases. Therefore, a locally fine-tuned model is more equipped to answer the question with the desired output than a cloud model, given the restrictions mentioned above.

Another possible reason why CodeLlama 7B outperformed the cloud models is that the local models were trained on a dataset which is extremely similar to the one used during evaluation. This could benefit the locally fine-tuned models in unexpected ways, by priming them to the formatting and conventions within the dataset which might not exist in code that the cloud models are trained to understand.

The given prompts may also have been too brief for the cloud models, since they had not been prepared beforehand on the subject in contrast to the local models.

RQ2 - How do the different fine-tuned LLMs compare to each other for vulnerability detection?

As shown in Table 5.4, the best performing model for vulnerability detection among the locally fine-tuned LLMs was CodeLlama 7B, across both zero-shot and few-shot prompting. Its F_1-score reached 12%, only slightly higher than the second best model, Qwen3-1.7B, which achieved 10%. Overall, the LLMs performed similarly regardless of the prompting method, with only marginal differences observed between zero-shot and few-shot results. For CodeLlama 7B, few-shot prompting performed slightly better, as expected, since the inclusion of two example functions provided clearer instructions on the required output format. However, contrary to expectations, some models performed better with zero-shot prompting despite the prompts being identical in all other aspects. This could be because fine-tuned models do not need the extra examples in few-shot prompting, since they have already been trained on similar cases.

None of the locally fine-tuned models performed at a high level. While catching 10% to 12% of vulnerabilities offers some utility over zero detection, this performance may be insufficient for real-world use.

Firstly, the models were restricted to a maximum of 30 output tokens, which forced them to state only whether a function was vulnerable or not without any elaborate reasoning. This was done to limit the time needed for evaluation and to remove the need of too many output tokens. If the LLMs would have access to reasoning, over 1 200 output tokens would have been needed per example.

Secondly, the models may have developed a bias during fine-tuning, as the training dataset primarily consisted of non-vulnerable functions, potentially leading the models to favor non-vulnerable classifications.

It was expected that CodeLlama 7B and DeepSeek Coder 6.7B would be in the top of performances, as both are specifically pre-trained on code, whereas the other three models are general-purpose LLMs. While it was no surprise that CodeLlama 7B performed best, it was unexpected that Qwen3-1.7B achieved the second highest F_1-score, despite having a significantly smaller parameter count than its competitors.

RQ3 - Does fine-tuning increase performance of local LLMs relative to their original pre-trained version for vulnerability detection?

Both Table 5.5 and Table 5.6 provide insight into the performance of the non-fine-tuned base models. Table 5.5 illustrates that the local base LLMs performed similarly across both evaluation cases, with negligible differences between zero-shot and few-shot prompting. CodeLlama 7B remains among the top performing base models, alongside Mistral 7B and Qwen3-1.7B.

Table 5.6 compared the locally fine-tuned models against their base versions. The highest performing local model was fine-tuned CodeLlama 7B, followed by the fine-tuned version of Qwen3-1.7B. These results suggest that supervised fine-tuning (SFT) generally improves performance over the base versions, with notable exception of Llama 3.2 3B. This aligns with the initial hypothesis, as fine-tuning is intended to specialize an LLM for specific task. Had the fine-tuned models underperformed compared to the base versions, it would have indicated a possible flaw in the fine-tuning pipeline. While a more substantial improvement than the increase from 2% to 12% was anticipated, the results still highlight the importance of fine-tuning for this application.

In Figure 5.2, the effects of SFT on the response times of the models can be seen. In every case, except for the few-shot Mistral 7B evaluation, the median time decreased when fine-tuning was implemented. This is likely due to the fine-tuning priming the models for the examples in the dataset,

enabling quicker response times. This may also be owing to heavily reducing the valid answer set for the models, essentially reducing it to only two options: vulnerable or benign.

The reason for Mistral 7B breaking this pattern during the few-shot run of the evaluation is unclear, but may be due to a strain on the machine performing the evaluation due to excessive use over a prolonged period of time.

Additionally, fine-tuning appears to be a crucial step in ensuring that the models format their responses correctly, as can be seen in Figure 5.1. DeepSeek Coder is the extreme example of this, as the base version of the model returned ':' or ')' in every test case of the dataset, instead of the result of a security analysis. This, in comparison to there being a single invalid answer from all fine-tuned models, indicates fine-tuning to be a critical step in vulnerability detection using local LLMs.

RQ4 - How do the different prompting methods compare?

In Table 5.6, it can be seen that for the local models, the different prompting methods had little to no impact on any of the F_1-scores, with the largest difference being 0.018 between zero-shot and few-shot for DeepSeek Coder 6.7B.

As for the base version of the Qwen3-1.7B model, there is an accuracy increase from 0.21 to 0.91 between the zero- and few-shot runs. This has multiple possible causes. One is that the few-shot examples greatly assisted Qwen3 in understanding how to correctly format its response, reducing the number of invalid outputs from the model, as can be seen in Figure 5.1.

Another possible reason for this sharp increase is the imbalanced nature of the dataset, and the fact that the model evaluated using zero-shot prompting classified most of its functions as vulnerable, while the few-shot prompting resulted in the model classifying most as non-vulnerable. As the dataset is heavily imbalanced with a strong majority of non-vulnerable examples, this difference in strategy increases the accuracy score significantly, while not actually being indicative of an increase in underlying understanding or performance.

A third reason might be that few-shot prompting limited the amount of rambling answers, reducing the risk of misclassifications.

The code for processing the raw output is this:

```
        if "non-vulnerable" in final_verdict:
            is_pred_vulnerable = 0
        elif "is vulnerable" in final_verdict:
            is_pred_vulnerable = 1
        else:
            is_pred_vulnerable = -1

```

With this logic, if the output mentions the word 'non-vulnerable' it will classify the function as such, even though it might mention the word in a reasoning context before the model was cut-off.

Regarding the time required, for the fine-tuned models, few-shot prompting increased the amount of time taken for all local models, as can be seen in Figure 5.2. This seems to be due to the greater processing time associated with a longer prompt. The time differs with at most 0.1 seconds, except for the case of Mistral 7B, for which it differs more than 1 second, an almost 5 times increased processing time for each function. It is unclear what caused this discrepancy for Mistral.

When studying the response times for the base versions of the models in Figure 5.2, this connection is reversed; in all cases, the processing time was reduced, if not essentially unchanged, when using few-shot prompting instead of zero-shot prompting. This is likely due to the examples in the few-shot prompt aiding the untrained models in deciding how to format their outputs, reducing the time sufficiently to negate the increased processing time from the longer prompt.

To summarize, few-shot greatly aided the non-fine-tuned versions of the local models in formatting their outputs correctly, which Figure 5.1 shows, as compared to zero-shot. However, save for the case of the base version of Qwen3, the difference between zero-shot and few-shot was unremarkable with respect to the relevant metrics, indicating that the choice between zero-shot prompting and few-shot prompting has a limited impact on the performance of LLMs as tools for vulnerability detection. Furthermore, the choice of prompting method appears to affect the processing time only a small amount. In the case of the fine-tuned models, few-shot prompting increased the required time, while decreasing for the base versions of the LLMs.

This indicates that the use of few-shot prompting and the use Supervised Fine-Tuning both affect the local models and their results in similar ways, to varying levels of success. Both reduce the amount of processing time

required, and both improve the formatting capabilities of the models. As was discussed with regards to Figure 5.2, these two techniques might in fact interfere with each other to some degree when used together, and since SFT is more effective with respect to increasing performance and enforcing correct formatting, it emerges as the superior choice.

6.2 Limitations and Threats to Validity

Several limitations were encountered during this research. Primarily, the NVIDIA GeForce RTX 4070 Ti SUPER used in this study is limited to 16 GB of VRAM, which limited the use of larger parameter models. Local models were selected by testing higher parameter sizes and descending until hardware stability was achieved. Consequently, it remains a point of interest how a model like Qwen3-32B would have performed, given that the 1.7B variant was among the top performers for the locally fine-tuned models.

Secondly, there was also a time limitation when fine-tuning the LLMs. Data preparation took longer time than expected, which lead to the implementation of fine-tuning being delayed. Due to this, the fine-tuning was only iterated twice before results had to be collected. Therefore, the hyperparameters could not be optimized further for better results.

Lastly, capping the number of output tokens of the models to 30, which was done to reduce the evaluation time and cost, severely limited the models in their reasoning capabilities. This meant that the models relied solely on their pattern recognition, without being able to arrive at a conclusion via logical steps.

6.3 Ethical Considerations

Ethical considerations are fundamental to the field of vulnerability detection. When a vulnerability is identified, patching it is critical. However, a dilemma arises regarding disclosure. Should the knowledge be shared publicly or kept secret? While public disclosure pressures vendors to provide a fix, it simultaneously provides malicious actors with a guide to exploit the vulnerability before a patch is deployed. The situation is further complicated by considering what is to be done in the case of older system that do not or can not receive updates. A common resolution is responsible disclosure,

where vendors are given a private window to resolve the issue before the information is made public. A further complication exists if the vendor fails to respond or patch it.

Additionally, ethical concerns involve the legality of stumbling upon a vulnerability. Without prior authorization, such discoveries can be legally interpreted as attempted hacking, regardless of the researcher's initial intent.

Finally, when employing LLMs for detection, a false sense of security may emerge if the model is blindly trusted. Continuous verifications remain essential to ensure the system is functioning correctly and has not been manipulated by malicious intent.

6.4 Environmental Considerations

A critical concern is the environmental impact of AI, ranging from high energy consumption to the significant water usage required for data center cooling. While the utility of LLMs is powerful, ethical discussions must address ownership, usage policies and accountability for the ecological footprint created by these technologies.

In the hype of AI usage, the potential for harm must not be overlooked. It is essential to evaluate whether AI provides a net benefit in every project and application. It could be argued that AI for vulnerability detection has its benefits since it will keep private information secure, but it still needs to be taken into account how it affects the world. Being informed within the field can make a difference in the footprint.

Chapter 7

Conclusion

This chapter details the conclusion along with suggestions for future research.

This thesis has evaluated five different local LLMs and two cloud-based LLMs, comparing locally fine-tuned models against their respective base versions and cloud-based models. The results indicate that while fine-tuning enhances local model performance, the improvement was less substantial than aimed for. Furthermore, a clear bias towards labeling functions as non-vulnerable was observed in the fine-tuned models.

Regarding prompting techniques, there is no definitive advantage for either zero-shot or few-shot approaches with respect to the performance metrics. The differences in F_1-score between the two was insignificant, with each method slightly outperforming the other depending on the specific model.

Cloud-based models exhibited lower performance than the locally fine-tuned models. Due to reasoning being disabled during evaluation, all models relied primarily on pattern recognition. The resulting bias toward non-vulnerable labels in fine-tuned models likely stems from the training dataset, which contained a significant higher proportion of non-vulnerable functions. Currently, these local LLMs are not suitable for real-world applications, a significantly higher F_1-score and a reduction in majority-class bias would be required before implementing a locally fine-tuned model for vulnerability detection in C/C++ code.

7.1 Future Research

During this research, several questions emerged that could not be fully addressed due to time and hardware constraints. These are presented in this section as opportunities for future research.

Chain of Thought Prompting

Future research should focus on evaluating these models using Chain-of-Thought (CoT) prompting [83], as this technique is documented to enhance model performance by requiring the generation of intermediate reasoning steps before a final verdict. Furthermore, the implementation of CoT requires significant computational resources due to high output token counts, which substantially increases the required time frame. Consequently, a comparative analysis of CoT across locally fine-tuned models remains a compelling area for further investigation to achieve better performance.

Retrieval Augmented Generation

Another area of interest is the implementation of Retrieval Augmented Generation (RAG), which provides LLMs access to an external knowledge base to retrieve relevant information during usage [36]. This technique could serve as an alternative or a complement to fine-tuning, allowing for a comparison of different methodologies to optimize performance. Specifically, future work could investigate whether RAG is a more suitable process than fine-tuning for identifying specific C/C++ vulnerabilities.

Supervised Fine-Tuning for Cloud LLMs

Lastly, the implementation of supervised fine-tuning (SFT) [77] for cloud-based models ought to be investigated. Such research would determine whether fine-tuning cloud models yields significant performance gains over their standard pre-trained versions. Furthermore, it would establish whether the performance changes observed in local fine-tuning are consistent with those achieved in cloud-based environments.

Bibliography

[1] Justus A Ilemobayo et al. 'Hyperparameter Tuning in Machine Learning: A Comprehensive Review'. en. In: *Journal of Engineering Research and Reports* 26.6 (June 2024), pp. 388–395. ISSN: 2582-2926. DOI: 10.9734/jerr/2024/v26i61188. URL: https://journaljerr.com/index.php/JERR/article/view/1188 (visited on 31/03/2026).

[2] *Advenica*. Jan. 2026. URL: https://advenica.com/about-us/ (visited on 03/02/2026).

[3] *AI Model & API Providers Analysis | Artificial Analysis*. en. URL: https://artificialanalysis.ai (visited on 24/02/2026).

[4] *Anthropic*. en. URL: https://www.anthropic.com (visited on 24/02/2026).

[5] Anthropic. *Dashboard | Claude Platform*. URL: https://platform.claude.com/dashboard (visited on 15/04/2026).

[6] Anthropic. *Introducing Claude Opus 4.7*. en. Apr. 2026. URL: https://www.anthropic.com/news/claude-opus-4-7 (visited on 20/04/2026).

[7] *API or In-house LLM?* en-US. Dec. 2023. URL: https://aimresearch.co/product/api-or-in-house-llm (visited on 09/02/2026).

[8] Antonio Avolio. *LLM-assisted generation of intentionally vulnerable containerized environments for cyber ranges*. Apr. 2025.

[9] Prashant Bansal. 'Prompt Engineering Importance and Applicability with Generative AI'. en. In: *Journal of Computer and Communications* 12.10 (2024), pp. 14–23. ISSN: 2327-5219, 2327-5227. DOI: 10.4236/jcc.2024.1210002. URL: https://www.scirp.org/journal/paperinformation?paperid=136500 (visited on 30/03/2026).

[10] Guru Bhandari, Amara Naseer and Leon Moonen. 'CVEfixes: automated collection of vulnerabilities and their fixes from open-source software'. In: *Proceedings of the 17th International Conference on Predictive Models and Data Analytics in Software Engineering*. PROMISE 2021. New York, NY, USA: Association for Computing Machinery, Aug. 2021, pp. 30–39. ISBN: 978-1-4503-8680-7. DOI: 10.1145/3475960.3475985. URL: https://dl.acm.org/doi/10.1145/3475960.3475985.

[11] Prabin Bhandari. *A Survey on Prompting Techniques in LLMs*. en. arXiv:2312.03740 [cs]. Apr. 2024. DOI: 10.48550/arXiv.2312.03740. URL: http://arxiv.org/abs/2312.03740.

[12] Dipkamal Bhusal et al. *SECURE: Benchmarking Large Language Models for Cybersecurity*. en. arXiv:2405.20441 [cs]. Oct. 2024. DOI: 10.48550/arXiv.2405.20441. URL: http://arxiv.org/abs/2405.20441.

[13] Sitanath Biswas. *Vulnerability Fix Dataset*. en. Jan. 2025. URL: https://www.kaggle.com/datasets/jiscecseaiml/vulnerability-fix-dataset (visited on 12/02/2026).

[14] Tom B. Brown et al. *Language Models are Few-Shot Learners*. en. arXiv:2005.14165 [cs]. July 2020. DOI: 10.48550/arXiv.2005.14165. URL: http://arxiv.org/abs/2005.14165.

[15] Nicholas Carlini et al. *Evaluating and mitigating the growing risk of LLM-discovered 0-days*. Feb. 2026. URL: https://red.anthropic.com/2026/zero-days/.

[16] Marco Carvalho et al. 'Heartbleed 101'. In: *IEEE Security & Privacy* 12.4 (July 2014), pp. 63–67. ISSN: 1558-4046. DOI: 10.1109/MSP.2014.66. URL: https://ieeexplore.ieee.org/document/6876249/.

[17] Yizheng Chen et al. 'DiverseVul: A New Vulnerable Source Code Dataset for Deep Learning Based Vulnerability Detection'. en. In: *Proceedings of the 26th International Symposium on Research in Attacks, Intrusions and Defenses*. Hong Kong China: ACM, Oct. 2023, pp. 654–668. ISBN: 979-8-4007-0765-0. DOI: 10.1145/3607199.3607242. URL: https://dl.acm.org/doi/10.1145/3607199.3607242.

[18] K. R. Chowdhary. 'Natural Language Processing'. en. In: *Fundamentals of Artificial Intelligence*. Ed. by K.R. Chowdhary. New Delhi: Springer India, 2020, pp. 603–649. ISBN: 978-81-322-3972-7. DOI: 10.1007/978-81-322-3972-7_19. URL: https://doi.org/10.1007/978-81-322-3972-7_19.

[19] Cristina Cifuentes et al. 'The role of program analysis in security vulnerability detection: Then and now'. en. In: *Computers & Security* 135 (Dec. 2023), p. 103463. ISSN: 01674048. DOI: 10.1016/j.cose.2023.103463. URL: https://linkinghub.elsevier.com/retrieve/pii/S0167404823003735.

[20] Roland Croft, M. Ali Babar and Mehdi Kholoosi. *Data Quality for Software Vulnerability Datasets*. en. arXiv:2301.05456 [cs]. Jan. 2023. DOI: 10.48550/arXiv.2301.05456. URL: http://arxiv.org/abs/2301.05456.

[21] *CWE - 2025 CWE Top 25 Most Dangerous Software Weaknesses*. URL: https://cwe.mitre.org/top25/archive/2025/2025_cwe_top25.html (visited on 23/02/2026).

[22] *CWE - About CWE*. URL: https://cwe.mitre.org/about/index.html (visited on 23/02/2026).

[23] *CWE - Common Weakness Enumeration*. URL: https://cwe.mitre.org/ (visited on 12/02/2026).

[24] *CWE - CWE-119: Improper Restriction of Operations within the Bounds of a Memory Buffer (4.19.1)*. URL: https://cwe.mitre.org/data/definitions/119.html (visited on 23/02/2026).

[25] *CWE - CWE-476: NULL Pointer Dereference (4.19.1)*. URL: https://cwe.mitre.org/data/definitions/476.html (visited on 23/02/2026).

[26] *Datasets: Class-imbalanced datasets | Machine Learning | Google for Developers*. en. URL: https://developers.google.com/machine-learning/crash-course/overfitting/imbalanced-datasets (visited on 09/04/2026).

[27] *DeepSeek Coder*. URL: https://deepseekcoder.github.io/ (visited on 09/04/2026).

[28] Tim Dettmers et al. *QLoRA: Efficient Finetuning of Quantized LLMs*. arXiv:2305.14314 [cs]. May 2023. DOI: 10.48550/arXiv.2305.14314. URL: http://arxiv.org/abs/2305.14314 (visited on 30/03/2026).

[29] Jesús Díaz-García et al. 'DOWNSAMPLING METHODS FOR MEDICAL DATASETS'. en. In: *Data Mining and Computational Intelligence* (2017).

[30] Cem Dilmegani. *Cloud LLM vs Local LLMs: 3 Real-Life examples & benefits*. en. May 2025. URL: https://research.aimultiple.com/cloud-llm/ (visited on 09/02/2026).

[31] Yangruibo Ding et al. *Vulnerability Detection with Code Language Models: How Far Are We?* en. arXiv:2403.18624 [cs]. July 2024. DOI: 10.48550/arXiv.2403.18624. URL: http://arxiv.org/abs/2403.18624.

[32] Shaza Elmorshidy. *Gradient Accumulation vs. Batch Size in Deep Learning*. en. July 2024. URL: https://medium.com/@shazaelmorsh/gradient-accumulation-vs-batch-size-in-deep-learning-92201055cc7a (visited on 31/03/2026).

[33] Hugging Face. *LoRA · Hugging Face*. URL: https://huggingface.co/docs/peft/package_reference/lora (visited on 16/04/2026).

[34] Jiahao Fan et al. 'A C/C++ Code Vulnerability Dataset with Code Changes and CVE Summaries'. In: *Proceedings of the 17th International Conference on Mining Software Repositories*. MSR '20. New York, NY, USA: Association for Computing Machinery, Sept. 2020, pp. 508–512. ISBN: 978-1-4503-7517-7. DOI: 10.1145/3379597.3387501. URL: https://dl.acm.org/doi/10.1145/3379597.3387501.

[35] Tom Fawcett. 'An introduction to ROC analysis'. en. In: *Pattern Recognition Letters* 27.8 (June 2006), pp. 861–874. ISSN: 01678655. DOI: 10.1016/j.patrec.2005.10.010. URL: https://linkinghub.elsevier.com/retrieve/pii/S016786550500303X.

[36] Yunfan Gao et al. *Retrieval-Augmented Generation for Large Language Models: A Survey*. en. arXiv:2312.10997 [cs]. Mar. 2024. DOI: 10.48550/arXiv.2312.10997. URL: http://arxiv.org/abs/2312.10997.

[37] Jingxuan He and Martin Vechev. 'Large Language Models for Code: Security Hardening and Adversarial Testing'. In: *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*. CCS '23. New York, NY, USA: Association for Computing Machinery, Nov. 2023, pp. 1865–1879. ISBN: 979-8-4007-0050-7. DOI:

10.1145/3576915.3623175. URL: https://dl.acm.org/doi/10.1145/3576915.3623175.

[38] *Heartbleed Bug*. Mar. 2025. URL: https://heartbleed.com/ (visited on 02/02/2026).

[39] Edward J. Hu et al. *LoRA: Low-Rank Adaptation of Large Language Models*. arXiv:2106.09685 [cs]. Oct. 2021. DOI: 10.48550/arXiv.2106.09685. URL: http://arxiv.org/abs/2106.09685 (visited on 30/03/2026).

[40] *Hugging Face – The AI community building the future.* Mar. 2026. URL: https://huggingface.co/ (visited on 25/03/2026).

[41] *Introducing Code Llama, a state-of-the-art large language model for coding*. en. Jan. 2024. URL: https://ai.meta.com/blog/code-llama-large-language-model-coding/ (visited on 09/04/2026).

[42] *Introducing GPT-5.2*. en-US. Feb. 2026. URL: https://openai.com/index/introducing-gpt-5-2/ (visited on 24/02/2026).

[43] Casper Jensen et al. *Inventering av testfall för verktyg som identifierar mjukvarusårbarheter*. Dec. 2024.

[44] Jason Jones et al. 'What do we know about Hugging Face? A systematic literature review and quantitative validation of qualitative claims'. In: *Proceedings of the 18th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*. ESEM '24. New York, NY, USA: Association for Computing Machinery, 2024, pp. 13–24. ISBN: 979-8-4007-1047-6. DOI: 10.1145/3674805.3686665. URL: https://dl.acm.org/doi/10.1145/3674805.3686665 (visited on 25/03/2026).

[45] Xabier Lareo. *Large language models (LLM)*. en. Feb. 2026. URL: https://www.edps.europa.eu/data-protection/technology-monitoring/techsonar/large-language-models-llm (visited on 03/02/2026).

[46] *Llama 3.2: Revolutionizing edge AI and vision with open, customizable models*. en. Sept. 2024. URL: https://ai.meta.com/blog/llama-3-2-connect-2024-vision-edge-mobile-devices/ (visited on 17/02/2026).

[47] *LLM Leaderboard 2025*. en. URL: https://www.vellum.ai/llm-leaderboard (visited on 24/02/2026).

[48] *LTH*. sv. URL: https://www.lth.se/ (visited on 12/02/2026).

[49] Alexandre Magueresse, Vincent Carles and Evan Heetderks. *Low-resource Languages: A Review of Past Work and Future Challenges*. arXiv:2006.07264 [cs]. June 2020. DOI: 10.48550/arXiv.2006.07264. URL: http://arxiv.org/abs/2006.07264 (visited on 23/02/2026).

[50] Kadin Matotek et al. *Evaluating the Limitations of Local LLMs in Solving Complex Programming Challenges*. arXiv:2509.15283 [cs]. Sept. 2025. DOI: 10.48550/arXiv.2509.15283. URL: http://arxiv.org/abs/2509.15283 (visited on 24/02/2026).

[51] Shervin Minaee et al. *Large Language Models: A Survey*. en. arXiv:2402.06196 [cs]. Mar. 2025. DOI: 10.48550/arXiv.2402.06196. URL: http://arxiv.org/abs/2402.06196 (visited on 03/02/2026).

[52] *Mistral 7B | Mistral AI*. en. Sept. 2023. URL: https://mistral.ai/news/announcing-mistral-7b (visited on 17/02/2026).

[53] Thomas F. Monaghan et al. 'Foundational Statistical Principles in Medical Research: Sensitivity, Specificity, Positive Predictive Value, and Negative Predictive Value'. en. In: *Medicina* 57.5 (May 2021), p. 503. ISSN: 1648-9144. DOI: 10.3390/medicina57050503. URL: https://www.mdpi.com/1648-9144/57/5/503 (visited on 24/02/2026).

[54] Kevin P. Murphy. *Machine learning: a probabilistic perspective*. en. Adaptive computation and machine learning series. Cambridge, MA: MIT Press, 2012. ISBN: 978-0-262-01802-9.

[55] Zabir Al Nazi, Md. Rajib Hossain and Faisal Al Mamun. 'Evaluation of open and closed-source LLMs for low-resource language with zero-shot, few-shot, and chain-of-thought prompting'. en. In: *Natural Language Processing Journal* 10 (Mar. 2025), p. 100124. ISSN: 29497191. DOI: 10.1016/j.nlp.2024.100124. URL: https://linkinghub.elsevier.com/retrieve/pii/S2949719124000724.

[56] Georgios Nikitopoulos et al. 'CrossVul: a cross-language vulnerability dataset with commit data'. In: *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ESEC/FSE 2021. New York, NY, USA: Association for Computing Machinery, Aug. 2021, pp. 1565–1569. ISBN: 978-1-4503-8562-6. DOI: 10.1145/3468264.

3473122. URL: https://dl.acm.org/doi/10.1145/3468264.3473122.

[57] *NVD*. URL: https://nvd.nist.gov/ (visited on 12/02/2026).

[58] U. S. Government Accountability Office. *CrowdStrike Chaos Highlights Key Cyber Vulnerabilities with Software Updates | U.S. GAO*. en. July 2024. URL: https://www.gao.gov/blog/crowdstrike-chaos-highlights-key-cyber-vulnerabilities-software-updates (visited on 12/02/2026).

[59] U. S. Government Accountability Office. *Cyber Resiliency: CrowdStrike Outage Highlights Challenges | U.S. GAO*. en. Sept. 2024. URL: https://www.gao.gov/products/gao-24-107733 (visited on 16/02/2026).

[60] *Ollama*. URL: https://ollama.com (visited on 17/02/2026).

[61] *OpenAI*. en-US. Feb. 2026. URL: https://openai.com/ (visited on 24/02/2026).

[62] OpenAI. *Platform Home Page - OpenAI API*. sv-SE. URL: https://platform.openai.com (visited on 15/04/2026).

[63] OpenAI et al. *GPT-4 Technical Report*. arXiv:2303.08774 [cs]. Mar. 2024. DOI: 10.48550/arXiv.2303.08774. URL: http://arxiv.org/abs/2303.08774 (visited on 12/02/2026).

[64] Venkatesh Balavadhani Parthasarathy et al. *The Ultimate Guide to Fine-Tuning LLMs from Basics to Breakthroughs: An Exhaustive Review of Technologies, Research, Best Practices, Applied Research Challenges and Opportunities*. en. arXiv:2408.13296 [cs]. Oct. 2024. DOI: 10.48550/arXiv.2408.13296. URL: http://arxiv.org/abs/2408.13296 (visited on 03/02/2026).

[65] Philipp Probst, Anne-Laure Boulesteix and Bernd Bischl. 'Tunability: Importance of Hyperparameters of Machine Learning Algorithms'. en. In: (Mar. 2019).

[66] *Qwen*. URL: https://qwen.ai/blog?id=qwen3 (visited on 17/02/2026).

[67] *Red Hat Bugzilla Main Page*. URL: https://bugzilla.redhat.com/ (visited on 12/02/2026).

[68] *red.anthropic.com*. URL: https://red.anthropic.com/ (visited on 24/02/2026).

[69] C.J. van Rijsbergen. *Information Retrieval*. 2nd ed. 1979. URL: https://www.dcs.gla.ac.uk/Keith/Preface.html (visited on 24/02/2026).

[70] S. Sathyanarayanan. 'Confusion Matrix-Based Performance Evaluation Metrics'. en. In: *African Journal of Biomedical Research* (Nov. 2024), pp. 4023–4031. DOI: 10.53555/AJBR.v27i4S.4345. URL: https://africanjournalofbiomedicalresearch.com/index.php/AJBR/article/view/4345/3327 (visited on 04/02/2026).

[71] *SEAL LLM Leaderboards: Expert-Driven Evaluations | Scale | SEAL by Scale AI*. en. URL: https://scale.com/leaderboard (visited on 24/02/2026).

[72] C E Shannon. 'Prediction and Entropy of Printed English'. en. In: *Bell System Technical Journal* (Jan. 1951).

[73] Ze Sheng et al. 'LLMs in Software Security: A Survey of Vulnerability Detection Techniques and Insights'. en. In: *ACM Computing Surveys* 58.5 (Nov. 2025), pp. 1–35. ISSN: 0360-0300, 1557-7341. DOI: 10.1145/3769082. URL: https://dl.acm.org/doi/10.1145/3769082 (visited on 14/01/2026).

[74] Alexey Shestov et al. *Finetuning Large Language Models for Vulnerability Detection*. en. arXiv:2401.17010 [cs]. July 2024. DOI: 10.48550/arXiv.2401.17010. URL: http://arxiv.org/abs/2401.17010 (visited on 30/01/2026).

[75] *Snyk AI Security Fabric | Secure Code, Models & Agents*. en-US. URL: https://snyk.io/ (visited on 12/02/2026).

[76] Cole Stryker and Eda Kavlakoglu. *What Is Artificial Intelligence (AI)? | IBM*. en. URL: https://www.ibm.com/think/topics/artificial-intelligence (visited on 12/02/2026).

[77] *Supervised fine-tuning | OpenAI API*. en. URL: https://developers.openai.com/api/docs/guides/supervised-fine-tuning/ (visited on 24/02/2026).

[78] Gemini Team et al. *Gemini: A Family of Highly Capable Multimodal Models*. arXiv:2312.11805 [cs]. May 2025. DOI: 10.48550/arXiv.2312.11805. URL: http://arxiv.org/abs/2312.11805 (visited on 12/02/2026).

[79] *Unsloth - Train and Run Models Locally*. URL: https://unsloth.ai/ (visited on 24/03/2026).

[80] *Unsloth Notebooks | Unsloth Documentation*. en. Mar. 2026. URL: https://unsloth.ai/docs/get-started/unsloth-notebooks (visited on 24/03/2026).

[81] Ashish Vaswani et al. *Attention Is All You Need*. en. arXiv:1706.03762 [cs.CL]. Aug. 2023. DOI: 10.48550/arXiv.1706.03762. URL: http://arxiv.org/abs/1706.03762 (visited on 26/05/2026).

[82] *Vi presenterar GPT-5.4*. sv-SE. Apr. 2026. URL: https://openai.com/sv-SE/index/introducing-gpt-5-4/ (visited on 14/04/2026).

[83] Jason Wei et al. *Chain-of-Thought Prompting Elicits Reasoning in Large Language Models*. en. arXiv:2201.11903 [cs]. Jan. 2023. DOI: 10.48550/arXiv.2201.11903. URL: http://arxiv.org/abs/2201.11903 (visited on 30/01/2026).

[84] Ratnadira Widyasari et al. *Let the Trial Begin: A Mock-Court Approach to Vulnerability Detection using LLM-Based Agents*. en. arXiv:2505.10961 [cs]. Dec. 2025. DOI: 10.1145/3744916.3773256. URL: http://arxiv.org/abs/2505.10961 (visited on 14/01/2026).

[85] Alperen Yildiz et al. *Benchmarking LLMs and LLM-based Agents in Practical Vulnerability Detection for Code Repositories*. en. arXiv:2503.03586 [cs]. Mar. 2025. DOI: 10.48550/arXiv.2503.03586. URL: http://arxiv.org/abs/2503.03586 (visited on 14/01/2026).

[86] Xue Ying. 'An Overview of Overfitting and its Solutions'. en. In: *Journal of Physics: Conference Series* 1168 (Feb. 2019), p. 022022. ISSN: 1742-6588, 1742-6596. DOI: 10.1088/1742-6596/1168/2/022022. URL: https://iopscience.iop.org/article/10.1088/1742-6596/1168/2/022022 (visited on 31/03/2026).

[87] Yaqin Zhou et al. *Devign: Effective Vulnerability Identification by Learning Comprehensive Program Semantics via Graph Neural Networks*. en. arXiv:1909.03496 [cs]. Sept. 2019. DOI: 10.48550/arXiv.1909.03496. URL: http://arxiv.org/abs/1909.03496 (visited on 04/02/2026).