Arab American University

Faculty of Graduate Studies

Department of Natural , Engineering and Technology Sciences

Master Program in Computer Science

A Comprehensive Framework for Software Vulnerability Prediction Using Large Language Models

Wala'a Yusuf Ata Shehada
202120189

Supervision Committee:

Dr. Huthaifa I Ashqar

Dr. Omar Darwish

Dr.Mahmoud Obaid

Dr.Hamed Abdelhaq

This Thesis Was Submitted in Partial Fulfillment of the Requirements for the Master Degree in Computer Science

Palestine, 2 / 2025

Arab American University
Faculty of Graduate Studies
Department of Natural, Engineering and Technology Sciences
Master Program in Computer Science

Thesis Approval

A Comprehensive Framework for Software Vulnerability Prediction Using Large Language Models

Wala'a yusuf ata shehada

202120189

This thesis was defended successfully on 23 / 2 / 2025 and approved by:

Thesis Committee Members:

	Name	Title	Signature
1.	Dr . Huthifa I Ashqar	Main Supervisor	
2.	Dr .Omar Darwish	Member of Supervision Committee	
3.	Dr.Mahmoud Obaid	Member of Supervision Committee	Mahmoud Obaid
4.	Dr.Hamed Abdelhaq	Member of Supervision Committee	

Palestine, 2 / 2025

Declaration

I declare that, except where explicit reference is made to the contribution of others, this thesis is substantially my own work and has not been submitted for any other degree at the Arab American University or any other institution.

Student Name : Wala'a Yusuf Ata Shehada

Student ID: 202120189

Signature: Wala'a

Date of Submitting the Final Version of the Thesis: 11-6-2025

Dedication

Alhamdulillah, who taught humankind what he did not know, and Alhamdulillah, who gave me the ability to complete this work, to Allah, belongs the favor first and last.

To my mother and father, I dedicate this work to you in gratitude for your countless favors. They always supported and backed me, instilled in me a love of knowledge and work, and worked hard and stayed up late for me.

To my beloved son Ashraf, your little smile motivated me to continue my journey despite the fatigue. I dedicate this humble work to you and hope it will be the start of pride for both of us.

To my dear brothers and sisters, who were the source of warmth and constant support

Thank you to my doctors and colleagues at the Arab American University for your continuous support and inspiration. I am also grateful to my friends who joined me on this journey and believed in my abilities. I am also thankful to the unknown soldiers who contributed to the success of this work. Without your help, I would not have been able to do it.

I dedicate this humble work to you all.

Wala'a Yusuf Ata Shehada

Acknowledgments

Thanks to God for enabling me to complete this work. I thank everyone who contributed to achieving this work with scientific or moral support.

First and foremost, I want to thank my supervisors, Dr. Huthaifa I Ashqar and Dr. Omar Darwish, for their significant role in guiding and directing me during the research phases and giving me invaluable time and effort; their confidence and encouragement were the primary sources of my completion of this achievement.

I also thank all my professors at the Arab American University for their valuable information.

Special thanks to my honorable family for their continuous support.

Last, I thank my study colleagues who shared this journey with me in its challenging and

rewarding moments.

Thank You All

A Comprehensive Framework for Software Vulnerability Prediction Using Large Language Models

Wala'a Yusuf Ata Shehada
Dr. Huthaifa I Ashqar
Dr. Omar Darwish
Dr.Mahmoud Obaid
Dr.Hamed Abdelhaq

Abstract

Numerous threats have emerged in our digital age, most notably software vulnerabilities and network intrusions. These threats can result in significant financial losses, data breaches, and system disruptions across various industries. due to the complexity and evolving nature of cyberattacks, addressing them requires new and advanced mechanisms to detect or prevent them before they occur. LLMs such as GPT, LLaMA, and BERT were evaluated for their effectiveness in classifying software code as vulnerable or non-vulnerable, and for intrusion detection, classical machine learning algorithms were employed on the NSL-KDD dataset. This thesis contributes to enhancing cybersecurity by addressing large language models and machine learning models in detecting software vulnerabilities and intrusions carefully and accurately. It also suggests prospects for improving the results and expanding the study.

Our study focused on two main goals: predicting software vulnerabilities using large language models and detecting network intrusions using machine learning models. To predict software vulnerabilities, we used large language models (GPT, Llama, and BERT) to analyze and classify the software code as normal or abnormal. The DiverseVul dataset was used and test data of three different sizes (1000, 5000, and 20,000 records) were extracted from the DiverseVul dataset. The performance of GPT and Llama was compared in the zero-shot and few-shot cases, and we noticed that GPT outperformed Llama in all cases. When comparing the performance of the CodeBERT-5000 and the CodeBERT-1000 with GPT and Llama, we noticed that the Codebert-5000 achieved the best results with an accuracy of 79.2%, which is considered suitable for a complex task like software vulnerabilities.

The main contribution of this study is to conduct three main experiments aimed at studying the performance of the models and providing deeper insights into their efficiency and reliability: (1) Consistency Check to ensure the stability of the models by repeating the experiment several

times; (2) Analyzing the ability of the GPT and Llama models to explain their predictions and understanding the logic behind predictions. and (3) measure the latency of each model when predicting 1000 records. These experiments aim to deeply analyze the models from multiple aspects, enhancing the results' reliability.

As for the second section, the network intrusion detection section used machine learning algorithms (“Logistic Regression, Random Forest, Support Vector Machine, and Decision Tree”) on the NSL KDD data to categorize the network traffic into normal and abnormal.

The results showed ideal performance for the Decision Tree model, which obtained an accuracy of 100%, and the Random Forest model, which had a rate of 99.8% higher than the Support Vector Machine and Logistic Regression models.

Keywords: Software Vulnerability, Large Language Models, Network Intrusion, Machine Learning.

Table of Contents

#	Title	Page

List of Tables

List of Figures

List of Definitions of Abbreviations

Abbreviations	Title
IDS	Intrusion detection system
AI	Artificial Intelligence
API	Application Programming Interface
APR	Automatic Program Repair
BERT	Bidirectional Encoder Representations from Transformers
BiLSTM	Bidirectional Long Short-Term Memory
BLEU	Bilingual Evaluation Understudy
Llama	Large Language Model Meta AI
LSTM	Long Short-Term Memory
CNN	Convolutional Neural Network
CWE	Common Weaknesses Enumeration
CodeBERT	extension of the BERT model for programming language and natural language
CoT	Chain of Thought
DAST	Dynamic Application Security Testing
DNN	Deep Neural Network
GPT	Generative Pre-trained Transformer
GRU	Gated Recurrent Unit
IAST	Interactive Application Security Testing
ICL	In-Context Learning
SDLC	Software Development Life Cycle
MLM	Masked Language Modeling
HPC	high-performance computing
GSZL	generalized zero-shot learning
LLM	Large Language Model
LMP	Language Model Programming
NLP	Natural Language Processing
NN	Neural Network
RNN	Recurrent Neural Network

SAST	Static Application Security Testing
SQL	Structured Query Language
URL	Uniform Resource Locator
HTTP	Hypertext Transfer Protocol
XML	Extensible Markup Language
XSS	Cross-Site Scripting
BGRU	Bidirectional Gated Recurrent Unit Neural Network
DL	Deep Learning
ROC-AUC	Receiver Operating Characteristic Area Under the Curve.
ML	Machine Learning
RNN	Recurrent Neural Network
GNN	Graph neural networks
CVEs	Common Vulnerabilities and Exposures
LR	Logistic Regression,
RF	Random Forest
SVM	Support Vector Machine
DT	Decision Tree
NSL-KDD	Network Security Laboratory KDD (standard benchmark for intrusion detection evaluation)
TF-IDF	Term Frequency -Inverse Dense Frequency
PL	Programming Language
NL	Natural Language
NSP	Next Sentence Prediction
HIDS	Host-Based Intrusion Detection System
NIDS	Network Intrusion Detection System
DoS	Denial-of-service
MitM	Man-in-the-middle
IDE	Integrated Development Environment
REST	Representational State Transfer

Chapter One: Introduction

Significant advancements in Internet and communication technologies, which have become essential parts of our lives, have driven the expansion of information and networks. This expansion of information and networks made them vulnerable to attacks by many criminals who developed many types of them, this creates a strong motivation to enhance cybersecurity mechanisms.

An intrusion detection system (IDS) is designed to protect the network and prevent unwanted access by detecting suspicious activities and issuing alerts upon discovery to take the necessary measures to address the threat; in this study, we aim to detect network intrusion using machine learning (ML) techniques ("Logistic Regression (LR), Support Vector Machine (SVM), Radom Forest (RF), and Decision Tree (DT)") on the NSL-KDD dataset. Because software applications form the foundation of many industries today, they drive innovation and efficiency. However, the risk of cyber threats has increased due to this reliance, as hackers and malicious actors are constantly trying to exploit software vulnerabilities (Z. Ahmad et al., 2021).

One of the key challenges is that traditional vulnerability detection methods are no longer effective against attackers due to their development, dynamism, and reliance on fixed rules and predefined vulnerability data. to address this, we explore the use of large language models (LLMs), such as GPT and Llama, which have demonstrated remarkable capabilities in understanding and analyzing natural language. The methodology involves using the DiverseVul dataset to evaluate LLMs in identifying vulnerabilities in code

The main contribution of this study is to conduct three main experiments aimed at studying the performance of the models and providing deeper insights into their efficiency and reliability: (1) Consistency Check to ensure the stability of the models by repeating the experiment several times; (2) Analyzing the ability of the GPT and Llama models to explain their predictions and understanding the logic behind predictions. and (3) measure the temporal performance (latency) of models in predicting 1000 records

These experiments aim to analyze the models deeply from multiple perspectives, enhancing the results' reliability. Because innovative methods are needed to detect complex vulnerabilities, LLMs are considered the best candidates to address this problem due to their superior multitasking and few-shot learning capabilities.

1.1 Background

As software applications grow in size and complexity, discovering software vulnerabilities has become challenging, as understanding the relationship between different attack elements has become difficult. Because software systems have invaded most aspects of our lives, any threat, even a minor one, can affect organizations, individuals, or IT infrastructure. Therefore, discovering and preventing these vulnerabilities before they occur is the focus of our research(G. Lin et al., 2020). Current approaches for detecting software vulnerabilities are practical, but each has its own set of limitations. Static Application Security Testing (SAST) tools frequently generate false positives and lack comprehensive end-to-end testing. Dynamic Application Security Testing DAST approaches typically offer low true positive rates and restricted coverage, whereas Interactive Application Security Testing IAST tools are particular to a language and need interaction. Hybrid (Guo et al., 2022a)and combination (Mateo Tudela et al., 2020). techniques have been proposed to address these limitations, showing gains over employing a single method. Nonetheless, these methods struggle to identify critical weaknesses that require contextual understanding and logical reasoning.

Network intrusion is another security challenge that is as serious as software vulnerability. With the current development of networks, their expansion, and increased interconnection, the possibility of network intrusion and unauthorized access increases, leaving harmful effects and potential risks to data confidentiality and availability. Therefore, proactively detecting and preventing network intrusions is a crucial aspect of modern cybersecurity. Traditional intrusion detection systems, such as signature-based methods, rely on known patterns, thus struggling to identify new ones. Here, machine learning algorithms are more adaptive by analyzing the movement of extensive data sets on the network to detect any potential threat. Machine learning models are commonly used to classify network activity into benign and malicious due to their ability to learn from classified data like the NSL-KDD dataset to recognize patterns known as assaults. This improves the system's ability to detect intrusion in real-time.

LLMs are increasingly common AI models. Commercial and open-source pre-trained LLMs with billions of parameters make applying AI models to under-investigated sectors like vulnerability detection more cost-effective, easy, and faster than before. These models anticipate the following word in a text sequence (a prompt). Their innovative architecture enables them to thrive in various NLP tasks without prior training.

We study LLMs' ability to analyze code snippets to predict and highlight the parts that pose security risks. We first collect a dataset of software vulnerabilities from previous studies and then apply different types of LLMs known for their superior natural language processing capabilities, such as GPT, Llama, and BERT (Vaswani, 2017), (Devlin, 2018).

1.2 Problem

Traditional methods for detecting software vulnerabilities are no longer sufficient to find complex vulnerabilities. Traditional methodologies such as static app security testing (SAST), dynamic security testing of applications (DAST), and interaction app security testing (IAST) have numerous limitations, including high rates of false alarms along with low genuine alert rates. Despite hybrid and composite model advancements, they remain limited in discovering vulnerabilities that require a broad contextual understanding. Also, Intrusion can be considered a real problem that threatens network security because any simple intrusion can lead to the loss and theft of essential data from the network or computer systems within seconds. Intrusion can also cause damage to the system's equipment and material losses (I. Ahmad et al., 2018; Alrowaily et al., 2019). Therefore, preventing and detecting intrusion is necessary for administrators to protect their organization's network from any harm that would cause damage at the physical or data level (Javaid et al., 2016). This is where the research problem is.

Our research seeks to close this gap by leveraging the capabilities of LLMs to provide a complete framework for detecting software vulnerabilities and developing a system using machine learning (ML) algorithms to detect intrusion on the NSL-KDD dataset. The research aims to address the following questions:

1. How effective are GPT and Llama models in detecting software vulnerabilities, measured by precision and recall, across different programming languages?

2. What are the main elements that affect how well the suggested model detects software vulnerabilities? These elements include data quality, model type (GPT, LLaMA, CodeBERT), and training strategy (zero-shot, few-shot, fine-tuning)

3. What are the challenges and limitations of the proposed LLM-based vulnerability detection framework, and how does it address the biases of traditional methods?

4. How do different machine learning algorithms, such as Logistic Regression, Random Forest, Support Vector Machine, and Decision Tree, perform in detecting network intrusions and monitoring data flow?"

5. What is the best algorithm based on accuracy to classify network traffic into abnormal and normal?

1.3 Purpose

Our primary goal in this thesis is to create a comprehensive security system to detect software vulnerabilities using (LLM). Thus, reducing cybersecurity risks and increasing security for organizations that use software applications. We also aim to determine the critical role of ML algorithms in detecting network intrusion by protecting and monitoring the network from any intrusion that threatens it. We are developing a system using machine learning (ML) algorithms to detect intrusion on the NSL-KDD dataset.

In addition, the thesis addresses the use of (LLM) in detecting software vulnerabilities, measuring their effectiveness, and overcoming the challenges of traditional methods due to the ability of LLM to process natural language and analyze code segments. The thesis will explain the mechanism for using LLMs trained on massive data to predict and detect software vulnerabilities in cases of zero and few shots and fine-tuning LLMs if possible. In addition, we seek to solve the problem of biases in old methods, provide recommendations for future studies in the field, and enrich the field of cybersecurity.

1.4 Goals

This thesis aims to discover and identify software vulnerabilities automatically without the need for manual methods by experts by using LLMs, in addition to Using various machine learning algorithms to detect network intrusion. as follows:

1. Verify the effectiveness of the LLM in identifying software vulnerabilities.
2. Determine the influencing elements of the effectiveness of our proposed model.
3. Identify the challenges and limits of the suggested paradigm.

4. Trying to solve an existing problem in network intrusion and monitoring the traffic and flow of data by employing different ML algorithms.
5. Determine the best algorithm based on Accuracy to create a model capable of classifying network traffic into abnormal and normal.

1.5 Methodology

We used quantitative experimental research in three stages: exploration, experimentation, and evaluation. We conducted a comprehensive literature review on software vulnerabilities and intrusion detection systems. We selected two datasets from previous studies. The first is the NSL-KDD dataset for detecting intrusion, and the second is the DiverseVul dataset for software vulnerabilities which contains code instructions with known software vulnerabilities, followed by a pre-processing process for the data and converting it into a suitable form for entering it into ML & LLMs. Then we chose suitable models from ML such as (LR, RF, SVM, and DT), and from LLMs such as GPT, Llama, and BERT to detect software vulnerabilities.

In this work, we suggest using large language models (LLMs) to identify vulnerabilities in software. In particular, we apply GPT and LLaMA to few-shot and zero-shot learning scenarios. We also finetune GPT and CodeBERT with domain-specific vulnerability data with different dataset sizes (1,000, and 5,000 samples). We then assess these models' efficacy in detecting software vulnerabilities by comparing their performance across various data sizes.

Initially, the model should be evaluated in zero-shot learning, followed by few-shot learning, and, if feasible, fine-tuned using domain-specific data related to software vulnerabilities to achieve optimal accuracy. Dividing the dataset into testing, training, and validation to ensure the effectiveness of the fine-tuned LLMs, we first train these large models on the training set, then verify their efficacy through the validation set, then finally, the testing process on a dataset that the model has not seen before, and finally using appropriate measures such as accuracy, precision, recall, and F1score.

1.6 Limitations

This study's limitation is finding software vulnerabilities in the code using LLMs. The research did not work to find all possible types of vulnerabilities in the programs, but it was limited to 150 types. The study was limited to using three trained models, GPT, Llama, and BERT, and did not include all possible models. In addition, the study was limited to evaluation by specific metrics such as accuracy, recall, and precision, and the study was conducted in a controlled environment instead of factual scenarios. Finally, it is necessary to address the resource constraints required to effectively train large LLMs such as GPT, Llama, and BERT, as these models require robust infrastructure such as GPUs, large memory, and long computation time, which was limited during the study, these constraints may limit further improvements or evaluation of multiple scenarios.

Also, there are many challenges in detecting network intrusions using traditional machine learning algorithms. The first is that using specific data for the training process may limit the models' effectiveness and results because many attacks in the real world are not adequately represented in the selected data, as the data focuses on specific types of breaches and does not include all possible breaches. In addition, we limited ourselves to machine learning models in this part of the thesis, and performance may differ when using more modern models. In addition, we limited ourselves to using metrics such as accuracy, precision, and recall, which may ignore other important indicators such as the false alarm rate.

1.7 Structure of the Thesis

Chapter 2 contains extensive background material on software vulnerability detection techniques, intrusion detection systems, and large language models. Chapter 3 describes the research methodologies used in the study. Chapter 4 summarizes the test results and examines the recommended method. Chapter 5 comments on the analysis and discusses the findings in light of the study's questions and objectives. Chapter 6 concludes the study with the author's comments and suggestions for future research.

Chapter Two: Literature Review

Background

This section provides an overview of LLMs and the fundamental AI concepts underlying their development. Furthermore, this chapter discusses software vulnerabilities, network intrusion, and how to detect them. The section also covers work in large language models, including improving software vulnerability detection algorithms and machine learning models in network intrusion prediction.

2.1 Large Language Models (LLMs)

A large language model (LLM) is an advanced form of language model that has been trained on massive amounts of data and can produce and translate text and other content, as well as conduct other natural language processing (NLP) tasks. LLMs are often built on deep learning architectures, notably RNNs (recurrent neural networks) and transformer structures (Vaswani, 2017), and trained on billions of tokens and other data points. LLMs have altered natural language processing (NLP) activities, enabling breakthroughs in various domains including text production and translation (Radford et al., 2019a), medical diagnosis (C. Xu et al., 2022), question answering (Lewis, 2019), and others. Their potential in software engineering, particularly in software vulnerability discovery, has recently garnered attention. LLMs can examine the source code, documentation, and security warnings for patterns that signal vulnerabilities. Despite challenges such as ethical issues and computational demands, the current research aims to increase the efficiency, ethics, and interpretability of LLMs, creating the foundation for safer and more trustworthy software systems. Understanding LLM's significance requires prior awareness of the limitations of their predecessors. This section will examine the fundamentals behind LLMs and why they are useful in AI.

2.1.1 Deep Neural Networks

(DNNs) considered the theoretical and technical foundation on which LLMs depend. LLMs use deep neural networks consisting of many layers to learn from extensive data, which allows for understanding and analyzing context. DNNs were among the first breakthrough AI models. The key difference between classic machine learning models is the complexity and breadth of the networks (LeCun et al., 2015) Figure 1 depicts the use of many interconnected hidden layers in DNNs. Deep learning approaches use extra layers to teach the network how to tackle increasingly challenging problems and enhance its performance across various activities. By adding additional hidden layers, the network may learn more features, increasing the model's accuracy. However, as the number of hidden layers increases, so does the model's computational complexity, putting extra load on the hardware.

Another negative consequence is the presence of gradients that vanish or explode. This happens at the backpropagation phase when a network's variables are updated using the gradients of the loss function. The network's overall learning rate decreases when these gradients get too tiny (vanishing) or too huge (exploding).

In the end, adding additional layers will yield marginal performance gains due to decreasing returns. Even so, some deep learning models solve these difficulties and exceed basic DNN approaches.

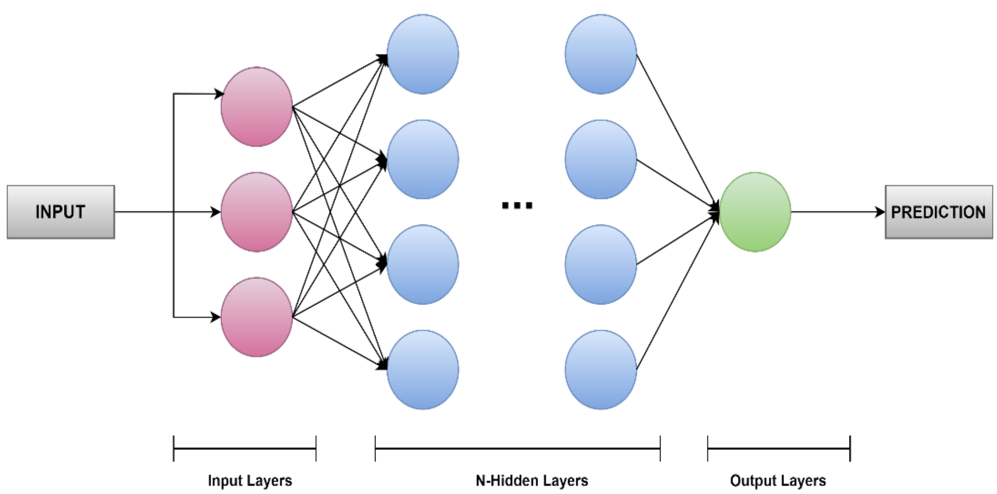

Figure 2.1: Deep Neural Network Architecture

2.1.1.1 Recurrent Neural Networks (RNN)

RNNs are a kind of neural network that processes data sequentially while maintaining a hidden state that contains details about previous inputs. RNNs provided the theoretical basis for understanding texts as interconnected chains. They are particularly well-suited for activities requiring precise input order, such as language modeling, speech recognition, and time series prediction, (Rumelhart et al., 1986)Recurrent neural networks' hidden state is updated based on the current stage's incoming inputs and its prior hidden state, allowing for the analysis of sequences of varying lengths. The hidden state with the feature vector is introduced as an input during the network's training process, allowing recurrent neural networks to better capture complex data linkages and dependencies, increasing prediction accuracy.

The problem of vanishing gradients is one of the difficulties encountered when using recurrent neural networks (Hu et al., 2021). They contract as gradients pass from the output layer to the input layer. Adjusting the weights becomes inefficient as they inevitably become smaller. This difficulty is exacerbated by the shared weights of the hidden state overall time cycles. This, in turn, makes it difficult for recurrent neural networks to acquire long-term dependencies. When the gradients flow backward through the layers, they diminish, preventing the network from being trained accurately (Bengio et al., 1994).

2.1.1.2 Long Short-Term Memory Networks (LSTM)

LSTM is a version of RNN that was developed to address gradient-vanishing issues and can handle sequential input with long-term dependencies, Hochreiter and Schmidhuber developed Long Short-Term Memory, an upgraded version of the recurrent neural network.

A typical RNN contains a single hidden state that is updated over time, making it difficult for the network to learn long-term dependencies. The LSTM model addresses this issue by incorporating a memory cell, which is a container capable of storing information for an extended length of time. LSTM architectures can learn long-term dependencies in sequential data, making them ideal for tasks like language translation, recognition of speech, and time series forecasting. The LSTM architectures have a memory cell that is managed by three gates: the input gate, the forget gate, and the output gate. These gates determine what information is added to, removed from, and output from the memory cell. The input gate determines what data is added to the memory cell. The forget gate determines what data is erased from the memory cell. The output gate determines what data is output by the memory cell.(Van Houdt et al., 2020). GRU networks can be considered as an alternative to LSTM networks due to their simplified architecture compared to LSTM. In solving the vanishing gradient issue, GRU networks contain just two gates, an update gate, and a reset gate, this renders them less complex than LSTM and requires less training time, yet they still capture long-term dependencies. This makes them a more efficient and powerful solution for tasks such as natural language processing (Staudemeyer & Morris, 2019).

2.1.1.3 Attention Mechanisms

In neural networks, attention mechanisms are important for tasks involved in natural language processing. They allow models to generate output based on multiple aspects of the input data, boosting the capacity to capture connections and linkages within the data. Query, key, and value vectors serve as key factors in attention mechanisms because they are used to calculate attention scores, which indicate the significance of various input elements. Dot-product attention scales the dot product of query and key vectors, whereas additive attention computes scores using a feed-forward network. Attention mechanisms were Introduced by Bahdanau et al, in 2014 as a strategy to solve flaws in what was then cutting-edge recurrent neural network (RNN) models used for translation by machine. Subsequent research included attention mechanisms into convolutional neural networks (CNNs) utilized for tasks like image captioning and visual question responding. This study shows (Mnih et al., 2014) Self-attention enables each point in the input sequence to be attentive to every possible position, making it excellent for capturing dependencies over distance(Shaw et al., 2018). Multi-head attention improves this by dividing the vectors into numerous heads, each with a unique attention score. Attention mechanisms have transformed tasks such as machine translation, summarizing texts, question answering, and picture captioning by allowing models to identify long-range correlations while providing interpretability via attention weights. See equation (1)

$$\boldsymbol{Attention(Q,K,V) = softmax\left(\frac{QK^{\top}}{\sqrt{\{d_k\}}}\right)V} \quad \textbf{(1)}$$

Where: $\boldsymbol{Q}$(Queries) for the matrix of queries with dimension $\boldsymbol{d_k}$. $\boldsymbol{K}$ (Keys) The matrix of keys also of dimension $\boldsymbol{d_k}$. $\boldsymbol{V}$(Values) The matrix of values where each is of dimension $\boldsymbol{d_v}$. $\boldsymbol{d_k}$ The dimensionality of the query and key vectors serve to scale the dot product to avoid overly large values. $\boldsymbol{d_v}$ The dimensionality of value vectors.

Using attention mechanisms in DNNs has the drawback of making computations more difficult. The processing power needed increases quadratically with the input sequences, especially with self-attention, leading to much longer training times and more significant costs. It is imperative to tackle memory and processing limitations early when working with extensive datasets.

2.1.2 Transformers

Vaswani et al. introduced the Transformer, a groundbreaking neural network architecture, in 2017 (Vaswani, 2017), which is notable for its efficiency and capacity to manage long-term dependencies in sequential data. Unlike classic models like RNNs and LSTMs, which evaluate outcomes sequentially, the Transformer is entirely built on self-attention processes, allowing it to examine every sequence part simultaneously while capturing relationships across distances. The design incorporates an encoder and a decoder, each with multiple layers. Each layer includes multi-head self-attention processes, which employ many attention heads in parallel to devote attention to distinct components of the series and position-wise feed-forward networks, which boost the model's capacity to learn complex representations. Positional encoding is used on input-embedded data to provide information about word order, compensating for the lack of natural sequential processing. Residual linkages and layer normalization increase training speed and stability. This method enables the Transformer to excel at various natural language processing applications, including translation, text summarizing, and question responding, while providing considerable benefits in training efficiency, parallelization, and scalability. The Transformer maintains this basic architecture by using layered self-attention and point-wise, completely connected layers for the decoder and encoder, as in Figure 2.2, left and right sides.

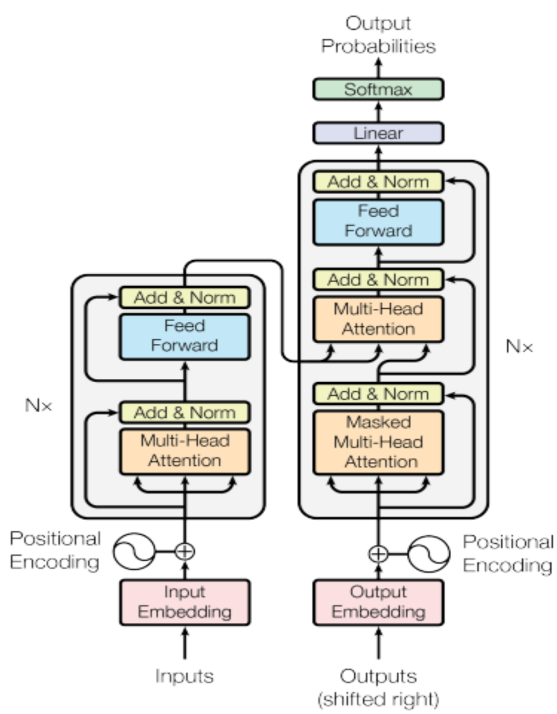

Figure 2.2: The Transformer architecture (Vaswani, 2017)

Transformers have proven helpful in a variety of industries. DALL-E is an AI model used in computer vision research to produce images from natural language text descriptions (Ramesh et al., 2021) The transformer design can also be used in computer vision to recognize images (Dosovitskiy, 2020). In speech analysis, the Conformer model (Gulati et al., 2020), which is built-in transformers, is taught to recognize speech. Transformers have been employed in pharmaceutical research to anticipate molecular interactions and develop new therapeutic medications (Shin et al., 2019)Although the transformer architecture is used in each of these applications, the whole structure is implemented differently. Transformer designs can be employed in encoder-only, decoder-only, or encoder-decoder networks, depending on the purpose. To understand the transformer architecture more straightforwardly, see Figure 2.3.

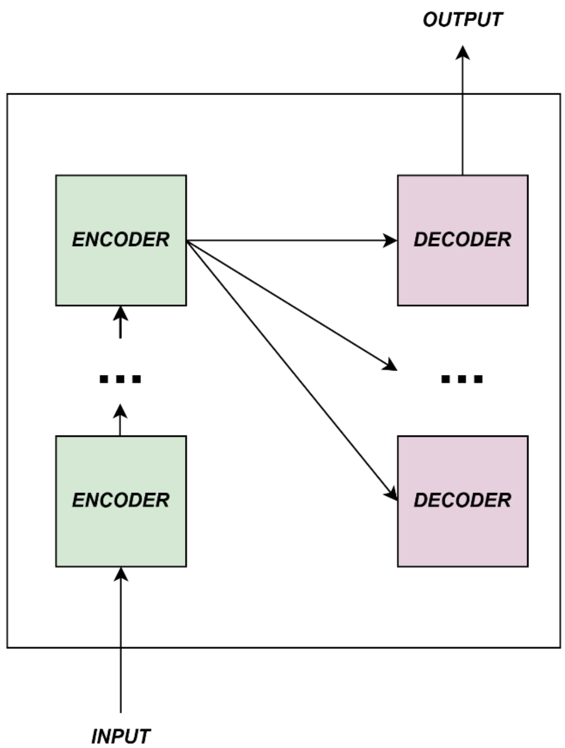

Figure 2.3: simplified representation of the Transformer architecture.

2.1.2.1 Encoder-only transformers

Encoder-only transformers are customized neural network topologies that solely use the transformer model's encoder component, making them ideal for jobs requiring a complete understanding and representation of input data. These systems comprise many layers, including multi-head self-attention mechanisms and position-wise feed-forward networks, which capture complex dependencies and contextual information from an input sequence. They use positional encoding to retain tokens in the correct order and multi-head attention to learn about different aspects of token relationships. Text categorization, named entity recognition, and document retrieval are all applications that depend primarily on understanding the input context.

Bidirectional Encoding Representations from Transformers (BERT) is an (LLM) built using an encoder-only transformer structure (Kenton & Toutanova, 2019a). It is pre-trained using a masked language simulation technique, in which some tokens in the input stream are periodically obscured while the model learns to predict them. The main idea behind such pre-training lets BERT generate complex symbol connections with left or right contexts. BERT encouraged the creation of many newer LLM models on top of its foundation: RoBERTa improves BERT by applying different masking dynamically together with larger-size batches to result in a good accuracy score by (Liu, 2019), and ALBERT reduces the usage of memory instead of improving model performance by (Lan, 2019). Other BERT variants like DistilBERT and MiniLM compress the model using information distillation approaches; such compression enables running it on lower-end hardware with performance equivalent to its bigger counterparts(Sanh, 2019) (W. Wang et al., 2020).

2.1.2.2 Decoder-only transformers

They are the type of neural network structure designed purely on the transformer model's decoder and, thus, very suitable for applications that mostly involve sequence generation, such as text generation, translation, and summarization. These models are special in that they comprise multiple layers with self-attention mechanisms with position-wise feed-forward networks through which they can generate a coherent and contextually relevant sequence by attending to all prior tokens. GPT, GPT-2, and GPT-3 Use decoder-only transformer design (Brown, 2020a), (Radford et al., 2019b). They are pre-trained on very large text datasets to be able to guess the immediate next token in a sequence. Hence, they have developed expertise in producing pretty much human-like writing and, thus, perform better than pure decoder transformers in every application of NLP where sequence generation is involved. Decoder-only transformers are most efficient for sequence generation, maintaining contextual coherence, scalability, and versatility across several applications of natural language processing, such as masking self-attention so that tokens cannot attend to future tokens and positional encoding to keep the order of tokens.

2.1.2.3 Encoder-Decoder transformers

The neural networks that use a transformer model decoder and an encoder are well suited for sequence comprehension and production tasks, for instance, automated translation, text summarization, and question answering. The encoder provides the input sequence with a context-rich representation that will be used by the decoder to generate the output sequence. It contains self-attention mechanisms as well as position feedforward networks. In addition, the decoder uses encoder-decoder attention to attend to the output of the encoder. According to (Chefer et al., 2021)Such an architecture enables the use of encoder-decoder converters for complex mapping of input relationships to contextually appropriate output generation

2.1.3 Pre-training

The first stage of training an LLM, in which it learn from a vast, diverse dataset containing billions of tokens. The purpose here is to gain a comprehensive understanding of language, and different sorts of information. Pre-training is typically extremely computationally intensive and needs massive volumes of data.

2.1.4 Finetuning

Where you take a previously learned model and train it on a more precise dataset. This dataset is usually smaller and focuses on a certain subject or job. The goal of fine-tuning is to improve the model's performance in specific situations or on activities that were not fully addressed by pre-training. The additional knowledge supplied during fine-tuning focuses on improving the model's efficiency in specific settings rather than broadening its general understanding

(Wei et al., 2021),(Radford, 2018). Therefore, the fine-tuning process has been considered a qualitative leap to enhance the efficiency of models for specific tasks, as this data is explained and customized for the required application. At this stage, the model applies the knowledge it gained from pre-training and benefits from it in the specific task. Throughout the fine-tuning procedure, the model's predictions are compared with the correct outputs, and adjustments are made to minimize errors. Thus, it uses the supervised learning process, see Figure 2.4.

LLM fine-tuning is an effective strategy for adapting them to particular fields and activities. However, fine-tuning has significant obstacles and drawbacks that must be addressed before being used for real-world tasks. Fine-tuning necessitates superior, comprehensive training data corresponding to the desired field and purpose. Quality data is accurate, reliable, regular, and sufficiently varied to encompass every potential scenario and change that the system may meet in the actual world. Poor-quality or inaccurate data causes overfitting and underfitting, reducing generalization and robustness. Fine-tuning LLM incurs additional costs related to training or maintaining the customized model. Fine-tuning might also require repetition anytime the data changes or a novel basic model is available. This includes regular monitoring and upgrading.

Finally, Fine-tuning is a recurring job (trial and error), so the hyperparameters must be carefully selected. Fine-tuning necessitates extensive investigation and testing to determine the optimal mix of the hyperparameters and variables for achieving the desired efficiency and quality.

In conclusion, despite its difficulty, Fine-tuned models produce optimal results in various application cases, illustrating the flexibility and significance of fine-tuning in expanding LLM abilities for particular business applications.

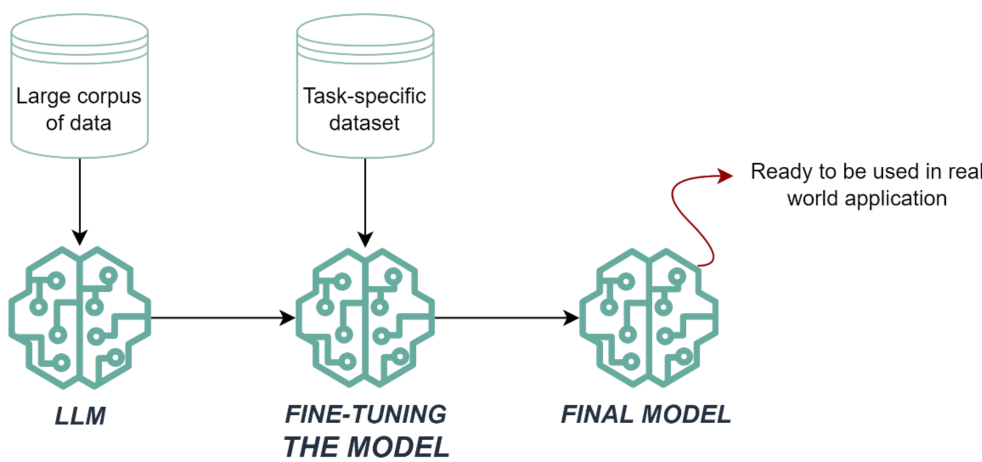

Figure 2.4: A Visualization of the Fine-Tuning Procedure.

2.1.5 Prompting

Prompting is essential in dealing with (LLMs). It requires offering particular and explicit instructions to direct the results of the model. When giving the model a specific task, the question must be clear and specific to direct the model to answer correctly. Including more details helps to define the task more accurately and thus enables the model to focus on information relevant to the topic and provide more accurate results, see Figure 5. In addition, defining the role of the model as an expert in a specific field can help improve the results significantly (White et al., 2023). For example, the question formula in the case of Zero-Shot was as follows: "As a software engineer, I need a short answer (yes, no) only if there is a software vulnerability in the code snippet provided in the user message," where we asked the model to act as a software engineer, analyze the code, and answer the question. In addition, giving the model examples can help the model answer more accurately, as in the case of a few shots.

While this strategy allows for in-context learning of LLMs, it adds the computational cost of model inference and the human work of manually crafted prompts, mainly when long and complicated prompts are used to direct and manage LLM activity. As a result, the LLM sector has seen a significant increase in effective prompting techniques (Chang et al., 2024).

Controlling the length of the prompt is also essential; keeping it brief and ensuring that every piece of information is useful keeps the model from being overwhelmed or confused (Giray, 2023). Various prompt engineering strategies can help to improve prompting. Zero-shot prompting asks the model to complete a job without prior instances, relying on the model's general understanding. Few-shot prompting uses a few samples to indicate the desired outcome, allowing the model to learn from them (Yong et al., 2023). Chain of thought prompting (CoT) (Z. Zhang et al., 2022) Directs the model through a logical sequence or thought process, which is beneficial for challenging problem-solving activities. According to the request, the model could require more information and context to respond. For instance, a model trained with data from 2022 cannot offer reliable predictions for a scenario in 2023. In such situations, data unknown to the model might be included in the prompt and utilized to act as an oracle to the model (Sakaoglu, 2023)The technique can be adapted to teach additional tasks within the model using ICL (In-context learning).

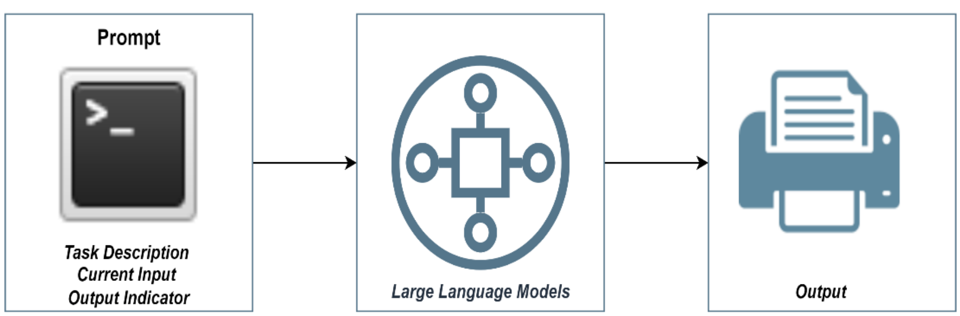

Figure 2.5: Overview of Prompt Engineering

2.1.5.1 In-context learning

In-context learning refers to the ability of LLMs to accomplish tasks by exploiting contextual information inside the input prompt, minimizing the need for extra training or fine-tuning. This technique takes advantage of the model's considerable pre-trained information, allowing it to flexibly adapt to new tasks by comprehending the examples and guidelines included in the prompt. The model can operate in zero-shot, one-shot, or few-shot formats, receiving no, one, or a few instances to infer task requirements and create accurate responses. By executing instructions and examples in real-time, the model can fine-tune its responses to the requirements of each task. This allows it to efficiently handle various applications, demonstrating its high adaptability and ability to meet multiple needs flexibly and efficiently (Brown, 2020b).

2.1.5.2 Zero-shot learning

Zero-shot learning is considered a subset of In-Context Learning ICL. According to its purpose and approach, it may easily exceed trained transformers. Zero-shot learning is a kind of machine learning that occurs when an artificial intelligence (AI) algorithm is trained to identify and classify items or concepts without prior exposure to any examples of these groups or concepts. When there are no labeled instances of the classes the model is training to learn, zero-shot learning issues use additional data such as text explanations, characteristics, embedded depictions, or other semantic details related to the job.

Instead of simply representing the deciding boundary between categories, zero-shot machine learning methods often generate probabilistic vectors indicating the probability that the input in question belongs to one of several classes. Generalized zero-shot learning (GSZL) techniques may include an initial detector that identifies if the sample relates to an existing or new category and proceeds appropriately. The disadvantage of this model is that it is less accurate and reliable compared to models that have been fine-tuned due to the lack of relevant examples during the training phase.

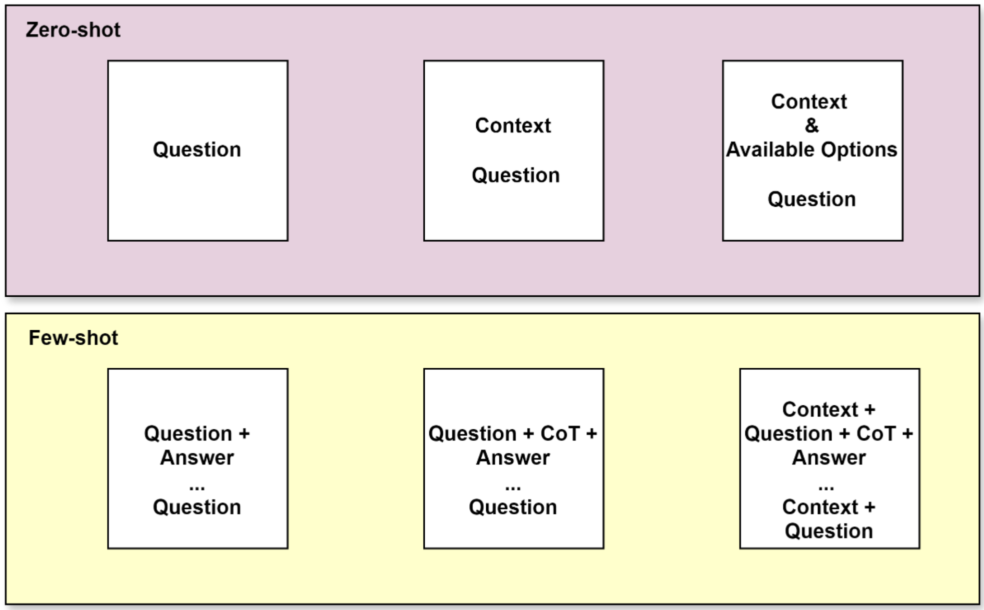

Figure 2.6: The difference between zero-shot and few-shot

2.1.5.3 Few-shot learning

A type of machine learning in which the model learns how to provide accurate predictions after training with a modest number of labeled instances. It is often utilized to develop models during tasks such as classification, where sufficient training data is limited. Since few-shot learning can use many algorithms or artificial neural network topologies, most techniques are centered around learning through transfer or meta-learning (or an amalgamation of both).

Few-shot learning is useful because it allows for developing machine learning techniques for real-world scenarios. In many situations, obtaining extensive data collection to train a model for machine learning might be difficult. Learning with a smaller dataset may substantially reduce the cost and work necessary for training machine learning algorithms. Few-shot learning has this potential since it allows algorithms to learn from small amounts of data (See Figure 6).

It can also help to create more adaptable machine-learning techniques. Conventional machine learning algorithms are often intended to work well on particular assignments and are trained on big data sets containing many labeled instances. This implies that the algorithms may not be able to generalize to new, unexplored data or execute effectively in tasks that are substantially distinct from those on which they have been trained.

Few-shot learning addresses this issue by allowing machine learning approaches to learn how to develop and adjust fast to new tasks using a small number of labeled instances. So, the models gain flexibility and adaptability (Brown, 2020a), Table 2.1 shows the Comparison of pre-training, fine-tuning, and context-learning for language models

Table 2.1: Comparison of pre-training, fine-tuning, and context-learning for language models

Aspect	Pre-training	Fine-tunning	In-context learning
Definition	Initial training phase using vast text data	Training a pre-trained model on a task-specific dataset	Adapting to new tasks or instructions provided within the input context
Objective	Learn general language patterns and structures	Specialize the model for specific tasks	Perform tasks based on the context given in the input
Techniques	- Masked Language Modeling (MLM) Next-Token Prediction	- Supervised learning with labeled data Task-specific training	- Providing examples or instructions in the input prompt
Advantages	- Develops broad language understanding Captures context, syntax, and semantics Foundation for various tasks	- Improves accuracy for specific tasks Requires less data and computation than pre-training Leverages pre-trained knowledge	- No additional training needed Rapid adaptation to tasks Generalizes from limited context and examples
Data requirements	Large, diverse text corpus	Smaller, task-specific labeled dataset	Minimal or no additional data beyond the input prompt
Computational Cost	High	Moderate	Low

2.2 Anomaly Detection

Also known as outlier detection, it is the discovery of observations, incidents, or data points that deviate from the usual, norm, or anticipated, rendering them incompatible with most data collections. Anomaly detection is frequently used in statistics, especially by researchers and scientists inspecting charts for unusual components. Nowadays, anomaly detection employs technology such as artificial intelligence (AI) and machine learning (ML) to detect odd changes in a dataset's typical behavior (Pang et al., 2021). Anomalous data can suggest severe issues under the hood, such as a network failure, an unanticipated change induced by an outside source, or security vulnerabilities. Abnormalities can also mean opportunities for structural reform or better advertising tactics.

Anomaly detection is a critical tool for maintaining business activities in a variety of industries. The usage of supervised, unsupervised, and semi-supervised learning algorithms will be determined by the type of data collected and the operational difficulty being addressed. Anomaly detection application cases include supervised learning (retail, weather forecasting), unsupervised learning (intrusion detection system, manufacturing), and semi-supervised learning (medical, fraud detection).

Detecting outliers is difficult because unusual anomalies and typical patterns are complex. Anomalies greatly affect systems and can lead to inaccurate results. Large language models can detect anomalies with great efficiency due to the information acquired during the pre-training process. They reduce the need for experts by providing powerful analytical capabilities, which help to detect complex anomalies.(Jin et al., 2024).

2.2.1 Software Vulnerabilities Detection

Software vulnerabilities are defects in software systems that criminals use to gain illicit access and inflict harm. Coding flaws, insufficient input validation, insecure setups, or the frequent usage of out-of-date libraries and dependencies cause them. These vulnerabilities can cause modest disruptions to significant security breaches, potentially resulting in data loss, financial damage, and reputational damage.

From January to mid-July 2022 to 2024, reported CVEs have consistently increased yearly. The figures increased from 14,249 in 2022 to 17,114 in 2023, then 22,254 in 2024. These figures correspond to annual growth rates of 24%, 20%, and 30%, respectively, underscoring a steady and substantial rise in the number of detected vulnerabilities. This trend suggests that the growing intricacy of software and the widespread usage of technology necessitates more sophisticated and adaptable vulnerability management solutions to combat the changing cybersecurity scenario.

Software vulnerabilities are defects or weaknesses in a system or program that attackers can use to risk security, integrity, or availability. Buffer overflow (CWE-120) occurs when data surpasses the boundary of a buffer and overwrites surrounding memory, potentially resulting in crashes or code execution. SQL injection (CWE-89) is another common vulnerability that includes applying noxious SQL code on input areas, allowing hackers to manipulate or access databases without authorization. Cross-site scripting (XSS) (CWE-79) enables hackers to insert malicious code into websites visited by other users, frequently exploited to steal cookies or session data. CWE-352: Cross-Site Request Forgery (CSRF)When a website's server is configured to accept an inquiry from a user with no way to verify that it was sent on purpose, an attacker may be able to fool a user into sending an inadvertent request to the website's server, which will be processed as a genuine request. This can be accomplished by a URL, XML HTTP request, image load, or any other method and can lead to data disclosure or inadvertent code execution (CWE - CWE-352(CSRF)).

Vulnerability identification is critical in protecting against such dangers because it identifies flaws and possible points of entry for criminals to exploit. Vulnerability detection methods use advanced scanning and penetration tests to thoroughly examine web applications and websites for weaknesses such as injection of SQL queries, cross-site- scripting (XSS), and improper verification schemes. Organizations can improve their online safety posture by actively detecting and fixing vulnerabilities, reducing the possibility of hacking, financial loss, and damage to reputation. In addition, identifying vulnerabilities enables firms to remain in compliance with industry norms and standards, proving their dedication to protecting important information and retaining client confidence. With the changing threat environment and more complicated means of attack, implementing strong vulnerability detection methods is critical for keeping ahead of cyber-attacks and guaranteeing the resilience of online platforms and applications (Cruz et al., 2023).

To comprehend which tools to employ in detecting vulnerabilities, we must first understand the differences between them. They can be categorized according to three tests for security groups: static analysis, dynamic analysis, and hybrid analysis.

2.2.1.1 Static Application Security Testing (SAST)

Static Application Security Testing has existed for over a decade. It lets programmers detect security flaws in apps' source code early in the software development procedure. It also guarantees compliance with coding standards and guidelines without running the fundamental code. By incorporating it into the initial phases of the SDLC (Software Development Life Cycle), SAST enables the early identification of security defects, which may substantially decrease the expenditures and work necessary to fix security concerns compared to discovering them after deployment.

One issue with SAST is the possibility of false alarms, which may consume time and money in researching and solving problems that are not present. Furthermore, because SAST focuses on static code examination rather than app behavior, it may overlook vulnerabilities that show only at execution or via interactions with other parts and systems (Shahrivar & Millar, 2024).

2.2.1.2 Dynamic Application Security Testing (DAST)

Dynamic application Security Testing can Identify security weaknesses and vulnerabilities in a functioning program, usually a web application. It accomplishes this by using fault injection strategies on a program, such as providing malicious information to the software, to detect typical security holes like SQL injection and crossed-site scripting (XSS). DAST may additionally highlight runtime concerns that static analysis cannot identify, such as authorization and server setup difficulties, as well as weaknesses that are only exposed when an authorized user signs in.

This technique has many other benefits. In the beginning, DAST examines the app from the outside, and it is an excellent solution for security professionals who do not have permission to see its base code or when conducting third-party service evaluations. Furthermore, DAST only evaluates an application's visible interfaces, mostly unaffected by the fundamental technologies utilized to develop the application. This renders DAST an adaptable instrument for security evaluations on a variety of platforms and topologies (Shahrivar & Millar, 2024).

2.2.1.3 Interactive Application Security Testing (IAST)

Interactive application security testing. Since SAST and DAST are prior technologies, some contend they need more security features for modern online and mobile applications. For example, SAST struggles to deal with various frameworks and libraries available in current apps. This is because static technologies only have access to the app's code base, which they may follow. Furthermore, libraries and third-party elements frequently cause static techniques to fail, resulting in "lost sources" and "lost sinks" warnings. The same thing goes with frameworks. Run a static analyzer on a public API, website service, or REST end-point, and it will detect no errors because it does not grasp the structure. IAST is intended to remedy the drawbacks of SAST and DAST by incorporating components from both systems. See Figure 7

IAST inserts an agent inside a program and does all of the monitoring in the program in real-time and at any point during the build process—IDE (Integrated Development Environment), continuous integration environment, quality assurance, or even operation (Sakaoglu, 2023).

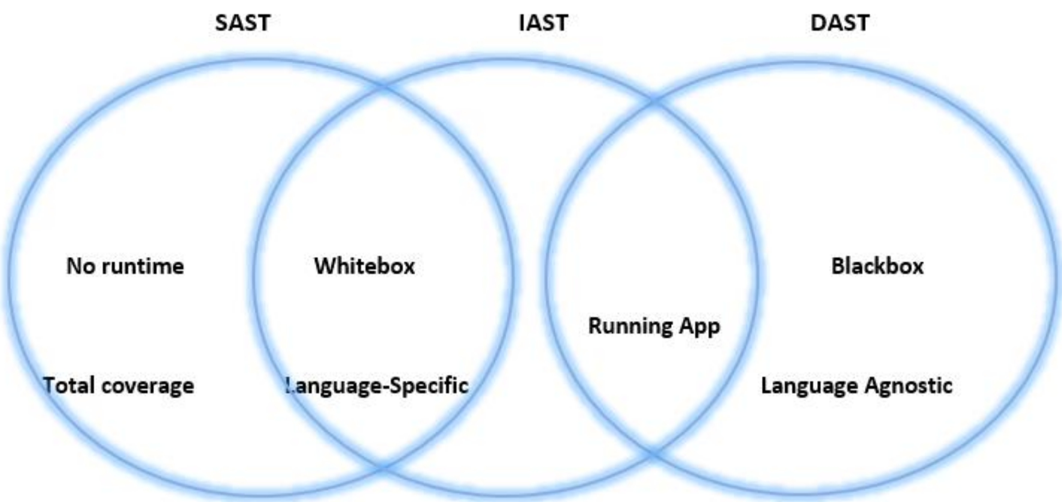

Figure 2.7: diagram of the SAST, DAST, and IAST methodologies

2.2.2 Intrusion Detection

Intrusion refers to unauthorized system access that harms or steals highly sensitive data. Due to the heavy reliance on digital environments, intrusion has threatened system security. Organizations constantly seek to develop their protection mechanisms against intrusion, so protection from intrusion and its detection have recently become common topics in cybersecurity.

In this chapter, we explore the role of intrusion in cybersecurity by explaining the methods of intrusion into systems, the mechanisms used to detect and repel intrusions, and the relationship between software vulnerabilities and intrusions into systems. This understanding helps us better protect systems and develop models capable of identifying vulnerabilities and protecting systems from intrusion.

In cybersecurity, the word "intrusion" may have many meanings. Still, it typically refers to illegal access efforts or activities that jeopardize the security, reliability, or accessibility of a computer system. The following are three prevalent kinds of cybersecurity intrusions:

Network intrusions are unapproved attempts to access a computer system. Criminals may attempt to obtain access to sensitive information, affect network services, or download viruses on linked devices. For Examples:

- Port scan: This is a typical method hackers use to uncover network vulnerabilities. A port scanning attack allows cyber attackers to identify open ports and determine whether they are acquiring or sending data. It may also indicate whether an organization uses active security equipment like firewalls.
- Denial-of-service (DoS) attacks are cyber assaults in which a hacker tries to make a device, such as a computer, inaccessible to its intended clients by disrupting its typical activity. DoS attacks typically operate by overloading or flooding a particular system with inquiries until regular traffic cannot be executed, resulting in a denial of service for other users.
- Man-in-the-middle (MitM) attacks: a broad phrase for when a criminal inserts himself into a user-application discussion, either to listen in or pretend to be one of the participants, giving the impression that a legitimate data exchange is taking place.

System intrusions: System intrusions are attempts to gain unapproved access to a particular system of computers. After gaining access, hackers might launch malware, steal data, or interfere with system functionality. Some examples of these attacks are:

- Password cracking involves identifying a lost or unknown password to a networked or personal computer using an app program. Additionally, it can assist an attacker in gaining unapproved access to systems.
- Exploiting software vulnerabilities: To obtain illicit entry to a system, criminals use common software vulnerabilities.
- Privilege escalation**:** The process by which attackers take advantage of flaws in a system to elevate their privileges, allowing them to gain access to resources or carry out actions that are not authorized.

Social Engineering Attacks: Instead of exploiting technological flaws, these attacks focus on human manipulation. Attackers use deception or trickery to get users to download malware, click on harmful links, or divulge private information. For Examples:

- Phishing attacks**:** Attackers impersonate respected firms in emails or chats to trick consumers into providing personal information.
- Pretexting**:** Attackers fabricate a situation to win over users' confidence and get private data**.**
- Baiting is a social engineering attack in which a fraudster falsely promises to entice the target into a trap that may steal private and financial data or infect the system with viruses. The enticement could be an illicit link with an appealing name.

It's crucial to remember that these groups might occasionally overlap. For example, a social engineering attack could acquire entry to a network or a system. Recognizing the various forms of invasion allows firms to take suitable safety precautions to reduce risks.

In cybersecurity, detection systems for intrusions are classified into two groups based on how they are used and data sources:

2.2.2.1 Network Intrusion Detection System (NIDS)

A security method that tracks and investigates traffic on the network to detect illicit behavior, illegal access, and policy breaches. A network intrusion detection system's principal duty is to identify and inform network administrators of any prospective or current attacks on the network.

NIDS scans data streams for specific patterns and characteristics that signal the existence of an intrusion. It may identify and notify network managers about attacks, including DoS, port scanning, malware, and unwanted access efforts. NIDS is an integral part of an overall network protection approach. See Figure 2.8.

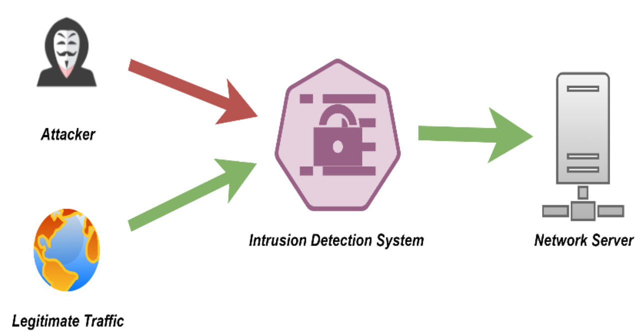

Figure 2.8: Intrusion Detection System

2.2.2.2 Host-Based Intrusion Detection System (HIDS)

It is a cybersecurity tool that scans IT systems for signals of illicit activity to identify unusual actions or behaviors linked with individuals or programs that might indicate an attack or attempted assault. HIDS platforms are named by the distinct host systems on which they work. In this sense, a host may refer to a server, a personal computer, or anything else that generates records, statistics, and additional data that can be tracked for security reasons.

(NIDS), (HIDS) are intended to identify attacks on networks and intrusions. They employ several detecting technologies to identify unusual traffic and anomalous activity. They use three basic detection techniques:

Signature-Based Detection

This approach checks traffic going via the network to identify signatures of attacks or behaviors. Attack signatures are predetermined traffic patterns linked to specific kinds of attacks. The NIDS notifies the network manager if the traffic fits an identified signature. Signature-based detection helps recognize identified attacks but cannot identify novel or unidentified ones.

Anomaly-Based Detection

This strategy entails identifying traffic that departs from typical network activity. A network intrusion detection system (NIDS) examines network activity and alerts when it detects behavior outside the normal range. Anomaly-based detection can help detect new or undiscovered assaults but can produce many incorrect results.

Hybrid Detection

This technique combines signature and anomaly detection techniques. The NIDS detects identified assaults first using signature-based detection, followed by unidentified assaults using anomaly-based detection. This method effectively detects both known and unknown attack types. Both detection solutions run concurrently, allowing one to give a means for filtering and grouping security alerts, reducing the number of alerts provided to the network administrator.

The three detection strategies (Signature-based, Anomaly-based, and Hybrid) have their advantages and disadvantages. The most appropriate technique is determined by criteria such as the network's particular safety requirements, available resources for IDS management, and a proper number of false positives.

2.3 Related Work

This section will comprehensively study the latest tools/frameworks used to predict software vulnerabilities.

2.3.1 Software Vulnerability Detection Tools Comparison

When discussing the tools used to detect software vulnerabilities, it is necessary to consider factors that help decision-making. This research (Das et al., 2021a) comprehensively reviews the tools used to detect vulnerabilities to determine their advantages and disadvantages. The review begins by looking at various software vulnerability detection techniques, such as scanning, fuzzing, and static analysis. Each approach is examined for a better understanding of its fundamental abilities and drawbacks in detecting vulnerabilities. The review illuminates the various methods used in vulnerability detection by comparing their strengths and weaknesses. This study (Kaur & Nayyar, 2020) Compare tools for analyzing static code, showing each has unique strengths and weaknesses. While Flawfinder and RATS detect the same vulnerabilities, RATS detects more vulnerabilities than Flawfinder, albeit with a more complicated installation procedure. However, CPPCheck grows as an incredibly effective tool, discovering the most vulnerabilities out of the three. Regarding Java static testing, Determine Bugs shows how it can identify vulnerabilities that PMD does not detect. The study emphasizes the need to assess and contrast tools for static code analysis for both the C++ /C and Java languages. Regardless of the techniques' efficiency, some vulnerabilities in the evaluated code go unnoticed. The authors express a future research goal of developing an additional tool to fill these shortcomings, aiming to improve vulnerability detection abilities beyond the limits of currently available open-source static analysis tools.

The paper (Shen & Chen, 2020) discusses the growing significance of solving software security problems, especially in open-source code, although widely used for its transparency and adaptability, also presents potential security vulnerabilities. With the increasing size and complexity of software platforms, common manual techniques for dealing with these problems have become inefficient. As a result, enhanced technologies, including deep learning, are critical for automating software security processes. The advancement of technology for deep learning has created new possibilities for addressing software security issues, prompting the development of various automation techniques. These techniques are assessed for identifying software vulnerabilities, repairing programs, and predicting defects. The authors believe that, despite progress, there are flaws in current research techniques, which are discussed with potential solutions. Looking ahead, they believe there is hope for possibilities for using deep learning technology in automated software safety projects, highlighting its growing importance in this area. The paper highlights the necessity of integrating deep learning and software analysis methods to enhance automated detection techniques, ultimately improving software quality and safety. Another study (Guo et al., 2022b) provides a hybrid approach that combines a Bidirectional Long Short-Term Memory (BiLSTM) model and a code property graph; the classifier analyzes the model's output to assess whether the sample of code is vulnerable. This approach's main drawback is that it necessitates several steps of preparation to bring data into the algorithm. The Joern tool—compatible with other programming languages—creates the code property graph. Word2vec, a pre-trained model, is used by the Bidirectional LSTM model. Despite being a white-box detection technique, it cannot identify the exact location of the vulnerability in the provided code snippet. Regarding detection accuracy, it outperforms the current rule-based vulnerability mining tools, such as Cppcheck and Flawfinder. Also, the paper (Medeiros et al., 2016) proposed an AI structure that deals with code like natural language. Common static analysis methods usually parse or extract code into an abstract structure for data. DEKANT, however, applies the NLP approach hidden Markov framework to the code. This approach is trained using an annotated large code corpus in PHP. To enhance the model's detection abilities, the authors first split the large code sections into slices like code snippets in HyVulnDetect (Guo et al., 2022b). The snippets are then converted into an intermediate language, which removes unnecessary lines of code and simplifies the rest with taint analysis vocabulary. As a result, the model requires fewer tokens for classification and can be used across multiple platforms as long as converters for its specific intermediate language are available. DEKANT discovered 4143 zero-day vulnerabilities throughout multiple open-source PHP plugins with great accuracy.

2.3.2 State-of-the-art Vulnerability Detection Tools

The emergence of LLMs has brought about a qualitative shift, especially in cybersecurity. LLMs have shown promising results in predicting software vulnerabilities through their ability to understand and analyze code due to their superior natural language processing capabilities. The paper (Thapa et al., 2022) shows that large transformer-based language models have the potential for natural language processing. This study demonstrates the effectiveness of large transformer-based language models for identifying software vulnerabilities, considering their cross-domain portability and similarity to programming languages such as C/C++. It presents a systematic approach to source code translation, model building, and inference. The qualitative examination uses software attack datasets containing C/C++ source code with vulnerabilities related to library function calls, use of pointers, use of arrays, and arithmetic expressions; experimental results show that the language models are capable of detecting vulnerabilities. These language models outperform state-of-the-art models such as LSTM and BGRU. Experimenting with language models is complex as it requires computer resources, systems, libraries, and interactions. The research analyzes major platforms to optimize models and provide advice for selecting platforms.

This study (Ferrag et al., 2023) Shows that when it comes to cyber threat identification, a classification model enhanced by LLMs performs better than well-known conventional Machine Learning (ML) and Deep Learning (DL) techniques like Convolutional Neural Networks (CNN) or Recurrent Neural Networks (RNN). Security LLM identifies fourteen distinct types of attacks with an astounding 98% accuracy during testing on a cybersecurity dataset. The paper (Omar & Shiaeles, 2023) proposed VulDetect, a novel transformer-based vulnerability detection system that leverages a pre-trained language model to correctly identify insecure code patterns with up to 92.65% accuracy, exceeding previous state-of-the-art methodologies. Compared to several modern vulnerability detection approaches, SyseVR and VulDeBERT 's proposed approach outperforms them.

The study (C. Wang et al., 2023) presents a novel approach to automatically collect detailed information about real-world Common Vulnerabilities and Exposures (CVEs). Utilizing the power of extensive language models produces more illustrative messages for these compiled vulnerabilities. This framework accumulated 4,466 CVEs from 2016 to 2023, and 30,987 messages are associated with the accrued patches. The excellent quality of these generated signals is confirmed by developer evaluations, which highlight how helpful they are in helping developers understand vulnerabilities. The article presents itself as a guide for researchers, including recommendations for enhancing data-driven bug discovery and automated patching methods in the area**.** The study (Chen et al., 2023) Provides DiverseVul, a new dataset for deep learning-based vulnerability detection with 18,945 vulnerable functions and 330,492 nonvulnerable functions from 7,514 contributions, outnumbering CVEFixes in size and diversity.

It compares 11 designs from the "GNN, RoBERTa, GPT-2, and T5" families, discovering that higher data variety improves vulnerability identification, particularly for larger models. The research (Das et al., 2021b) introduces V2W-BERT, a unique Transformer-based system that combines natural language processing, link prediction, and transfer learning. It outperforms previous techniques in addressing common and rare CWE situations, demonstrating substantial advances in using historical data for future CVE forecasting. They achieved up to 97% accuracy in random data partitioning and 94% accuracy in temporal data dividing**.** The research (Seyyar et al., 2022) proposes a novel model using BERT and deep learning to distinguish between normal and anomalous HTTP requests. The study achieves excellent success rates by using BERT for URL representations with an MLP classification, achieving 99.98% accuracy and a 98.70% F1 score in classification. Notably, the model has a speedy web assault detection time of 0.4 ms, beating existing literature approaches significantly. Previous studies were not limited to predicting software weaknesses, but some studies used LLMs to measure their accuracy in fixing software weaknesses, the paper (Pearce et al., 2023) examines how LLMs can repair software vulnerabilities in a zero-shot situation. It discovered that black-box LLMs can fix security flaws when given a specific prompt, as seen in all artificial and handmade scenarios. However, despite being considered a cutting-edge approach, its efficiency evaluation indicates that it falls short of delivering true value in a program repair framework.

This study (Noever, 2023) Assesses the efficiency of (LLMs), with a particular emphasis on OpenAI's GPT-4, to detect vulnerabilities in software. A comparison is made with traditional static code detectors that include Snyk and Fortify. The study discovered that GPT-4 determines around four times more vulnerabilities than the others, based on repositories from prestigious sources such as NASA and the Department of Defense. Notably, it offers possible fixes for each vulnerability while producing an extremely low rate of false positives. In 129 code samples from eight programming languages, PHP and JavaScript have the most significant vulnerability rates. GPT-4's code corrections result in a substantial 90% decrease in vulnerabilities while adding only 11% more code lines; an important observation is the LLMs' ability to self-audit, recommending fixes for discovered vulnerabilities and emphasizing their accuracy in detecting vulnerabilities and repair. The author suggests that future research focus on exploring system-level vulnerabilities and integrating several static code analysis tools to gain a comprehensive perspective on the capabilities of LLM.

DeepCode AI Fix (Berabi et al., 2024) was proposed as a new learning-based system for self-correcting non-trivial errors in code and vulnerabilities. DeepCode AI Fix accurately learns to fix coding errors by incorporating program analysis into the machine learning pipeline, concentrating on critical code information extracted by CodeReduce rather than uncovering long-range dependencies. The authors solve the data collection challenge by creating an excellent data set from millions of GitHub commits. DeepCode AI Fix outperforms previous modern techniques such as TFix, leveraging program analysis to simplify attention learning and enhance performance across various setups.

2.3.3 Intrusion Detection Tools Comparison

Many techniques were used to anticipate and prevent intrusion, and the accuracy of the results differed according to the data set used in the detection process, the research (Bhatti & Virparia, 2020) proposed a model for intrusion detection based on the Support Vector Machine, and 99.92 % accuracy was obtained on the dataset NSL-KDD. However, the study does not specify the size of the training and testing datasets, nor the validation strategy used, which limits reproducibility. Also, the Support Vector Machine's performance decreases when ample data is included, especially for large data traffic, such as when analyzing an intrusion detection network.

The paper (H. Wang et al., 2017) proposed a hyper-model for the detection process DLHA that combines two classifiers, the naive base and support vector machine. This model achieved higher results than applying one classifier by getting detection rates of 96.67% and 100% to R2L and U2R, respectively. DLHA exhibits exemplary performance in identifying uncommon attacks.

The authors (Wisanwanichthan & Thammawichai, 2021) employed a hybrid SVM-KPCA-GA model to detect intrusions, and the system had a 96% detection rate. They used the KDD CUP99 dataset When they tested their system. However, there are limitations to this dataset. For example, redundancy leads to bias in the classifier toward records that occur more frequently. They used KPCA to decrease the number of features, but it has limitations due to the possibility of leaving out crucial features because the top percentages of the primary component are chosen from the principal space. The SVM is also unsuitable for handling large amounts of data, such as observing the network's high bandwidth. The research (Kuang et al., 2014)created an RF-based framework for an IDS. They evaluated the performance of their model on an NSL-KDD dataset, and the results showed a 99.67% detection rate; the RF algorithm's main problem is that it may be too slow for real-time prediction due to multiple trees.

The authors (Ravale et al., 2015) proposed a hybrid approach that combines K-means with SVM. They generated the experimental outcomes using the KDD Cup 99 dataset. As mentioned earlier, this data suffers from limitations such as redundancy.In (Nisioti et al., 2018) The paper highlights the importance of feature engineering in security detection and provides an in-depth review of unsupervised and hybrid approaches to detecting intrusion. Another paper (Mohammadi et al., 2019) proposed a model for IDs based on feature selection and decision tree with an accuracy of 95.3%.

A novel study (Awajan, 2023)suggested using deep learning techniques to create an Intrusion Detection System (IDS) aimed at improving the security of IoT networks. It consists of a four-layer fully connected neural network, and its modular structure allows it to be deployed in different IoT settings without being constrained by communication protocols. The model was trained on data from a testbed IoT network which allowed it to adapt to the specific features of the network and solve the heterogeneity problem typical in IoT systems. It proved to be very effective for the detection of Blackhole, DDoS, Sinkhole, and other attacks with an average accuracy of 93.74%% (precision, recall, F1 score, all above 93%). The study also outlines plans to create a trimmed-down version of the system intended for IoT devices with limited resources.

A deep learning-based Network Intrusion Detection System (NIDS) (Hnamte & Hussain, 2023) was developed to try and solve the increasingly difficult challenge of detecting sophisticated cyberattacks in constantly changing network environments. This particular study applied deep learning because it provided great results in detection and classification and the problems in question required such methods. The proposed model was developed and tested on two live traffic data streams, CICIDS2018 and Edge_IIoT, which were benchmarked in the context of multitask classification. The model accomplished remarkable accuracy values of 100% and 99.64%, respectively, which provided evidence that it can reliably distinguish various types of network intrusions. This advancement demonstrates that there is much work to be done leveraging deep learning to improve performance in intrusion detection systems aligned with current and evolving network systems.

2.4 Summary

Based on our literature review, we have found that traditional methods for detecting software vulnerabilities are insufficient for identifying complex vulnerabilities. Techniques such as (SAST), (DAST), and (IAST) have various limitations, including high false alarm rates and low accurate positive rates. Although previous studies have made progress using hybrid and composite models, they still struggle to identify vulnerabilities that require a broad contextual understanding and insight into system interactions. In contrast, in our research, we employed large language models to detect software vulnerabilities and analyze source code effectively.

Given that our research focuses on the DiversVul dataset (text data consisting of functions), Large language models (LLMs) possess the ability to analyze programming code by understanding syntax, control flow, and context, enabling them to identify potential vulnerabilities and thereby enhance software security. Previous studies have not fully explored the comparative performance of large language models such as GPT, Llama, and BERT in predicting software vulnerabilities. Many have either concentrated on a single model or conducted only limited comparisons. In addition to software vulnerability detection, several previous studies have also explored intrusion detection systems (IDS) using machine learning techniques. This study aimed to classify network traffic and identify abnormal behavior, with the NSL-KDD dataset being one of the most commonly used benchmarks in this domain.

This research aims to bridge this gap by comprehensively comparing large language models to evaluate their effectiveness in predicting software vulnerabilities. We focus on leveraging the capabilities of large language models to create a complete framework for software vulnerability detection. We also develop a system that uses machine learning algorithms to identify vulnerabilities using the NSL-KDD dataset.

Chapter Three: Methodology

This section provides an overview of the research methodology used to achieve this study's desired objectives using ML algorithms and LLMs. The methodology is divided into two parts: the first focuses on using large language models to predict software vulnerabilities, and the second uses machine learning algorithms to predict network intrusions.

3.1 General Framework

This section follows the standard research procedures of exploration, experimentation, and evaluation. The research began with an extensive review of the continuously updated literature to understand the topic and then identify appropriate models to use and select datasets. After the literature review, the scope of the experiment and technique were created to address the identified shortcomings. The experiment was designed according to the selected models and datasets. The best-performing models are evaluated and compared using the evaluation method described in Section 3.8. In the post-analysis phase, we discuss the experiment's results, share our experiences and lessons learned, and plan future research goals. Figure 3.1 illustrates the complete framework.

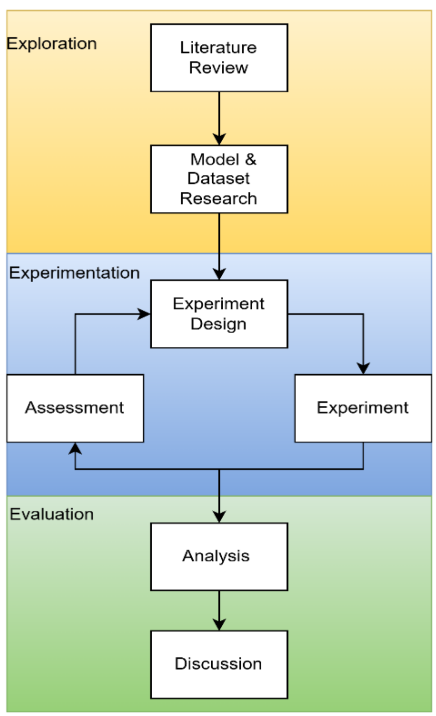

Figure 3.1: General Framework

3.2 Proposed Methodology

This section provides an overview of the research method used in each part. First, we present the experimental design flowchart, followed by the data section, then a detailed explanation of the techniques used to evaluate the accuracy of the data used, then an explanation of the methods used in data analysis, and finally, the evaluation to assess the proposed method.

This section is divided into two main sub-sections:

3.2.1 Software Vulnerability Methods

This subsection will discuss the methods used to predict software vulnerabilities and provide an overview of each LLMs and ML algorithm. Figure 10 demonstrates the flowchart for experimental design in this subsection.

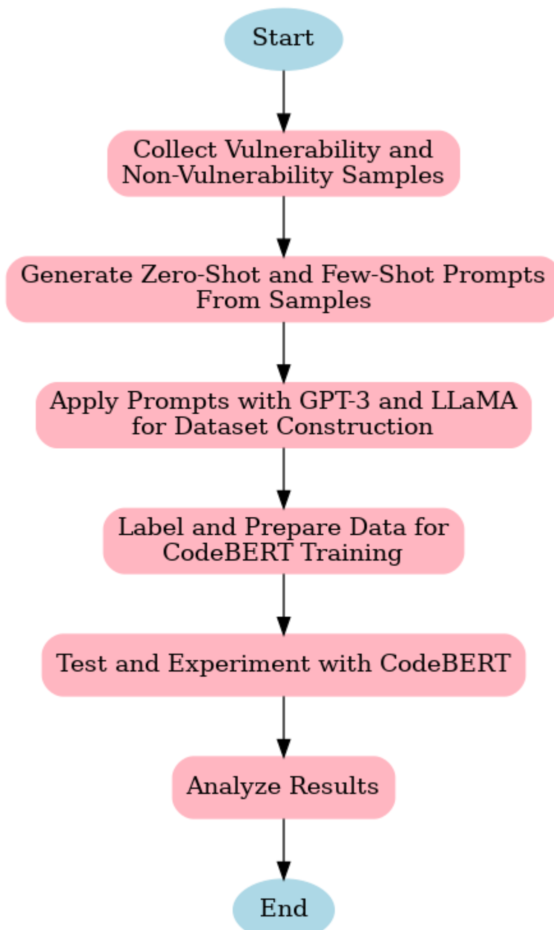

Figure 3.2: Experimental design in software vulnerability detection

The experiment's structure is divided into three stages: preparing the data, testing the model, and analyzing. The first stage involves preparing data for training and testing, then evaluating the models on the prepared dataset and comparing and analyzing their performance.

This experiment aims to measure the performance of LLMs (GPT, Llama, and codeBERT) in predicting software vulnerabilities on test data of three sizes (1000, 5000, and 20,000 records). The performance of GPT and Llama models was measured in zero-shot and few-shot cases, while codeBERT was finetuned on 1,000 and 5,000 records, and the performance of CodeBERT-1000 and CodeBERT-5000 models was tested on test data. Fine-tuning on the 20,000-record dataset was not performed due to the high computational resources required, the available hardware limitations made fine-tuning such a large dataset impractical.

The performance of the models was measured using key metrics such as accuracy, precision, recall, and F1-Score to analyze the effect of training and testing data sizes on the three models' performance and determine which is the best at classifying software vulnerabilities according to the specified metrics. The steps are as follows:

Step 1: Collect vulnerability and non-vulnerability samples

Step 2: Divide the dataset into three test set sizes (1,000, 5,000, and 20,000 records).

Step 3: Generate zero-shot and few-shot prompts from samples

Step 4: Apply the prompts to GPT and Llama for the test dataset

Step 5: Collecting the results from the GPT and Llama models

Step 6: Prepare training datasets and train two models using CodeBERT (One was trained on 1000 records and the second on 5000 records)

Step 7: Testing and experimenting with trained CodeBERT models into the three test set sizes (1,000, 5,000, and 20,000 records)

Step 8: Collecting the results from trained CodeBERT models

Step 9: Analysis Results

3.2.2 Intrusion Detection Methods

This subsection focuses on the methods used in network intrusion prediction, providing an overview of each.

The experiment's structure is divided into three stages: preparing the data, testing the model, and analyzing. The first stage involves preparing the NSL-KDD dataset for training and testing, then evaluating the machine learning models on the prepared dataset and comparing and analyzing their performance. Figure 3.3 demonstrates the flowchart for experimental design in this sub-section.

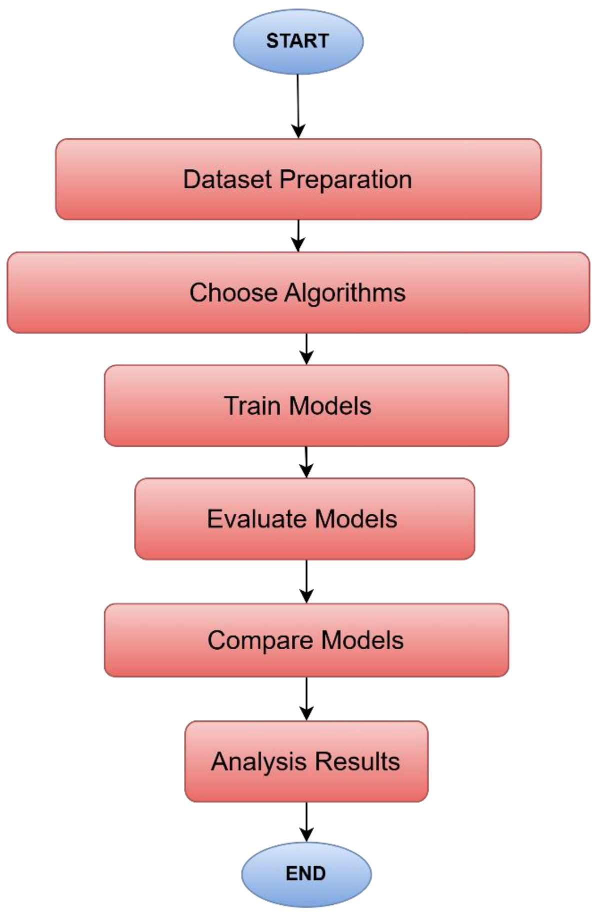

Figure 3.3: Experimental design in network intrusion detection

This experiment aims to measure the ability of ML models to predict network intrusion using the NSL-KDD dataset and evaluate the performance of the models using key metrics such as accuracy, precision, recall, and F1-Score. The steps are as follows:

Step 1: Dataset Preparation

Select the dataset (NSL-KDD), split it into 85% training and 15% testing, and preprocess the data.

Step 2: Choose Algorithms.

Select machine learning models for evaluation: Logistic Regression (LR), Random Forest (RF), Support Vector Machine (SVM, and Decision Tree (DT), These algorithms were chosen because they are effective for structured data classification, offering strong performance and interpretability

Step 3: Train Models

Train each model on the training data with hyperparameter optimization to improve the performance of the models.

Step 4: Evaluate Models

Testing the performance of trained models on testing datasets and calculating performance metrics (Accuracy, Precision, Recall, F1-Score).

Step 5: Compare Models

Compare the performance of models and determine the best model according to performance metrics.

Step 6: Analysis Results

Analyze and discuss which model achieved the best performance.

3.3 Datasets Description

This section describes the dataset utilized in this study, including the "DiverseVul" dataset for predicting software vulnerabilities and the "NSL-KDD" dataset for predicting network intrusions. Consequently, this section is organized into two sub-sections:

3.3.1 Software Vulnerability Dataset

The DiverseVul dataset, an expansive dataset for deep learning-based vulnerability identification, contains 18,945 vulnerable functions spanning 150 Common Weakness

Enumerations (CWEs) gathered from security problem websites and GitHub commits. It also includes 330,492 non-vulnerable functions derived from 7,514 commits.

The research (Chen et al., 2023) obtains this dataset by systematically collecting data from security issue websites, so its real-world functions from open-source applications Specifically, they extract Commits that fix vulnerabilities and the related source code from various software projects. Notably, this dataset encompasses data from 295 more projects than all previous datasets combined also, DiverseVul (Wagner-Group/DiverseVul) is limited to two programming languages, which are C and C++. This data contains eight features (Function /Target /Cwe (Common Weakness Enumeration) /Project /commit_id /hash/ size /message).

The following discussion highlights how these features are capable of helping detect vulnerabilities:

- Function '**func**': This feature denotes the code snippet. In vulnerability detection, this describes the section of code that could contain the vulnerability. This section will focus on vulnerability detection techniques to analyze and discover possible vulnerabilities.

- Target '**target**': shows if the code snippet is vulnerable (1) or not (0).

- CWE (Common Weakness Enumeration)'**cwe**': CWE is a community-generated list of standard software and hardware flaws that could be present in code. It's used to identify software flaws and vulnerabilities. The CWE field identifies the category of the vulnerability. It provides details about the type of issue found within the code. For instance, CWE-119 refers to a buffer overflow, while CWE-79 indicates a cross-site scripting (XSS) vulnerability.

- Project "**project**": This field specifies the project's name associated with the commit. It helps clarify the context in which the vulnerability occurs.
- Commit ID "**commit_id**": This is the unique identifier for the commit. Each commit in a version control system has its specific ID, which is vital for tracking changes made to the codebase. Keeping track of this ID ensures the discovered vulnerabilities can be traced back to the code modification.

- Hash "**hash**": This field represents the unique identifier of the code snippet within the commit. It can be used to trace changes in the code and is essential for verifying its integrity and monitoring alterations that might introduce vulnerabilities.
- Size "**size**": This indicates the length of the code snippet. Understanding the complexity and size of the vulnerable code helps assess its nature.
- Message "**message**": The commit message summarizes or describes the changes made. It often includes helpful information that might offer additional benefits.

Here is an example of one of the functions and its features; see Table 3.1.

Table 3.1.: an example of one of the functions and its features


```
{"func": "static char *make_filename_safe(const char *filename
TSRMLS_DC)\n{\n\tif (*filename && strncmp(filename, \":memory:\",
sizeof(\":memory:\")-1)) {\n\t\tchar *fullpath =
expand_filepath(filename, NULL TSRMLS_CC);\n\n\t\tif (!fullpath)
{\n\t\t\treturn NULL;\n\t\t}\n\n\t\tif (PG(safe_mode) &&
(!php_checkuid(fullpath, NULL,
CHECKUID_CHECK_FILE_AND_DIR)))
{\n\t\t\tefree(fullpath);\n\t\t\treturn NULL;\n\t\t}\n\n\t\tif
(php_check_open_basedir(fullpath TSRMLS_CC))
{\n\t\t\tefree(fullpath);\n\t\t\treturn NULL;\n\t\t}\n\t\treturn
fullpath;\n\t}\n\treturn estrdup(filename);\n}",
"target": 1,
"cwe": ["CWE-264"],
"project": "php-src",
"commit_id": "055ecbc62878e86287d742c7246c21606cee8183",
"hash": 211824207069112513181516095447837228041,
"size": 22,
"message": "Improve check for: memory: pseudo-filename in SQlite"}
```

3.3.1.1 Sampling

In this section, the sampling process divides the dataset into smaller datasets to test the model performance. Samples were taken for 1000, 5000, and 20000 records. to balance evaluation efficiency and computational cost, randomly selected subsets were used instead of the full dataset. This approach allows for controlled performance assessment while maintaining result generalizability.

They were used to test the efficiency of models concerning datasets of different sizes. Using larger subsets respectively, the aim is to capture the effect of data size on the performance of large language models such as GPT, Llama, and BERT.

After dividing the datasets, the performance of each dataset was tested in terms of accuracy, precision, recall, and F1 score. This approach allowed for an analysis of the model's ability to perform under different dataset conditions, providing insight into their adaptability and effectiveness in handling datasets of different sizes. The datasets and their characteristics can be seen in Table 3.2.

Table 3.2: Testing Datasets and their Attributes

Name	Total	Vulnerability	Not Vulnerability
Test set 1	20,000	10417	9583
Test set 2	5,000	2500	2500
Test set 3	1,000	500	500

3.3.2 Intrusion Dataset

The efficiency of the entire system is entirely dependent on the dataset, the more representative and relevant the data, the more effective the system will be.

Therefore, collecting and selecting data is one of the most significant phases influencing the system's efficiency. To ensure the highest level of effectiveness, we used the NSL-KDD dataset from the Kaggle database (NSL-KDD). The NSL-KDD dataset is a standard benchmark for intrusion detection and network security research, an advanced version of the KDD Cup 99 dataset that aims to address problems in the original data, such as data duplication. The dataset involves details about network activity from both normal and abnormal activities, and abnormal activity has four types (Dos, Probe, U2R, R2L). The dataset's features extracted from network data packets include service types, flag information, protocol types, and connection statistics. The dataset was split into two training (125,973) and testing (22,544). The main goal of the NSL-KDD dataset is to offer an accurate and representative environment for testing intrusion detection systems; see Table 3.3.

Table 3.3: Testing and Training NSL-KDD Datasets and their Attributes

Dataset	KDD Train	KDD Test
Total	125973	22544
Normal	67343 (53%)	9711 (43%)
Dos	45927 (37%)	7458 (33%)
Probe	11656 (9.11%)	2421 (11%)
U2r	52 (0.04%)	200 (0.9%)
R2l	995 (0.85%)	2654 (12.1%)

3.3.2.1 Sampling

In this subsection, we relied on classifying the data by dividing it into two main categories, normal and abnormal, without going into details about the types of attacks to facilitate detecting intrusion in general without the need to know the kind of attack accurately. Accordingly, the records representing attacks of any type (Dos, Probe, U2R, R2L) were classified into the abnormal category, and regular activities were classified into normal to improve the efficiency of the system in terms of accuracy and speed and reduce the need for complex models to classify the type of attack. This broader classification was intentionally adopted to align with the study's focus on general intrusion detection rather than attack-type identification, See Table 3.4.

Table 3.4: Summary of the Dataset Distribution (1)

Dataset	Total	Normal	Abnormal
KDDTrain	125973	67343 (53%)	58630 (47%)
KDDTest	22544	9711 (43%)	12733 (57%)

The target population in this sub-section consists of network traffic records from the NSL-KDD dataset, which are classified into two categories: normal and abnormal. In this context, regular traffic is labeled 0 (not vulnerable) and abnormal traffic 1 (vulnerable). By focusing on this specific category of network traffic patterns, the goal is to develop machine learning algorithms that can efficiently and accurately predict network intrusions.

3.4 Models

This section covers the models utilized in this study, including large language models for predicting software vulnerabilities and machine learning models for forecasting network intrusions:

3.4.1 LLMs Description

3.4.1.1 GPT

Designed by OpenAI, Generative Pre-trained Transformers, or GPT, are neural network models that use the transformer structure mentioned in section 2.1.2. They significantly improve artificial intelligence (AI), providing generative AI applications like ChatGPT. GPT models enable applications to generate human-like language and content and conversationally answer questions. These models are pre-trained on massive amounts of data containing different texts across all domains. This makes it a practical choice for tasks requiring natural language processing, as it can handle functions with or without training data by exploiting the previously acquired knowledge(Roumeliotis & Tselikas, 2023). GPT includes several Transformer Layers with decoder-only structure and uses the Self-Attention Mechanism, enabling it to learn long-range dependencies between words (see section 2.1.1.3)

$$Attention(Q, K, V) = softmax\left(\frac{QK^{\top}}{\sqrt{\{d_k\}}}\right)V$$

Where: Q(Queries) for the matrix of queries with dimension d_k. K (Keys) The matrix of keys also of dimension d_k. V(Values) The matrix of values where each is of dimension d_v. d_k The dimensionality of the query and key vectors serve to scale the dot product to avoid overly large values. d_v The dimensionality of value vectors.

It is characterized by many parameters, amounting to hundreds of billions, considerably enhancing its capability to understand and produce cohesive and contextually correct text. In the pre-training stage, GPT uses unsupervised learning, being taught to predict the next word in a sentence from the context (OpenAI)

Even while it can generate language that appears coherent, (Asmiddin et al., 2023)ChatGPT has difficulties recognizing more complex situations; thus, its answers are not always exact. It depends solely on data from training and has no actual knowledge. If not directed correctly, this may result in inaccurate replies. Additionally, some inquiries are not addressed with sufficient contextual awareness, requiring human input to ensure clarity. As a result, the reliability and accuracy of its responses cannot be fully guaranteed, and the output may be overly generic, failing to provide precise answers to specific queries.

This study used GPT-3.5 to classify software vulnerabilities in the zero-shot and few-shot cases. Performance was measured using the primary metrics of accuracy, precision, recall, and F1 score on three different data sizes (1000, 5000, and 20000 records) to evaluate the model's effectiveness in detecting software vulnerabilities.

We also undertook a fine-tuning step to a GPT model pretrained to align with a specific dataset concerning software vulnerabilities. Two datasets of different sizes were used to assess the impact of the dataset scale on the model's performance. The first dataset contained 1,000 records, while the second included 5,000 records.

Following OpenAI's fine-tuning procedure, all training data was organized in the JSONL format. Each record consisted of a prompt-response interaction that showcased the target domain.

Using the OpenAI API, fine-tuning was performed with hyperparameters set by the server based on the dataset's size and task type. Both datasets were trained for three epochs. Additionally, the batch size and learning rate multiplier were modified to tune the most optimal parameters. Table 3.5 below captures the more relevant training parameters.

Table 3.5: Summary of the Dataset Distribution (2)

Parameter	GPT-1000	GPT-5000
Epochs	3	3
Batch Size	1	9
Learning Rate Multiplier	2	2
Trained Tokens	~2,032,917	~10,270,128

The fine-tuning process resulted in two GPT models: one based on the 1,000-record dataset (GPT-1000) and the second on the 5,000-record dataset (GPT-5000). Both models were tested on three different-sized test datasets 1,000, 5,000, and 20,000 records, to assess their performance and generalization more comprehensively. The evaluation used uniform measures of performance which included accuracy, precision, recall, and F1-score.

3.2.1.2 Llama

A series of advanced models of languages. Famous as Llama (Large Language Model Meta AI), Designed by Meta AI (formerly Facebook), it seeks to understand and produce human-like texts, making them valuable tools for processing natural languages, text production, and chatting AI. Llama is intended to be adaptable, which means it can be trained and fine-tuned on a wide range of hardware configurations within powerful GPUs to readily available computing platforms. This makes it easier for scientists and programmers with limited resources, expanding the availability of advanced models of languages. Llama has produced outstanding results across various NLP benchmarks, including text categorization(Edwards & Camacho-Collados, 2024) and machine translation(H. Xu et al., 2023). This performance demonstrates the complex architecture and training approaches Meta's AI researchers use.

Llama is based on the Transformer architecture, which has been used by most recent large language models involving GPT and BERT. Transformers with decoder-only structures use self-attention processes to process input sequences simultaneously, resulting in rapid training and superior language modeling (as mentioned in sections 2.1.1.3 and 2.1.2).

Llama models have numerous layers of Transformer modules. Each module consists of a multi-head self-attention mechanism, then a network of feedforward neurons. The number of layers depends on the individual Llama version; it is available in a variety of sizes, each one having a unique set of parameters: Llama-7B contains 7 billion parameters, Llama-13B contains 13 billion parameters, Llama-30B contains 30 billion parameters, Llama-65B includes 65 billion parameters. This study used (Llama-2-70b) to classify software vulnerabilities in the zero-shot and few-shot cases. On three different data sizes (1000, 5000, and 20000 records), the performance was measured using the primary metrics of accuracy, precision, recall, and F1 score to evaluate the model's effectiveness in detecting software vulnerabilities.

This experiment aims to evaluate the performance of both GPT and Llama models in zero-shot and few-shot cases to measure their ability to analyze software code and discover software vulnerabilities. In the case of zero-shot, the performance of the models was measured without training them on any examples, "as a software engineer, I need a short answer (yes, no) only, if there a software vulnerability in code snippet provided in user message" The question was formulated precisely and clearly to ensure the model responded directly. Starting with the preparation of the prompt, the models were asked to adopt the role of the technical expert through the phrase "As a Software Engineer." to provide accurate answers and not just respond as a general model; the purpose of using the words is to motivate the model to use its technical knowledge and analyze the programming code accurately to answer correctly and directly. (Prompt Engineering - OpenAI API, n.d.). Also, the answer was restricted to Yes or No by saying, "I need a short answer (yes, no) only." In the case of a few shots, five examples were provided to guide the models in improving their performance. The experiment will be conducted on three different data sets consisting of 1,000 records, 5,000 records, and 20,000 records. Then, the ability of the models in each case will be determined by measuring the accuracy of the two models

To measure the ability of large language models GPT and Llama to predict and analyze software vulnerabilities, we will use a dataset of 50 records to measure the performance of the models in the zero-shot case in analyzing software vulnerabilities and providing an explanation for their answers. This size was selected to assess the models' performance on small amounts of data, which consistently determines how well they can manage software vulnerability analysis with little input. We chose this size due to resource constraints because it was adequate to offer initial insights.

This experiment aims to know the logic that these models have in answering and measure their ability to explain security vulnerabilities and their types and to know their criteria for answering (B. Wang et al., 2023)The question for the models is: "As a software engineer, I need a short answer with an explanation if there is a software vulnerability in the code snippet provided in the user message."

To verify the consistency and reliability of the model's performance, we will experiment using GPT and Llama in a zero-shot scenario with a dataset of 1,000 records. We will run the code on the same dataset five times without saving memory between runs to ensure that the accuracy of the results remains stable across repetitions. The aim is to assess the reliability of these models and confirm that the results are consistent and do not conflict with one another.

3.2.1.3 BERT

Bidirectional Encoder Representations from Transformers, designed by Google AI, employs a transformer-based neural network structure to grasp and generate like human language. BERT has an encoder-only architecture. The original Transformer structure had two decoder and encoder modules. The decision to use an encoder-only design in BERT indicates that the primary purpose is to evaluate sequences entered rather than generate output sequences.

Conventional language models analyze text in sequence, either left to right or right to left. This strategy confines the model's knowledge to the immediate context before the desired word. BERT has a bidirectional strategy, considering both the right and left context of words in a phrase; rather than examining the text in sequence, BERT examines all of the words in the sentence simultaneously. BERT is pre-trained on a massive set of unlabeled textual information. The approach learns contextual embeddings, which are word representations considering their surrounding context in a phrase. BERT performs several unsupervised pre-training tasks. For example, it may learn to predict the absence of words in a sentence, comprehend the relationship between two phrases, or guess the following words in a pair(Kenton & Toutanova, 2019b).

BERT is designed to generate a language approach; therefore, only the encoder architecture is used. A series of tokens are supplied to the Transformer encoder. The tokens in question are first converted into vectors before being examined by the neural network. The end output is a collection of vectors, each corresponding to an input token and including contextualized representations.

While learning language models, specifying a prediction target is difficult. Numerous models forecast the next word in a series, which is a directed strategy that may impede context learning; BERT solves this difficulty by implementing two novel training approaches Masked Language Model (MLM) Next Sentence Prediction (NSP).

CodeBERT is an expansion of the BERT introduced by Microsoft in 2020. It is a mixed pre-trained framework for programming language (PL) and natural language (NL), capable of handling several downstream (NL-PL) tasks (Feng et al., 2020)This model was trained using NL-PL pairs in numerous programming languages. It uses the same model architecture as BERT. Still, it is trained on parallel code and text data for code summarization, code search, and vulnerability detection.

In this experiment, we trained the Codebert model on two datasets, once on 1,000 and another on 5,000 records. To prevent overfitting, the dataset was split into 60% for training, 30% for validation, and 10% for testing. The validation set helped tune the model's hyperparameters and prevent it from fitting too closely to the training data. The test set, which was never seen by the model during training or validation, allowed for an unbiased evaluation of its performance.

The CodeBERT model, which has been designed using the Transformer framework, was employed for forecasting security bugs in source code. The pre-trained base model and the tokenizer were first imported from the Hugging Face library. CodeBERT is specifically designed to be capable of representing source code snippets well.

The validation and training sets were generated by loading JSON files

containing labeled code chunks that are utilized to denote the existence or non-existence of a vulnerability.

To augment the learning context, every code chunk was followed by some extra metadata, including CWE vulnerability IDs as well as project names, to provide more extensive information at training. The data preprocessing involved tokenizing the concatenated text inputs using the RobertaTokenizer with padding and truncation to standardize input lengths. Labels were converted into TensorFlow tensors compatible with the model. The model was constructed with the Adam optimizer and a learning rate of 2e-5, and trained using the Sparse Categorical Crossentropy loss function, which accommodates non-activated logits for multi-class classification. The training was executed for three epochs with a batch size of eight to promote convergent stability and validation tracking. Once trained, the model weights were also exported to an external file(model_weights.h5) and were used subsequently in the eval uation and inference process. See figure3.4.

Two models were produced from the fine-tuning process: CodeBERT-1000 and CodeBERT-5000 models. This fine-tuning process helps the model adapt to predicting software vulnerabilities. The performance of the trained models was then tested on three datasets (1,000, 5,000, and 20,000 records) using the primary metrics of accuracy, precision, recall, and F1 score to evaluate the effectiveness of the CodeBERT in detecting software vulnerabilities compared to LLMs trained on texts such as GPT and Llama. CodeBERT integrated into software vulnerability detection further extends its capability to fill the gap between natural language semantics and code structure, showcasing the model's great potential for improving automated vulnerability analysis in large systems.

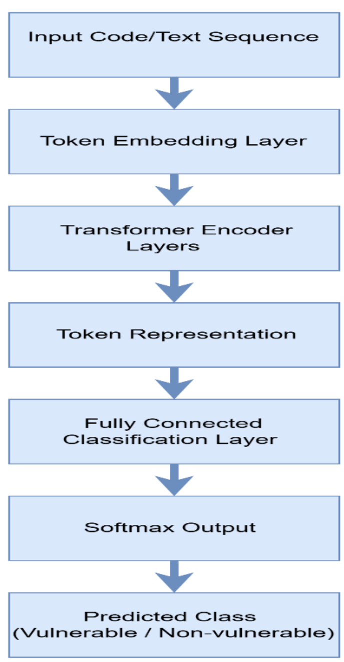

Figure 3.4: Network Architecture of CodeBERT Used for Fine-Tuning

Hyperparameters are high-level factors that control the training model procedure. They maintain control over the model's behavior and efficiency. Choosing the optimum parameters is critical for the research because insufficient configurations can lead to incorrect results. Hyperparameters cannot be predicted; hence, the ideal configuration must be determined by experience. In the literature, repeating through a vast pool of variable combinations and evaluating them for optimal performance is known as "hyperparameter tuning" and "hyperparameter optimization" (Feurer & Hutter, 2019)Therefore, we tried to carefully optimize the hyperparameters of this research to achieve high model performance and computational efficiency. However, due to limited resources, we used a manual search strategy. This method, which was based on previous experience and the literature, involved testing different value combinations within a predetermined range. Our goal was to strike a balance between computational efficiency and model performance. The limited resources forced us to concentrate on these crucial parameters, which were modified for the best results, even though an extended search could have been ideal. Table 3.6 shows the final parameters that were employed in the training CodeBert.

Table 3.6:Hyperparameter settings for training CodeBert

Parameter	Value	Description
Learning rate	2e-5	The model's step size while adjusting its parameters.
Epochs	3	The number of times the whole dataset is run across the model
Batch size	8	Number of training instances handled in one iteration
Optimizer	Adam	An optimization algorithm was employed for training
Loss function	SparseCategorical Crossentropy	This function is used to classify tasks with integer labels.

The amount of the training dataset may influence the model's performance. The potential impact increases as the dataset size decreases. To show how data size affects the trained model's performance measures, we built two datasets with identical quantities of data in each label class, with the only difference being the total size, as mentioned in Section 3.6; Table 8 explains this in detail. The goal of constructing this dataset was to explore whether additional training data influences the model's performance.

After experimenting with measuring the performance of LLMs on the DiverseVul dataset to predict software vulnerabilities, we use three models of machine learning models in this experiment; these models are known for their effectiveness in the field of classification. At first, the data was divided into three groups with three sizes, 1000, 5000, and 20000 records, to determine the extent to which the data size affected the performance of the models. Then, the data was divided into 80 % for training and 20% for testing. The TF-IDF technique converted the programming text into a numerical representation. following three machine learning algorithms were used, known for their effectiveness in the field of classification

1. Random Forest: Its principle is based on building decision trees and collecting their results
2. Decision Tree: It works on dividing the data based on specific criteria to reach the best division
3. Support Vector Machine: works on determining the optimal boundaries between classes using a linear kernel.

The details of these three algorithms are explained in detail in section 3.4.2.

3.4.2 ML Algorithms Description

3.4.2.1 Logistic Regression (LR)

LR is a form of supervised machine learning utilized for classification problems. It is a statistical procedure that examines the relationship between two data variables to predict whether an instance belongs to a specified class (Zaidi & Al Luhayb, 2023). This approach is based on the logistic function, a sigmoid function (2), as seen in Figure 13. This function in LR converts a linear combination of weights of variables into absolute values ranging from 0 to 1. These actual values might be regarded as probabilities: instead of forecasting a class, predicting the likelihood of belonging to a specific data class.

$$\sigma(x) = \frac{1}{1 + e^{\{-x\}}} \qquad (2)$$

Where: $\sigma(x)$: The sigmoid function, output between 0 and 1. x: Characteristics of linear combination. e: Euler's number, which is roughly equivalent to 2.71828. $e^{\{-x\}}$: The Exponential decay Scales the output. $1 + e^{\{-x\}}$: To make sure that the function is bounded between 0 and 1. $\frac{1}{1 + e^{\{-x\}}}$: to convert x Into a probability.

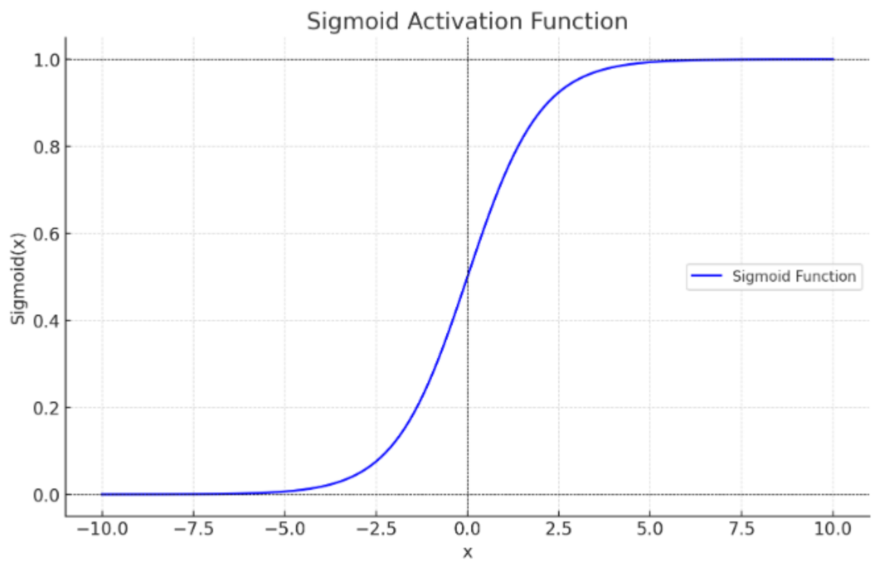

Figure 3.5: The sigmoid activation function

In this research, we used LR, an effective model for binary classification tasks, to measure its effectiveness in predicting network intrusion. The Maximum iteration was set to 1000 to ensure convergence to the optimal solution, especially with high-dimensional data (max_iter=1000). We also used the (random state=42) property to ensure that the same results are reproduced when the code is rerun to achieve stability and reliability.

3.4.2.2 Random Forest (RF)

The RF algorithm is a powerful tree-learning method. It generates numerous Decision Trees during the training phase. Each tree is constructed by selecting a random chunk of the data set and evaluating a random number of features for each division. This unpredictability adds variability to the separate trees, reducing the risk of overfitting and improving overall prediction accuracy (L. Zhang & Suganthan, 2014).

In prediction, the algorithm aggregates the outputs of all trees using voting (for classification tasks) or averaging (for prediction tasks). This cooperative decision-making process, supported by several trees and their insights, produces consistent and exact results. RF is often used for functions such as regression and classification because it deals with intricate data, reduces overfitting, and provides accurate forecasts in various circumstances, as shown in Figure 3.6.

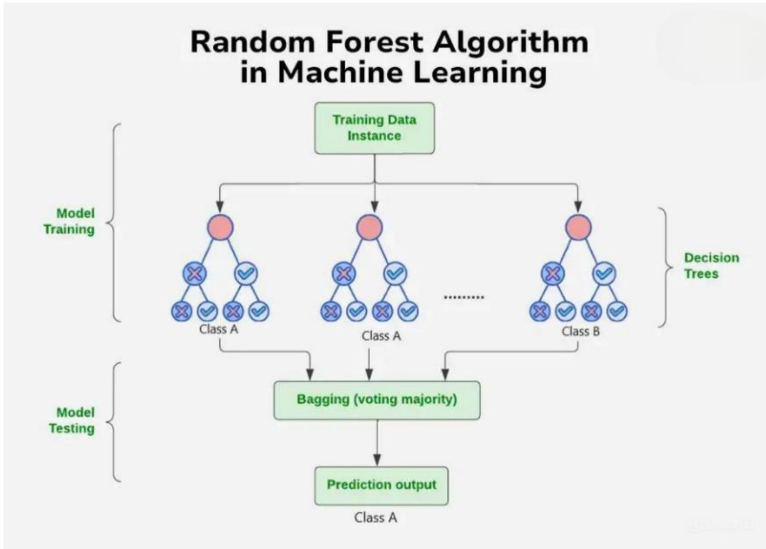

Figure 3.6: Random Forest algorithm in machine learning (L. Zhang & Suganthan, 2014)

In this research, we used the RF model to predict intrusion on the network. In the training phase, the model was adjusted using 100 decision trees (n_estimators=100), and the tree depth reached 10 levels (max_depth=10) to achieve balance for the model. To ensure the repeatability of the models, random_state was set to a value of 42 (For more details, see Appendix E).

3.4.2.3 Support Vector Machine (SVM)

SVM is a strong machine learning technique commonly used for linear and nonlinear classification, regression, and identification of outliers' applications. SVMs are highly adaptable and thus appropriate for many applications, such as recognizing texts, picture classification, identifying spam, handwriting recognition, and detecting anomalies.

SVMs are highly efficient as they seek the largest separation hyperplane among many classes in the desired feature, rendering them suitable for use in binary and classification with multiple classes, so the fundamental goal of the SVM method is to find the best hyperplane in a space with N dimensions which efficiently divides points of data into various categories in the domain of features. The approach maximizes the distance between the nearest points of distinct classes, often referred to as support vectors. The number of features determines the hyperplane's dimensions. For example, if only two feature inputs, the hyperplane appears as a line; if three input characteristics exist, the hyperplane transforms into a two-dimensional plane. As the number of features exceeds three, the difficulty of displaying the hyperplane increases (Pisner & Schnyer, 2020). See Figure 3.7.

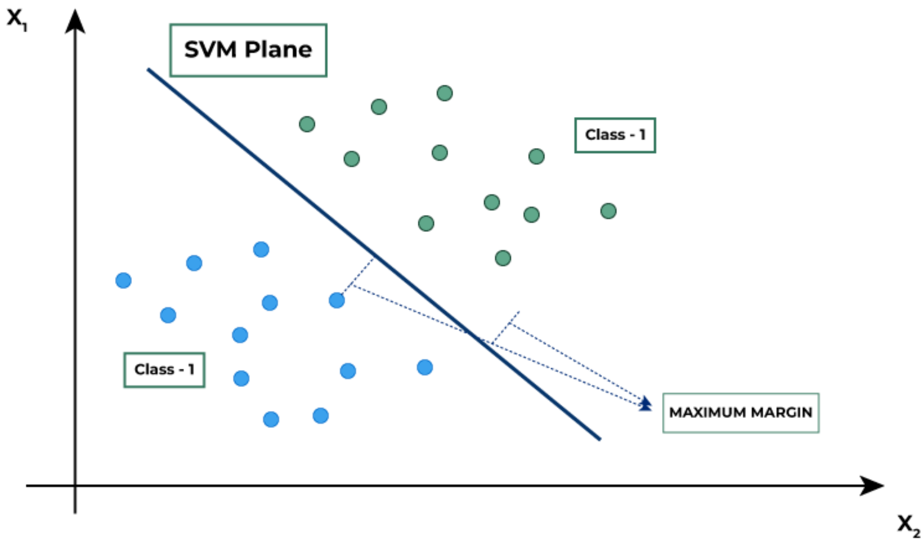

Figure 3.7: Support Vector Machine (SVM)

In this research, we used the SVM algorithm to measure its ability to detect intrusion on the network. During the training phase, RBF (radial basis function) was chosen to improve the model's ability to deal with non-linear relationships in the data (kernel='rbf) and increase the model's balance; the C coefficient was set to 1.0 (C=1.0).

3.4.2.4 Decision Tree (DT)

Decision Trees (DTs) are a non-parametric supervised learning technique used in classification and regression. The goal is to build a model that can predict the value of a target variable based on basic decision rules derived from data attributes. A tree represents a piecewise constant approximation. DT is a popular and effective technique used in a range of fields, including data analysis, machine learning, and analytics. They model the interplay of various elements, making it simple to make data-driven decisions. A decision tree is a flowchart-style design used to make decisions or predictions. It is made up of nodes that represent attribute selections or tests, branches that reflect the outcomes of these selections, and leaf nodes that indicate the ultimate results or predictions. Each internal node represents an attribute test, each branch represents the assessment outcome, and each leaf node represents a category labeling or a continuous value (Charbuty & Abdulazeez, 2021; Hafeez et al., 2021).

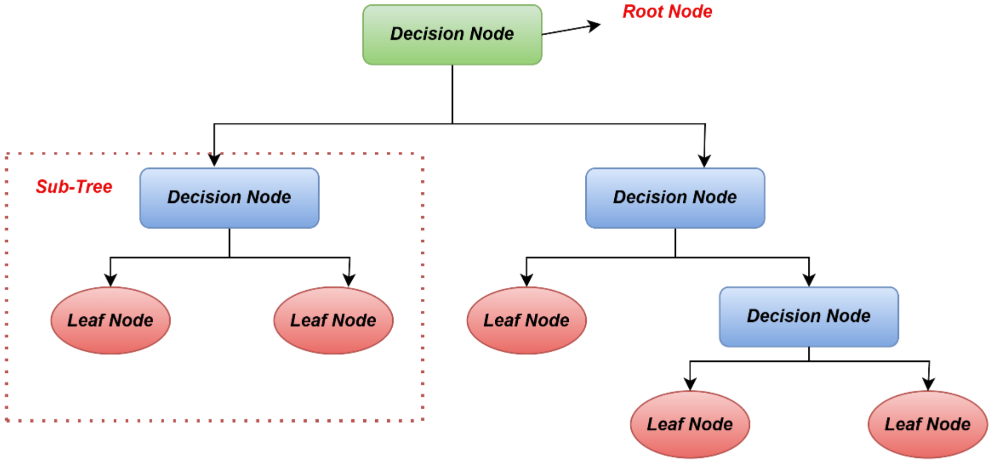

Figure 3.8: Decision Tree (DT) Structure

This research used the DT algorithm to measure its ability to predict network intrusion. The tree depth was set to 10 levels during model training to avoid overfitting.

Hyperparameters, the high-level variables influencing the training model approach, are crucial in controlling the model's behavior and efficiency. Their importance cannot be overstated, as choosing the best parameters is essential for the study's accuracy and success. This emphasis on their significance makes the audience feel the weight of their role in the research. (Feurer & Hutter, 2019)Therefore, we attempted to properly tweak this research's hyperparameters to achieve excellent model performance and computational efficiency; Table 3.7 shows the hyperparameter configuration used for training the ML algorithm.

Table 3.7: Hyperparameter configuration for training ML

Model	Parameter Value
Logistic Regression	max_iter =1000 random_state =42
Decision Tree	max_depth =10 random_state =42
Random Forest	n_estimators =100 max_depth =10 random_state =42
Support Vector Machine	Kernel =rbf C =1.0 Gamma =scale random_state =42

3.5 Test Environment Setup

The experiment was evaluated in a clearly defined test environment to ensure its stability and reproducibility. The project repository includes datasets for training and testing and code for fine-tuning BERT and using GPT and Llama in zero- and few-shot contexts. It also features scripts for evaluating model performance, comparing results, and visualizing insights, all designed to ensure a structured workflow that can be reproduced for the research work and experimental scripts, uploaded to GitHub (*Walaashe/Software-Vulnerability-Detection-Using-LLM*, n.d.)

3.5.1 Hardware/Software Requirements

The experiments in this work will be carried out on a system equipped with the machine running the Windows 11 Pro operating system; an 11th Gen Intel (R) Core (TM) i5-1135G7 processor powered it at 4 cores and 8 logical processors, running at 2.40GHz based. The system was supported with 8GB RAM to ensure smooth data processing for efficient model training. Graphics processing is catered to by the integrated Intel(R) Iris(R) Xe Graphics supported by a dedicated NVIDIA GeForce MX350 GPU for the smooth performance of the deep learning task executions. The software environment includes Python 3.11, with the essential machine learning libraries TensorFlow 2.15.0(Weber et al., 2021) and Transformers 4.33.0,(Wolf et al., 2020), Which are critical for training and testing transformer-based models(Gillioz et al., 2020), and evaluating their performance on the datasets used in this study, libraries such as pandas, NumPy, and scikit-learn were used for the machine learning models and evaluation (Rajamani & Iyer, 2023)Visualization tasks were performed using Seaborn and Matplotlib in Visual Studio Code's development environment. This setup performed well during the experiment.

3.6 Ensuring Data Validity and Reliability

For software vulnerability detection, the datasets were carefully selected and verified for their validity and reliability to obtain strong performance from the experiment; the data was obtained from known accurate sources and ensured that they represented realistic scenarios in the dataset for the software vulnerability classification task; we received it through (Chen et al., 2023).The data consists of abnormal and normal samples, and we were keen on a balanced distribution to avoid bias (see Table 8).

To ensure that the distribution of zero and one categories was almost equal across all groups, the data was divided into groups with 1000, 5000, and 20,000 records. This step prevented bias towards a particular category by exposing the model to all categories equally and testing the performance of LLMs (GPT, Llama, and BERT) on these data while ensuring that they were unseen data to the models.

As for the training data for the BERT model, the data was divided during the training phase into three parts: 70% for training, 20% for validation, and 10% for testing on data size of 1000 and 5,000 records, taking into account that the distribution of zero and one categories across all groups was almost equal. This ensures that the model is exposed to all categories equally during the training and testing. Thus, we provide the model is not biased towards any categories, contributing to the results' reliability; Table 3.8 indicates that.

As for network intrusion detection, the datasets were carefully selected and verified for their validity and reliability to obtain strong performance from the experiment; the data was obtained from known accurate sources and ensured that they represented realistic scenarios in the dataset for the network intrusion detection task; The dataset was obtained from Kaggle (NSL-KDD). The data consists of normal and abnormal traffic, and we were keen on a balanced distribution to avoid bias (see Table 5 in section ***3.3.2.1)***. To ensure the accuracy and reliability of the results, and to test datasets to test the performance of machine learning models LR, RF, SVM, and DT on these data while ensuring that they are unseen data to the models.

Table 3.8: Summary of Dataset Distribution and Splits

Dataset type	Total size	Target 1	Target 0
BERT Training Dataset 1	5000	2370	2630
~~Split~~			
Training	3500	1645	1855
Validation	1000	485	515
Testing	500	240	260
BERT Training Dataset 2	1000	475	525
~~Split~~			
Training	700	334	366
Validation	200	95	105
Testing	100	46	54
Additional testing			

Test set 1	1000	500	500
Test set 2	5000	2500	2500
Test set 3	20,000	9583	10417

3.7 Inference Performance (Latency)

The average time to produce a prediction is a significant parameter for assessing the efficacy of an LLM. The model's synchronous applications suffer greatly when the inference latency is too high. A rapid approach could allow for real-time discovery of vulnerabilities during development, significantly increasing the security of the software. We test the latency of our LLM by executing the prediction process with 1000 samples. The experiment will be carried out on a laptop to determine the viability of the real-time detection scenario, the specifications for the hardware of which are described in Section 3.5.1.

3.8 Frameworks for Evaluation

Evaluation metrics measure a statistic or artificial intelligence model's performance and efficacy. These metrics offer details about how well the algorithm performs and aid in comparing alternative models or algorithms.

In this study, we used several well-known metrics to ensure a comprehensive evaluation of the performance of all models equally.

Accuracy: It measures the number of correct predictions from the total predictions and is calculated as a percentage.

$$Accuracy = \frac{TP+TN}{TP+TN+FP+FN} \qquad (3)$$

precision: The percentage of positive values correctly predicted out of the total positive predictions.

$$Precision = \frac{TP}{TP+FP} \qquad (4)$$

Recall: The ability of the model to detect all actual positive values in the data.

$$Recall = \frac{TP}{TP+FN} \qquad (5)$$

F1 (F1-Score): The harmonic mean between precision and recall.

$$F1 = \frac{2*Precision*Recall}{Precision+Recall} \quad (6)$$

where TP (true positive), TN (true negative), FP (false positive), and FN (false negative). (Maree & Shehada, 2024).

We also added each confusion matrix and AUC-ROC to provide a complete picture of the model's performance in the intrusion detection task.

The matrix is a graph that shows the number of (TP), (FP), (FN), and (TN) in every category. The matrix's columns reflect the predicted class labels, while the rows reflect the actual class. The confusion matrix is a valuable tool for evaluating the efficiency of classification models. It allows us to assess the model's overall performance, its effectiveness in every category, and where errors arise. We construct several metrics by examining the values in the confusion matrix; this allows us to evaluate the model's advantages and disadvantages.

Chapter Four: Result

Experimental Results and Analysis

In this section, we present the results we obtained from the experiments during this study. In the first section, we present the results of large language models and machine learning algorithms in the task of predicting software vulnerabilities. In the second section, we present the results of machine learning models in predicting network intrusions.

Part 1: LLMs to Predict Software Vulnerabilities

4.1 GPT and Llama Performance (zero-shot and few-shot)

In this experiment, we evaluated the performance of both GPT and LLAMA models in zero-shot and few-shot cases to measure their ability to analyze software code and detect software vulnerabilities. In the case of zero-shot, the performance of the models was measured without training them on any examples, while in the case of few-shot, five examples were provided to guide the models to improve their performance.

These examples are as follows: 1. (PUBLIC bool file_save(const char fname, const void *packet, size_t length) { int fd = open(fname, O_WRONLY|O_CREAT|O_TRUNC, 0644); if (fd == -1) return false; if (write(fd, packet, length) != (size_t)length) { close(fd); return false; } close(fd); return true; }) Answer: 'yes'

2. (int init_dummy_netdev(struct net_device *dev) { memset(dev, 0, sizeof(struct net_device)); dev->reg_state = NETREG_DUMMY; atomic_set(&dev->refcnt, 1); return 0; }) Answer: 'no'

3. (append_numopt(char *s, const char *opt, unsigned int num) { char buf[32]; snprintf(buf, sizeof(buf), %u, num); return append_opt(s, opt, buf); }) Answer: 'no'

4.(static int validate_tmpl(int nr, struct xfrm_user_tmpl *ut, u16 family)\n{\n\tu16 prev_family;\n\tint i;\n\n\tif (nr > XFRM_MAX_DEPTH)\n\t\treturn -EINVAL;\n\n\tprev_family = family;\n\n\tfor (i = 0; i < nr; i++) {\n\t\t/ We never validated the ut->family value, so many\n\t\t * applications simply leave it at zero. The check was\n\t\t * never made and ut->family was ignored because all\n\t\t * templates could be assumed to have the same family as\n\t\t * the policy itself. Now that we will have ipv4-in-ipv6\n\t\t * and ipv6-in-ipv4 tunnels, this is no longer true.\n\t\t */\n\t\tif (!ut[i].family)\n\t\t\tut[i].family = family;\n\n\t\tswitch (ut[i].mode) {\n\t\tcase XFRM_MODE_TUNNEL:\n\t\tcase

XFRM_MODE_BEET:\n\t\t\tbreak;\n\t\tdefault:\n\t\t\tif (ut[i].family != prev_family)\n\t\t\t\treturn -EINVAL;\n\t\t\tbreak;\n\t\t}\n\t\tif (ut[i].mode >= XFRM_MODE_MAX)\n\t\t\treturn -EINVAL;\n\n\t\tprev_family = ut[i].family;\n\n\t\tswitch (ut[i].family) {\n\t\tcase AF_INET:\n\t\t\tbreak;\n#if IS_ENABLED(CONFIG_IPV6)\n\t\tcase AF_INET6:\n\t\t\tbreak;\n#endif\n\t\tdefault:\n\t\t\treturn -EINVAL;\n\t\t}\n\n\t\tswitch (ut[i].id.proto) {\n\t\tcase IPPROTO_AH:\n\t\tcase IPPROTO_ESP:\n\t\tcase IPPROTO_COMP:\n#if IS_ENABLED(CONFIG_IPV6)\n\t\tcase IPPROTO_ROUTING:\n\t\tcase IPPROTO_DSTOPTS:\n#endif\n\t\tcase IPSEC_PROTO_ANY:\n\t\t\tbreak;\n\t\tdefault:\n\t\t\treturn -EINVAL;\n\t\t}\n\n\t}\n\n\treturn 0;\n}), Answer: 'yes'

5.(static int verify_one_alg(struct nlattr **attrs, enum xfrm_attr_type_t type)\n{\n\tstruct nlattr *rt = attrs[type];\n\tstruct xfrm_algo *algp;\n\n\tif (!rt)\n\t\treturn 0;\n\n\talgp = nla_data(rt);\n\tif (nla_len(rt) < xfrm_alg_len(algp))\n\t\treturn -EINVAL;\n\n\tswitch (type) {\n\tcase XFRMA_ALG_AUTH:\n\tcase XFRMA_ALG_CRYPT:\n\tcase XFRMA_ALG_COMP:\n\t\tbreak;\n\n\tdefault:\n\t\treturn -EINVAL;\n\t}\n\n\talgp->alg_name[CRYPTO_MAX_ALG_NAME - 1] = '\\0';\n\treturn 0;\n},) Answer: 'no'.

Experiments were conducted on three different data sets consisting of 1,000 records, 5,000 records, and 20,000 records, then the ability of the models in each case was determined by measuring the accuracy of the two models to answer the following two research questions:

1) How effective are the LLMs in identifying software vulnerability (Zero-shot and Few-shot cases)?

2) Do the GPT and Llama models have the ability to understand and analyze the code?

The answer to these two questions will be through analyzing the results of the experiment recorded in Table 4.1:

Table 4.1: Accuracy of GPT and Llama Models Across Dataset Sizes

Model	Dataset	Zero-Shot Accuracy	Few-Shot Accuracy
GPT	1,000	53.40%	50.1%
GPT	5,000	60.4%	54.91%
GPT	20,000	60.19%	53.93%
Llama	1,000	53.83%	43.8%
Llama	5,000	59.9%	41.1%
Llama	20,000	52.99%	33.68%

The table shows the performance of the GPT and Llama models in the zero-shot and few-shot cases on three different dataset groups (1000, 5000, 20.000). The results showed better performance for the models in the zero-shot case than in the few-shot. In the first data set of 1000 records, the accuracy of GPT was 53.40% in the zero-shot case compared to 50.1% in the few-shot case. As for the Llama model on the same data, the accuracy in the zero-shot case was 53.83% compared to 43.8% in the few-shot case. As for the 5000 datasets, the accuracy of GPT and Llama in the zero-shot case was 60.4% and 59.9%, respectively, compared to 54.91% and 41.1% in the few-shot case for both GPT and Llama. Finally, for the 20.000 datasets, the accuracy of GPT was 60.19%, and the accuracy of Llama was 52.99% in the case of zero-shot versus 53.93% and 33.68% in the case of few-shot

The results showed that the performance of the models in the zero-shot case outperformed the few-shot case, which indicates that the models in the zero-shot case depend on the general knowledge acquired during the pre-training phase, which includes many fields. This superiority is due to the nature of the programming code itself, which is built on specific rules and patterns that are exploited in the knowledge acquired during the pre-training phase of the models. In contrast, providing limited examples in the few-shot case can make the model scattered and confused, especially if the examples do not cover all programming patterns. These results do not mean that the models are weak in the few-shot case but confirm their strength in the zero-shot case and the importance of pre-training them to analyze the programming code. They also indicate the need to improve the comprehensive selection of examples in the few-shot case. (Brown, 2020c) OpenAI has demonstrated this through its published research on training GPT-3, where models performed better in zero-shot conditions than in few-shot conditions in many scenarios, especially when the data was small and non-comprehensive.

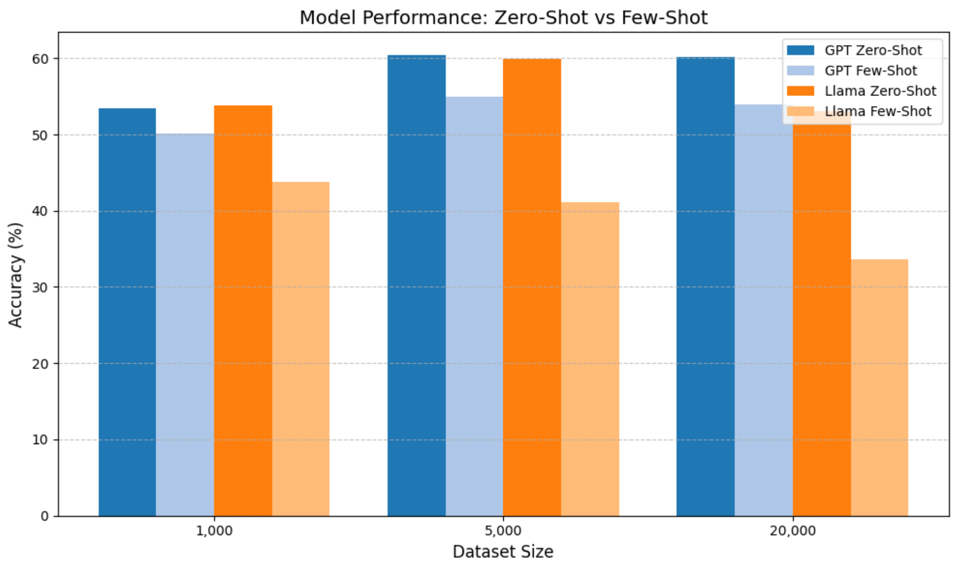

Figure 4.1: Models performance in zero-shot and few-shot across different dataset sizes

Figure 4.2 shows the performance of both models. GPT outperforms Llama in all scenarios, especially the zero-shot scenario. This is due to the generalization and knowledge acquired during the pre-training phase and the model's being affected by the examples in the few-shot scenario.

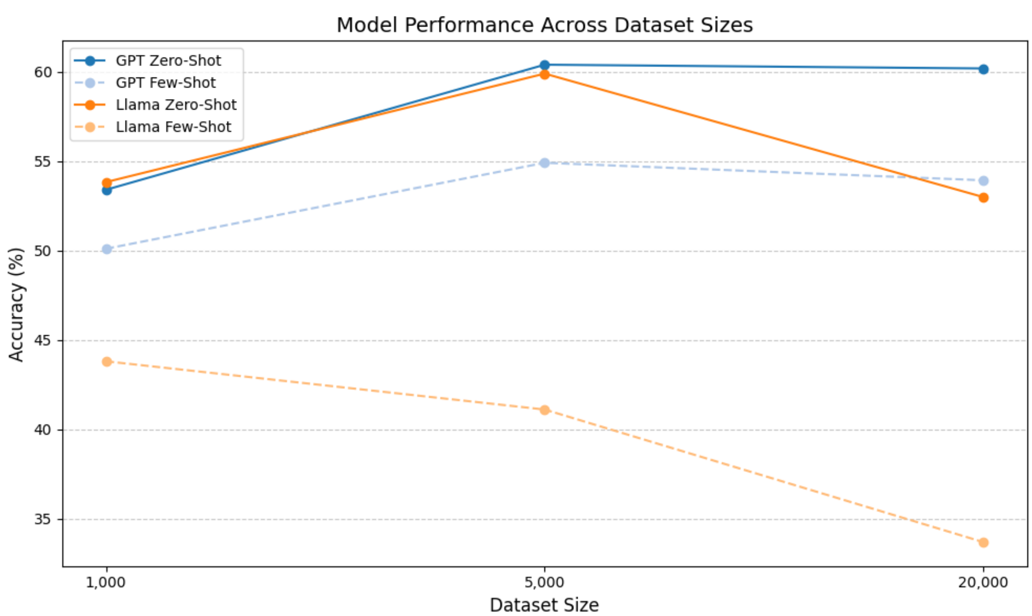

Figure 4.2: Linear representation of model performance

Figure 18 showed the GPT model's performance in the zero-shot case was superior across all data sizes. The model accuracy was stable with increasing data size until it peaked with 5000 records and achieved 60.4%. However, in the case of a few shots, the model accuracy decreased compared to zero shots, as the highest accuracy was 54.91% with 5000 records. This indicates that GPT benefits from the given examples, but in a limited way, thus making its performance more generalizable, even in zero-shots (Beltagy et al., 2022).

The results showed that the Llama model's performance in the zero-shot case was better than in the few-shot case, with a clear decrease when the data size increased to 20,000, and the accuracy reached 52.99%. However, the accuracy decreased significantly in the few-shot case versus the zero-shot case, especially with increasing data size, which indicates that it is more sensitive to the quality of the examples used during training.

Therefore, we can say that the GPT model showed greater stability and adaptability to different data sizes, whether in zero-set or few-shot, unlike Llama, whose performance seems to decline with increasing data size. This indicates the need for more improvement in exploiting and understanding examples in few-shot cases (Xian et al., 2018).We discussed this by highlighting the superiority of the zero-shot over the few-shot in many cases that require broad generalization. The study also highlights the zero-shot's flexibility in dealing with resource-limited environments.

We conclude from the results that the GPT and Llama models can understand and analyze the programming code and predict security vulnerabilities without any training or adjustment on this task specifically. Rather, these results resulted from the general knowledge acquired during the pre-training process. The model as it reached the highest accuracy of 60.4% for GPT and 59.9% for Llama. The results of zero shots for both models were good and proved our theory that the models can analyze the programming code.

However, the performance of the models is quite good but not perfect, and they are limited to a certain level of accuracy, so improvements must be made to the models. One effective method to increase accuracy is Fine-Tuning for models (X. Lin et al., 2024; R. Zhang et al., 2024), where models can be tuned to detect software vulnerabilities.

To adapt a pretrained GPT model to a domain-specific dataset of software vulnerabilities, a fine-tuning procedure was used. The effect of the dataset scale on the model's performance was evaluated using two datasets of varying sizes. There were 1,000 records in the first dataset and 5,000 records in the second. See table 4.2:

Table 4.2: Summary of the Dataset Distribution

Model	Accuracy on 1000 Dataset	Accuracy on 5000 Dataset	Accuracy on 20000 Dataset
GPT-1000	60.3%	69.8%	68.6%
GPT-5000	59.1%	61.19%	64.4%

Two models, one of which is called GPT-1000, were fine-tuned on each dataset containing 1000 records, while another, GPT-5000, was fine-tuned on a dataset containing 5000 records. All models were tested with three benchmark datasets containing 1000, 5000, and 20000 records respectively, and were evaluated on their classification accuracy. Table 1 shows classification accuracy for all models in the respective test datasets:

the GPT-1000 model surpassed the accuracy benchmarks for 5,000 and 20,000 record test sets against the GPT-5000 model, which underperformed on all datasets, and out of logic, GPT-5000 did considerably worse. One can conclude that fine-tuning on a smaller dataset (1,000 records) in this case enhanced generalization based on, in all likelihood, data quality, overfitting, or model complexity.

4.2 Consistency Check

We have experimented with GPT and Llama on 1000 records in a zero-shot scenario to ensure the model is consistent across runs. The code, which does not save memory for runs, has been rerun five times for the same dataset to make sure that the accuracy of the results does not vary with each run. According to Figure 4.2, five experiments show that in all results files, the performance of both GPT and Llama models is very similar, and the difference is minute. It depicts stability: at a zero-shot setting, these models are reliable since the difference between their scores is within a considerable limit; hence, the values do not diverge hugely, meaning the models do not contradict each other, and therefore the outcomes derived from them are reliable.

Table 4.3: Model performance stability

Iteration	GPT Results	Llama Results
1	0.65	0.452
2	0.654	0.441
3	0.652	0.44
4	0.656	0.454
5	0.658	0.468
Mean	0.654	0.451
Std Dev	0.003162278	0.011401754

According to Table 4.3, the standard deviation of both models indicates slight fluctuations in performance, while the performance of the models shows stability across multiple experiments.

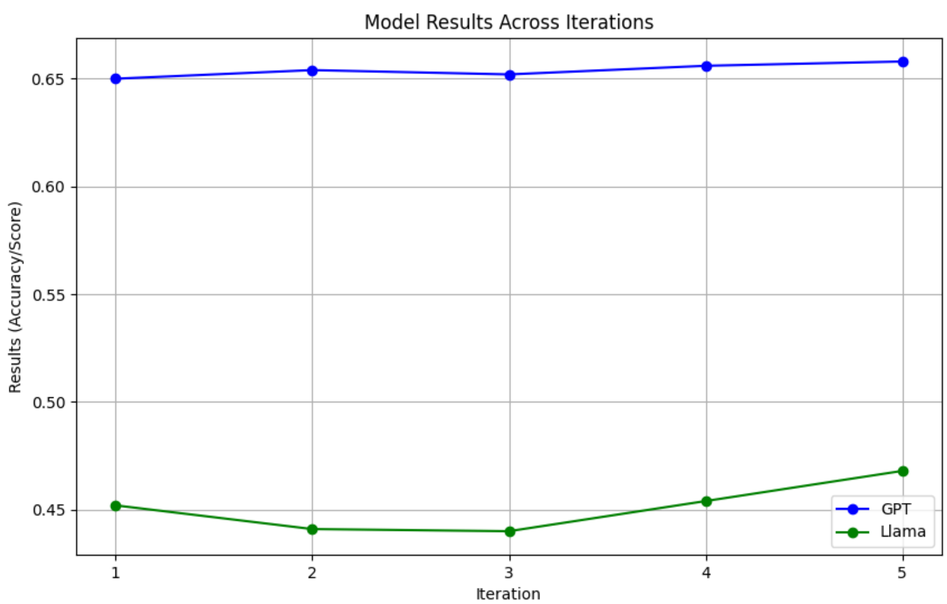

Figure 4.3: GPT and Llama stability

Previous Research has indicated this stability in GPT and LLM models, such as the research "Harmonic LLMs are Trustworthy" (Kersting et al., 2024),The results showed a high level of reliability, as the models were highly stable when repeated tests were conducted. The authors invented a method to measure the stability of the models.

4.3 Explanatory Power of GPT and Llama

In this experiment, we used a dataset consisting of 50 records to measure the performance of the models in the case of zero shots in understanding and analyzing the programming code and predicting security vulnerabilities and their types. The performance was varied, certain cases were successfully identified by the models, whereas some vulnerabilities went undiscovered. The failures could be due to the complex nature of the code, the difficulty of analyzing the relationship between its parts, or maybe a general reliance on the general linguistic context rather than on the specific programming pattern. However, they can still provide logical responses on the vulnerabilities and their types. This goes to show how far they benefit from pre-training and obtaining general knowledge about programming codes. As for the results, the models are good at handling the vulnerability detection tasks. Still, they need to be trained on specific examples or adjusted on the task to improve their capabilities and increase accuracy. For example, let us discuss the following examples and how the models responded to them:

- In case 1, a software vulnerability exists; both models provide correct explanations, as shown in Table 4.4.

Table 4.4: GPT and Llama predictions on the test case (1)

Element	Description

Input	`static int jpc_ppt_getparms (jpc_ms_t *ms, jpc_cstate_t *cstate, jas_stream_t *in)`
	`tjpc_ppt_t *ppt = &ms->parms.ppt;`
	`/* Eliminate compiler warning about unused variables. */`
	`cstate = 0;`
	`ppt->data = 0;`
	`if (ms->len < 1) {`
	`goto error;}`
	`if (jpc_getuint8(in, &ppt->ind)) {`
	`goto error;}`
	`ppt->len = ms->len - 1;`
	`if (ppt->len > 0) {`
	`ppt->data = jas_malloc(ppt->len * sizeof (unsigned char));`
	`if (! ppt->data) {`
	`goto error;}`
	`if (jas_stream_read (in, (char *) ppt->data, ppt->len)! = JAS_CAST (int, ppt->len)) {`
	`goto error;}`
	`} else {`
	`ppt->data = 0;}`
	`return 0;`
	`error:`
	`tjpc_ppt_destroyparms(ms);`
	`return -1;`

Prompt	"As a software engineer, I need a short answer with an explanation: Is there a software vulnerability in the code snippet provided in the user message?
Target	1
GPT Answer	"Yes, there is a software vulnerability in the code snippet provided. The vulnerability lies in the allocation of memory for the 'ppt->data' buffer. If 'ppt->len' is an exceptionally large value, then the multiplication 'ppt->len * size of (unsigned char)' could potentially result in an integer overflow, leading to a buffer overflow when allocating memory for 'ppt->data'. To address this vulnerability, a proper input validation check should be added to ensure that the multiplication result does not exceed the maximum value that can be represented by the data type used for the calculation. Additionally, using a safe memory allocation function like 'calloc' or 'realloc' which automatically handles size calculations based on the size of the object being allocated, could help mitigate this vulnerability."
Llama Answer	"The vulnerability is in the 'jas_malloc' function call, which is susceptible to a buffer overflow attack. The size of the memory block being allocated ('ppt->len * sizeof (unsigned char)') is not properly checked for integer overflow, and an excessively large value for 'ppt->len' could result in memory corruption. Implementing a validation mechanism to verify that the calculated size does not exceed the allowable memory limits could prevent this issue."

In this example, the function contains a security vulnerability. When analyzing the model's correct answers, we can interpret them logically for several reasons. First, the code has a memory allocation procedure (jas_malloc). It is known that if (ppt->len) is a large value, it opens the way for a (Buffer Overflow) vulnerability. The models were able to determine that the code is vulnerable to this type of vulnerability because they did not verify the result of (ppt->len * size of (unsigned char)) or verify that it does not exceed the maximum limit. Here, the models relied on their prior knowledge of the vulnerabilities they acquired during pre-training, and by linking the code to familiar security risks, they could provide accurate answers, see Figure 4.4.

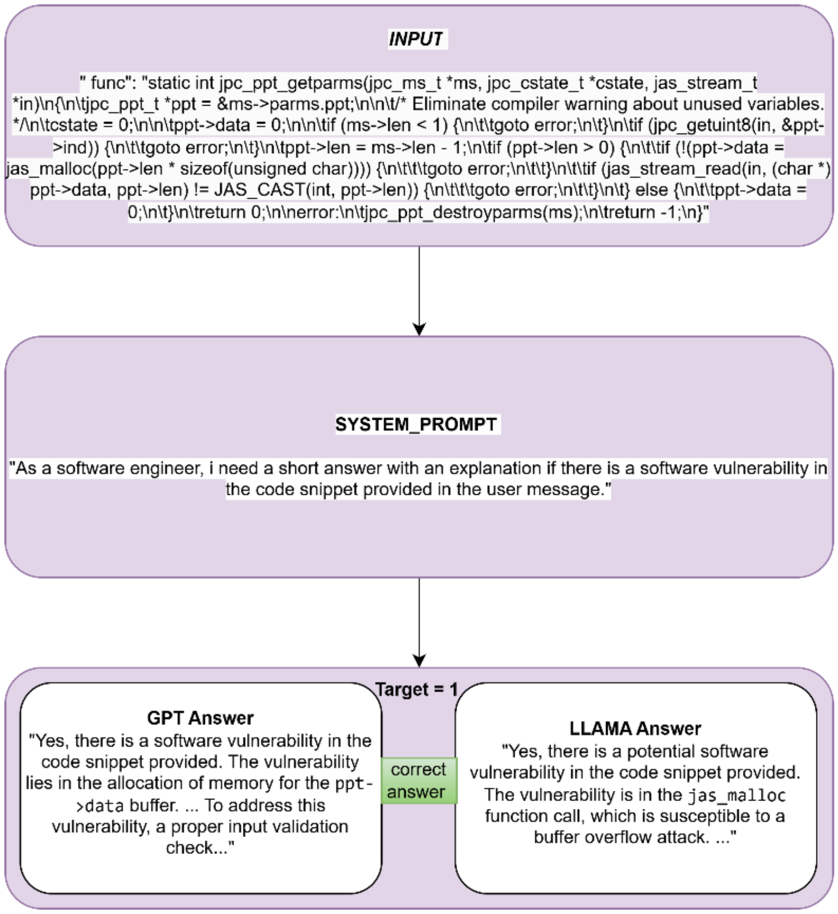

Figure 4.4: GPT and Llama predictions on the test case (1):

- In case 2, there are no software vulnerabilities; however, both models provide incorrect explanations, as shown in Table 4.5

Table 4.5: GPT and Llama predictions on the test case (2)

Element	Description

Input	`"func": "int compat_do_execve (char *filename,`
	`compat_uptr_t __user *argv,`
	`compat_uptr_t __user *envp,`
	`struct pt_regs *regs){`
	`struct linux_binprm *bprm;`

	`struct file *file;`
	 `int retval;`
	 `int i;`
	 `retval = -ENOMEM;`
	 `bprm = kzalloc(sizeof(*bprm), GFP_KERNEL);`
	 `if (! bprm)`
	 `goto out_ret;`
	 `file = open_exec(filename);`
	 `retval = PTR_ERR (file);`
	 `if (IS_ERR (file))`
	 `goto out_kfree;`
	 `sched_exec ();`
	 `bprm->p = PAGE_SIZE * MAX_ARG_PAGES - sizeof (void *);`
	 `bprm->file = file;`
	 `bprm->filename = filename;`
	 `bprm->interp = filename;`
	 `bprm->mm = mm_alloc ();`
	 `retval = -ENOMEM;`
	 `if (! bprm->mm)`
	 `goto out_file;`
	 `retval = init_new_context(current, bprm->mm);`
	 `if (retval < 0)`
	 `goto out_mm;`
	 `bprm->argc = compat_count(argv, bprm->...");`

Prompt	"As a software engineer, I need a short answer with an explanation: Is there a software vulnerability in the code snippet provided in the user message?
Target	0

GPT Answer	Yes, there is a software vulnerability in the provided code snippet. The vulnerability is related to memory management and potential use-after-free issues. Specifically, in the error handling paths (e.g., 'out:' label), the 'bpm' structure is being accessed and manipulated after it has been freed, which can lead to undefined behavior and security vulnerabilities. For example, after the 'bpm' structure is freed at the end of the function using 'kfree(bpm)', there are subsequent attempts to access fields of the 'bpm' structure such as "bpm->security", "bpm->mm", and "bpm->file". This can result in memory corruption and exploitation by a malicious actor. To address this vulnerability, the code should be modified to ensure no access is made to the 'bprm' structure after deallocating it. Proper cleanup and checks should be implemented to avoid such problematic scenarios.
Llama Answer	Yes, there is a software vulnerability in the provided code snippet. The vulnerability is in the 'compat_copy_strings' function, which is used to copy the environment and argument strings to the new memory space of the process being executed. The function uses a "strcpy" operation to copy the strings, but it does not properly check the length of the strings being copied. If an attacker can manipulate the environment or argument strings to be longer than the allocated buffer size, they can exploit this

	vulnerability to overwrite memory.

According to the function provided and the analysis of the answers of the models, we can offer a logical explanation for the failure of the two models in this test case. First, the function compat_do_execve deals with a complex technical field in operating systems, which is process execution, in addition to the code containing several dynamic operations. These operations are memory allocation (kzalloc), error checking and handling (goto), environment management, and contexts (mm_alloc, compat_copy_strings); all these complexities make it difficult for the models to analyze the code, track its context, and analyze its precise flow, which results in incorrect answers.

In short, the models here could not predict the correct answer because of the complexity of the code. There is no specific training; rather, it relies on general knowledge. However, they tried to provide answers by linking the concepts they learned. This shows their strength in generalization and emphasizes the need for specialization in tasks that require it, see Figure 4.5.

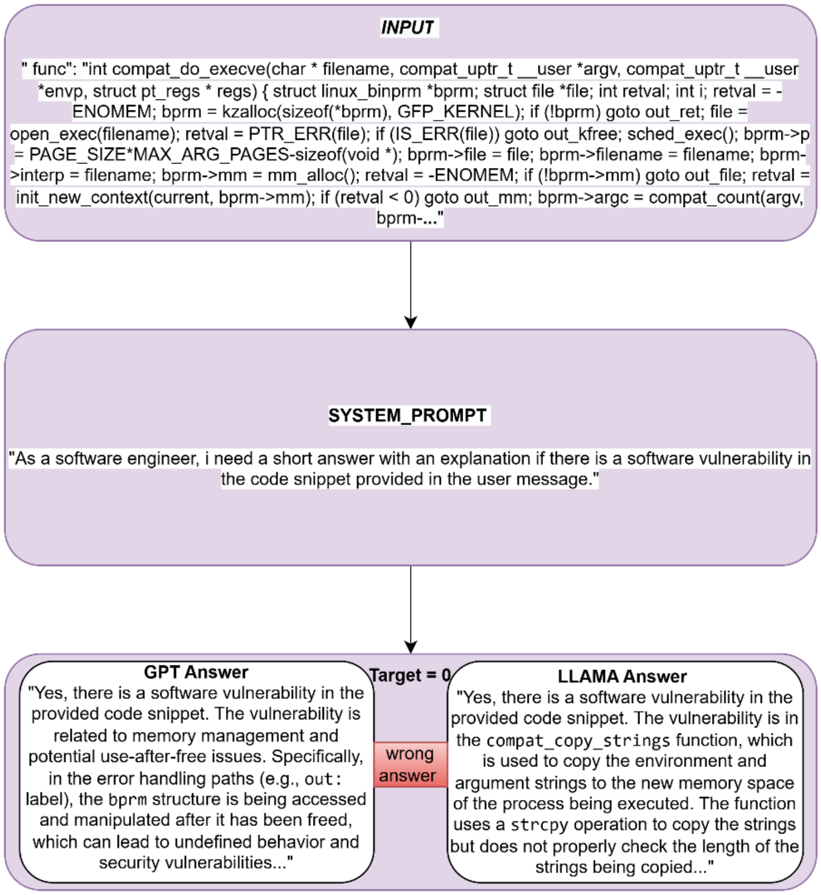

Figure 4.5: GPT and Llama predictions on the test case (2)

- As shown in Figure 4.6, there are many cases where the GPT model succeeded in providing correct predictions while the Llama model failed or vice versa. This can be explained by the difference in sources during the training of the models, and because the models were not trained specifically to detect vulnerabilities, but rather predicted based on the general knowledge acquired.

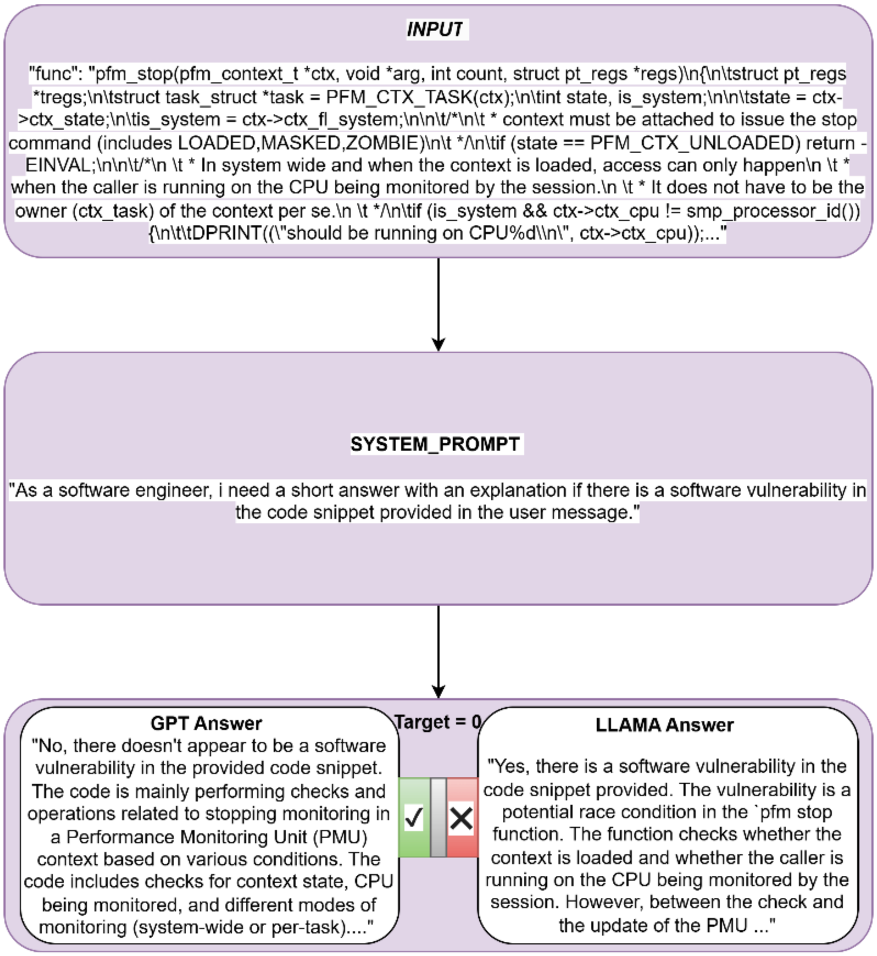

Figure 4.6: GPT and Llama predictions on the test case (3)

This experiment proves that the GPT and Llama models clearly can interpret and analyze the programming code. The answers were often logical, even in cases where they made mistakes, and their performance showed that they could provide logical explanations that supported their expectations, which reflected an understanding of the task. Despite the disparity in answers between the models, the errors are usually caused by the complexity of the code or by cases where the models were not directly exposed to them during training, highlighting their capabilities and limitations.

4.4 CodeBERT Performance

After fine-tuning the Codebert model on two datasets, once on 1,000 and another on 5,000 records. Two models were produced from the fine-tuning process: CodeBERT-1000 and CodeBERT-5000 models. The performance of the trained models was then tested on three datasets (1,000, 5,000, and 20,000 records) using the primary metrics of accuracy, precision, recall, and F1 score to evaluate the effectiveness of the CodeBERT in detecting software vulnerabilities compared to LLMs trained on texts such as GPT and Llama. Table 4.6 shows the results from this experiment:

Table 4.6: Performance Measures of CodeBERT Models Across Dataset Sizes

Model	Training data	Testing Data	Accuracy	Precision	Recall	F1 score
codeBERT-1000	1000	1000	59.3%	56.9%	76.6%	65.3%
codeBERT-1000	1000	5000	68.7%	64.5%	83.3%	72.7%
codeBERT-1000	1000	20000	67.8%	61.9%	85.4%	71.7%
codeBERT-5000	5000	1000	61.1%	71.4%	37%	48.7%
codeBERT-5000	5000	5000	67.7%	74%	54.6%	62.8%
codeBERT-5000	5000	20000	79.2%	80.6%	74.7%	77.5%

When analyzing the performance of the codeBERT-1000 and codeBERT-5000 models, we notice the effect of the size of the training data on the performance of the models when tested on three data sets. The performance of the codeBERT-5000 model outperformed the performance of the codeBERT-1000 model in all metrics, especially on large data sets. Increasing the training data size increases the model's ability to generalize and improve performance with unseen data. Considering small test data sets (1,000 samples), the F1 score for the codeBERT-1000 model was higher than that of the codeBERT-5000 model, reaching 66.72% compared to 48.75% for the codeBERT-5000 model. This means that despite the limited training data in the codeBERT-1000 model, it was more adaptable to small test data scenarios. However, as the test data size gets larger, this advantage lessens.

When tested on 1,000 and 5000 datasets, the performance of the two models was superior to that of codeBERT-5000. For example, when tested on 20,000, the codeBERT-5000 model had a higher F1 score of 77.54% than the codeBERT-1000 model's 71.67%. This superiority is due to the ability of the model trained on larger data to capture complex patterns and relationships in the data, see Figure 4.7.

In the end, small training data sets may be sufficient for small test scenarios, but larger training data is necessary to achieve high performance in larger test scenarios. This reveals a very important balance of training data size concerning the size and difficulty that one can expect in real applications.

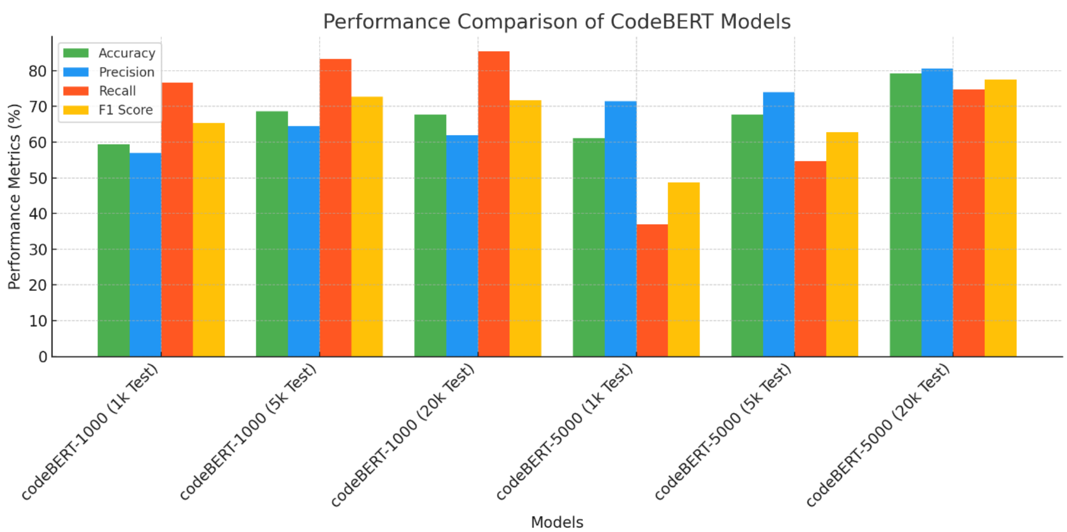

Figure 4.7: Performance Comparison of CodeBERT Models

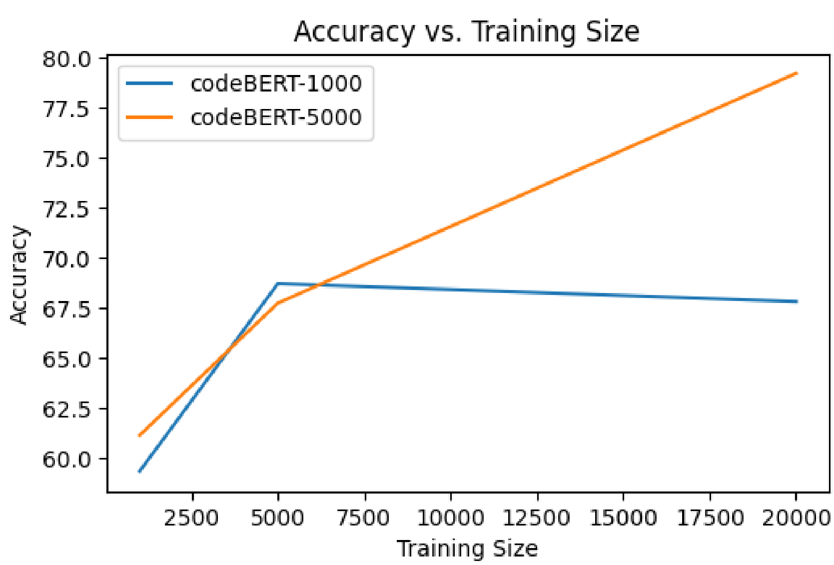

Figure 4.8: Line chart of Accuracy vs. Training size

According to Figure 4.8, the codeBERT-5000 model shows stability and efficiency with extensive data, as its performance increases with increased data and accuracy improves linearly. This indicates that the model can absorb more information to improve its performance. This is due to the structure of the model, which was trained on more extensive data. This made it more prepared to deal with extensive data due to more parameters. This made it more capable of generalization and avoiding problems of excessive complexity or fatigue.

Unlike the codeBERT-1000 model, which is evident from the graph, it suffers from overfitting as it reaches peak performance at 5000 records and then begins to decline with increasing data size.

This is because the model's structure was not trained on sufficient amounts of data, which made it unable to generalize to extensive data. The model learns very few features from small data, thus the small number of parameters acquired. It is considered a suitable model for small data scenarios but not large ones.

In short, the codeBERT-5000 model is better for dealing with extensive data due to its stability and ability to generalize, so with the increase in data size, the performance gradually decreases. Thus, we notice that increasing the number of training data contributes significantly to improving the performance of the models.

Based on these experiences, when analyzing the performance of the six LLMs (GPT-Zero Shot, GPT-Few Shot, Llama-Zero Shot, Llama-Few Shot, CodeBERT-1000, and CodeBERT-5000) in predicting software vulnerabilities, we note that the Codebert-5000 model has stability and its performance improves with increasing training data size, as the performance was strong and balanced. GPT showed a slight improvement in the zero-shot case compared to the few-shot, but its performance remains lower than the Codebert models. The Llama model suffers from fluctuations in its performance, especially in the few-shot case and with increasing data size, which means there are difficulties in benefiting from the large amount of data. Finally, we note that the Codebert model, which was pre-trained on the code, demonstrated a clear superiority in classifying software vulnerabilities, exceptionally when trained on large data sets, making it the most preferred model among the other models. Figure 4.9 shows the difference in the performance of LLMs (GPT, Llama, and BERT).

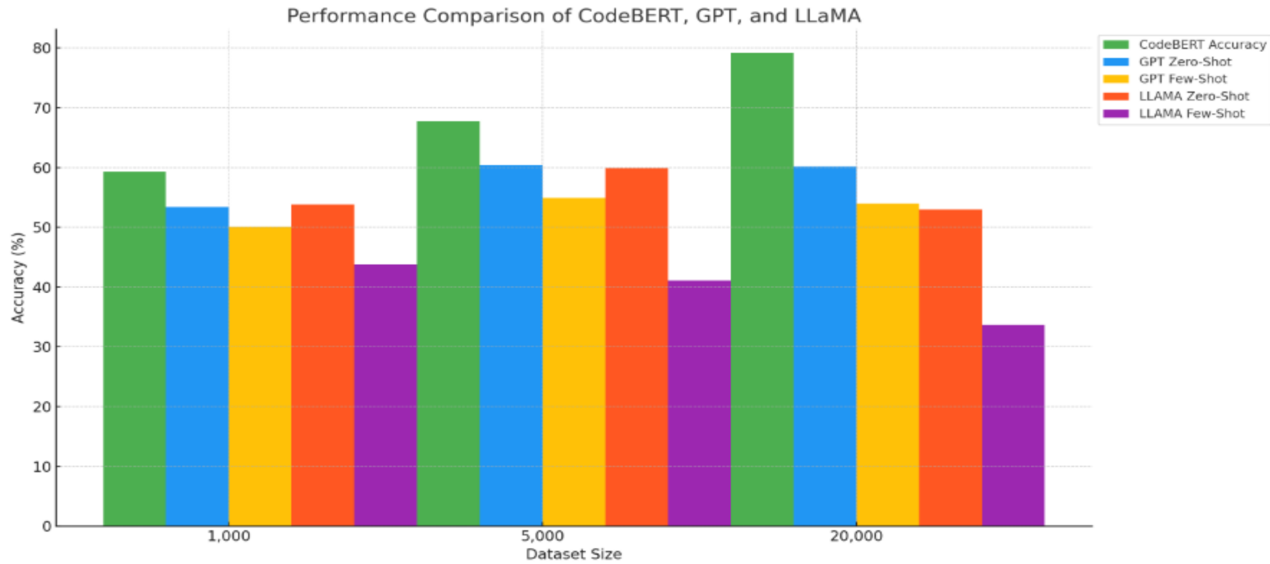

Figure 4.9: Accuracy Comparison of CodeBert-5000, GPT, and Llama

4.5 Inference Performance

In this experiment, we measured the Inference Latency of large language models using a dataset of 1,000 records. We experimented using the GPT and Llama models in the zero-shot case and the CodeBert-5000 and CodeBert-1000 models. The time taken to predict the results was as follows:

Table 4.7: models delay for forecasting 1000 records in seconds

Model	Records	Time (seconds)
GPT	1000	643
Llama	1000	901
CodeBERT-5000	1000	1800
CodeBERT-1000	1000	2132

Table 4.7 shows that the GPT model was the fastest, taking only 643 seconds to predict 1,000 records. This indicates its quick response. In contrast, the Cobert-1,000 model was the slowest, taking 2,132 seconds for the same dataset. This may be due to its unique design for processing programming codes that require complex operations. The Llama model took 910 seconds to predict 1,000 records, so its performance can be average.

It is worth noting that the CodeBert-5000 model took only 1,800 seconds less time than the model trained on 1,000 records. This means that the CodeBert 5,000 model is more efficient in managing data. Because CodeBert 5,000 has been trained on more data, it has become more capable of managing data, which is intriguing.

The interpretation is that the model trained on more extensive data learns broader patterns and, therefore, requires fewer computational adjustments to reach the prediction; unlike the model trained on 1000 records, the patterns overlap and are unclear, so it needs more time and more computational adjustments. Therefore, introducing more data will make the decision boundaries separating the classes less noisy and more refined.

In addition, with the increase in training data, the model weights become more stable. For example, the model with a thousand codes may face difficulty due to insufficient data, while the model trained on five thousand does not face trouble due to the higher confidence in its predictions and also has better generalization capabilities, which increases the speed of inference.

The authors (Valle-Perez et al., 2018) Indicated in their research that increasing the size of the data reduces the complexity of the calculations during the prediction, and other researchers, such as (Kaplan et al., 2020) Mentioned that increasing the training data makes the model more stable and reduces fluctuations during the prediction. We conclude that increasing the size of the training data improves the performance in terms of efficiency and speed of the models.

4.6 ML Algorithms Performance

In this section, we applied three machine learning algorithms to predict software vulnerabilities. These algorithms are Random Forest (RF), Support Vector Machine (SVM), and Decision Tree (DT) on three datasets, 1000, 5000, and 20,000 samples from the DiverseVul dataset; the data divided into 80% training and 20% testing, The TF-IDF technique converted the programming text into a numerical representation, we obtained the following results according to Table 4.8:

Table 4.8: Results of ML algorithm in software vulnerability detection

Data size	Model	Accuracy	Precision	Recall	F1-score
20,000	Random Forest (RF)	79.4%	79%	79%	79%
	Decision Tree (DT)	71.9%	72%	72%	72%

	Support Vector Machine (SVM)	81.5%	82%	81%	81%
5000	Random Forest (RF)	85.1%	85%	85%	85%
	Decision Tree (DT)	81.8%	82%	82%	82%
	Support Vector Machine (SVM)	90.3%	91%	90%	90%
1000	Random Forest (RF)	88%	88%	88%	88%
	Decision Tree (DT)	83.5%	83%	83%	83%
	Support Vector Machine (SVM)	88.5%	88%	88%	88%

From the table, we notice the following: In the data size of 20,000, the SVM achieved the best performance, with an accuracy of 81.5% better than RF and DT, which recorded an accuracy of 79.4% and 71.9%, respectively.

With a data size of 5000, SVM was the best, as its accuracy reached 90.3%, higher than RF's 85.1% and DT's 81.8%. This superiority in performance over Data 5000 is due to the model's ability to generalize and learn better on medium-sized data, as this size provides enough information to improve accuracy without being too large or too small.

For the 1000-data-size dataset, SVM achieved an accuracy of 88.5%, higher than RF's 88% and DT's 83.5%.

Due to its high accuracy, the SVM model seems the most effective in dealing with large data. RF and DT also performed well but less than SVM in all cases. We also note that RF performed well even with small data, which means it is stable in dealing with small data (see Figures 4.10 & 4.11).

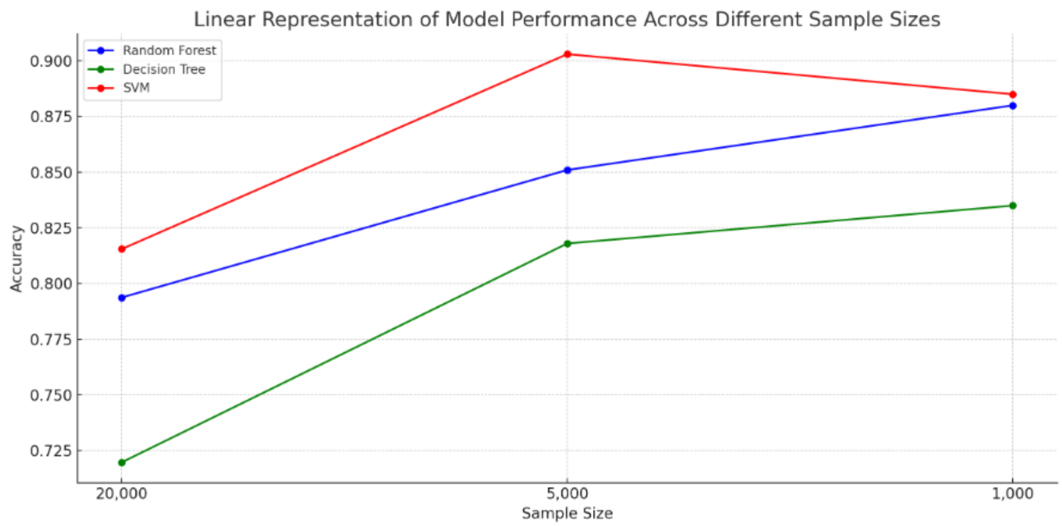

Figure 4.10: Liner Representation for ML Models Performance

This linear representation shows the comparative performance of ML models based on accuracy with data sizes. The figure reinforces the results shown in the table.

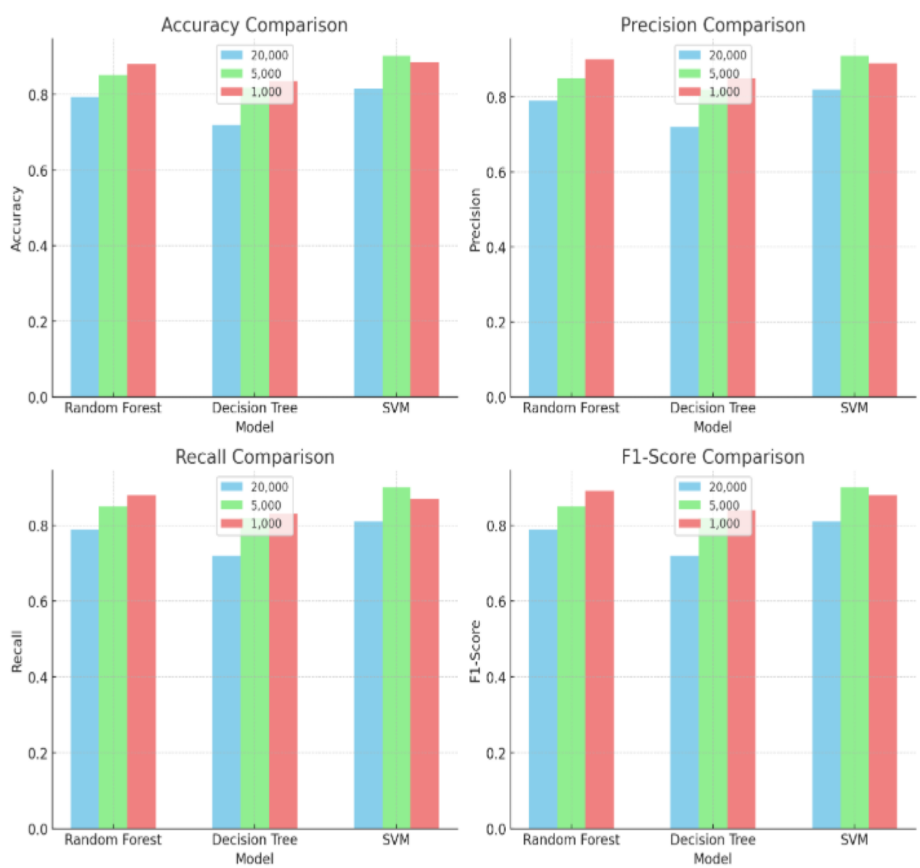

Figure 4.11: Performance metrics of machine learning models

Figure 4.11 shows a Graphical comparison of performance metrics of machine learning models in detecting software vulnerabilities, supporting previous findings.

Part 2: ML Algorithms to Detect Intrusion

4.7 ML Algorithms Efficiency

In this experiment, typical machine learning algorithms were employed to predict network incursion on the NSL-KDD dataset, an improved version of the KDD Cup 99 dataset. In addition to regular traffic, this data comprises several attacks divided into groups. As stated in Chapter 3, the data was condensed to normal and abnormal without describing the nature of the abnormality. Following the first data processing, four machine-learning techniques were selected: "LR, SVM, RF, and DT. We assessed the models' performance using five key metrics: accuracy, precision, recall, F1-score, and ROC-AUC. The models were trained using the training data and then tested on the test data (85%,15%). We obtained the following results according to the table 4.9:

Table 4.9: Accuracy of ML Algorithms on NSL-KDD DATASET

Model	Accuracy (%)	Precision (%)	Recall (%)	F1-score	ROC-AUC
Logistic Regression	97%	100%	94%	96.9%	97%
Random Forest	99.8%	99.8%	99.6%	99.8%	99.8%
SVM	96.4%	99.9%	92.8%	96.3%	96.4%
Decision Tree	100%	100%	100%	100%	100%

By analyzing the performance of the models according to Table 16, we notice that the DT algorithm provided an ideal performance of 100% in all performance metrics. Thus, it is the most efficient model for this task. As for the RF algorithm, it also provided an ideal performance close to the DT, as it achieved an accuracy of 99.8% with high recall and ROC-AUC, which indicates its high performance. On the other hand, the LR model achieved good performance. Still, it is less efficient than the DT and RF models, especially since the recall is 94%, which means the possibility of missing positive cases. Finally, the SVM recorded the lowest performance, especially in the recall by 92.8% and ROC-AUC by 96.4%, indicating challenges in accurately distinguishing the classes. In general, we notice that the performance of DT and RF achieved the highest performance. Thus, they are ideal choices for predicting intrusion on the network, while LR and SVM were good but not perfect, see Figure 4.12.

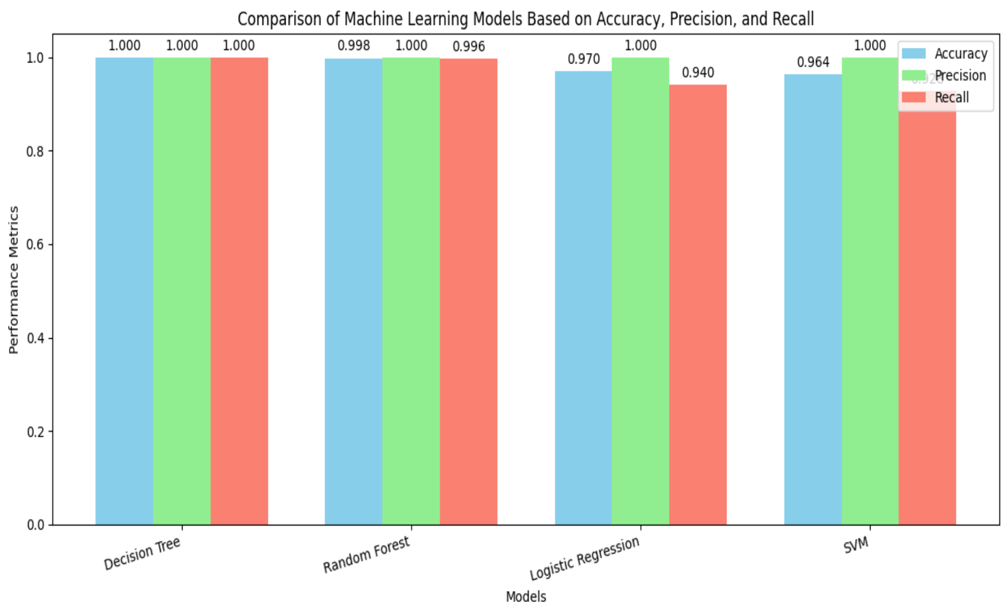

Figure 4.12: Model Performance Metrics on NSL-KDD Dataset

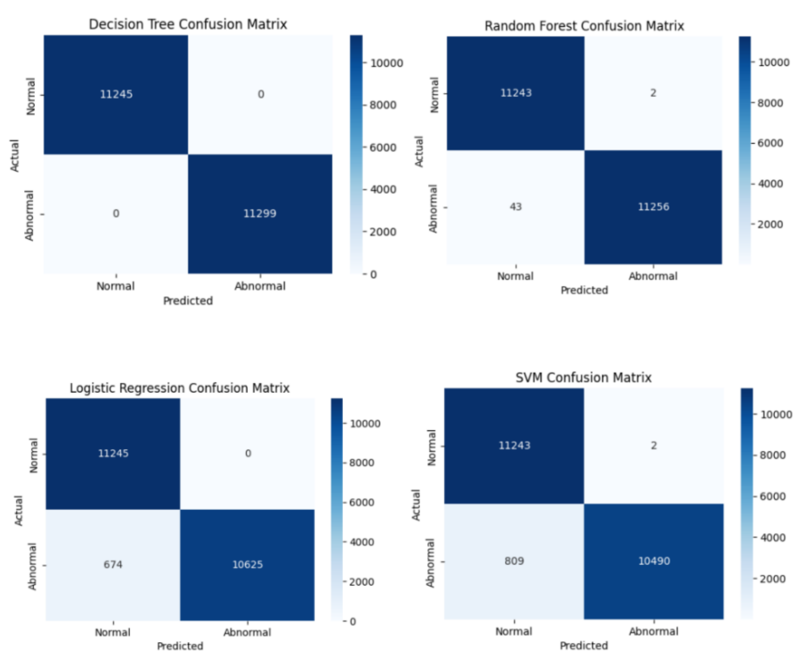

Figure 4.13: The Confusion Matrix for Models

According to the confusion matrix in Figure 4.13 of the four models used to detect intrusion on the network, we can observe the performance of each model in classifying cases as normal and abnormal. In general, DT shows a correct classification of all cases without any errors, followed by RF, which showed a small percentage of mistakes, as 43 cases of false negatives were not correctly identified and two instances of false positives. As for LR and SVM, they showed a high error rate according to the images, especially the false negative type. This means that some threats were not detected, which is more dangerous because it ignores actual threats. Accordingly, the DT is the most suitable for cases requiring high accuracy without errors, followed by the RF. LR and SVM are less reliable options, especially for detecting intrusion on the network, which requires high precision.

Chapter Five: Discussion

In this section, we assess our study results and discuss research questions from Section 1.2.

5.1 The Performance

Performance is the most important consideration when introducing a new tool to detect software vulnerabilities. Therefore, the first research question was: "Verify the effectiveness of the LLM in identifying software vulnerabilities.".

In our study, we evaluated the performance of the proposed models from multiple perspectives. First, we measured the model's performance using the primary metrics. The top-performing model has an accuracy of 79.2%, Precision of 80.6%, and F1 scores of 77.5%. This is an outstanding achievement, considering the difficulty of the task. Discovering vulnerabilities of this type requires a long time and expertise in cybersecurity. Our proposed method automates the procedure, reducing both resources and time.

Inference latency is another way to measure performance, especially in applications that require fast data processing. Table 14 shows the period that each model needs to predict every thousand cases, as one to two cases can be processed per second, depending on the model used. This is good performance for applications that require speed, accuracy, and expert knowledge in the field.

5.2 Performance Factors

we want to address the research question. "Determine the influencing elements of the effectiveness of our proposed model."

The best way to improve our method's performance is to increase the amount of the training dataset(X. Lin et al., 2024)Table 13 shows that the CodeBERT model's performance increased when trained on 5000 records compared to 1000. With the increase in training data, the performance increased even more, but due to a limitation, we could not increase the training data further.

In addition to the hyperparameters used during the training process, such as learning rate and batch size. Optimizing hyperparameters can increase the performance and effectiveness of the model (Feurer & Hutter, 2019).

5.3 Challenges and Limitations

Our third research question was "Identify the challenges and limits of the suggested paradigm." When applying LLMs to predict software vulnerabilities, we faced many challenges that affected the efficiency of the models. One of the most significant challenges we faced was the limited resources available. These models require high computing resources, such as high GPU processing capabilities and large memory capacity, to benefit from training efficiently.

Because our resources were limited, we could not increase the size of the training data, and the processing time increased.

Due to the limited resources, we could not fine-tune the models, knowing that adjusting the models on the task could have led to a significant improvement in the performance of the models. Another challenge we faced was changing the hyperparameter training settings, as we could not try all possibilities due to the limited resources, and the least experiment could take 10 hours, as the process of training the BERT code on 5,000 samples took 36 continuous hours.

Another challenge we faced when dealing with the GPT and Llama models was the number of requests allowed during the day for GPT, 500 RPM (request per minute), and 10,000 RPD (request per day)(Replicate, n.d.-OpenAI API, n.d.) For example, processing extensive data is difficult because the account allows only 10,000 requests daily, and you must wait until the next day to complete them.

Despite these challenges, the CodeBERT model achieved a good performance of 79.2%, which reflects its ability to predict vulnerabilities well. However, more improvements, whether in resources or training data, are still needed to achieve better results.

Table 16 shows the experiment results to answer the 4 and 5 research questions. We used machine learning algorithms (RF, DT, SVM, and LR) that performed well in predicting network intrusion. The best of these models was the Decision Tree, which achieved an ideal performance of 100%.

5.4 Summary

When comparing models' performance in software vulnerability, it is crucial to remember the conceptual difference between large language models (CodeBERT, GPT, Llama) and conventional machine learning models (RF, DT, SVM). Figure 5.1 shows the performance comparison of large language models and traditional machine learning models.

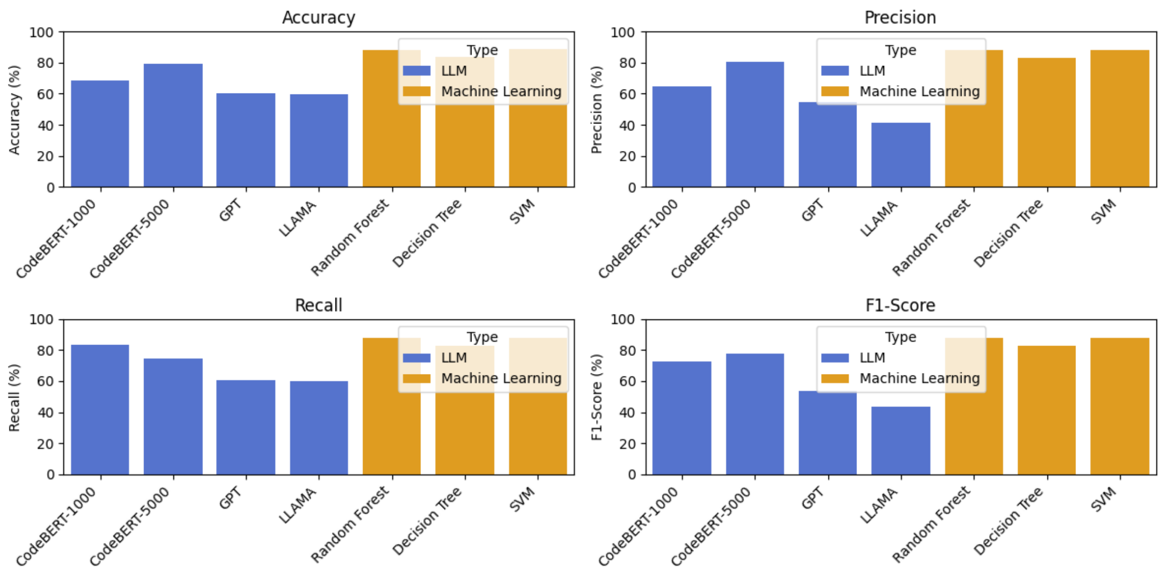

Figure 5.1: Performance Comparison: ML and LLMs

We conclude from the above that model training methods differ according to the task and the available resources. Zero-shot models, such as GPT and Llama, were the least expensive regarding resources and data as they depend on the general knowledge acquired during the pre-training phase. They provide predictions without any additional training based on the built-in capabilities of these large models. However, their performance is lower than that of other models. As for the fine-tuning approach for models such as CodeBERT, it works to customize the pre-trained models to perform a specific task, thus achieving better performance and higher and more accurate results, but it is costly as it requires significant computational resources in addition to dedicated training data and fine-tuning stages to avoid losing previously acquired knowledge, taking into account that the CodeBERT model was trained on programming data, so its structure is dedicated to understanding and analyzing programming sections, which improves its performance compared to GPT and Llama models. In contrast, supervised learning is less expensive than fine-tuning as the models are trained from scratch on the available data. However, its performance may be lower than models improved by finetuning, especially for tasks requiring deep understanding, analysis, and integrated knowledge. Accordingly, choosing the appropriate training approach should be done according to the nature of the task, the available resources, and the required level of accuracy, taking into account that comparing large language models GPT and LAMA in the zero-shot case with traditional supervised learning models is not fair, as LLM relies on its basic structure without any additional training, which reflects its ability to generalize based on its basic structure only To achieve competitive performance with LLMs with supervised learning models, they need Fine-Tuning on the target tasks. Table 5.1 shows the difference between the three methods of training models.

Table 5.1: Comparison of Training Methods

Type	Training	Advantages	Disadvantages
Zero-Shot	without additional training	No additional data is needed, thus saving time and effort.	Performance might be worse compared to trained models.
Fine-Tuned	Additional training	Greater accuracy and more customization of tasks	It is time-consuming and costly in terms of resources
Supervised	Full training	Good accuracy if enough data is available.	Requires resources and data.

Conclusions and Future Work

Conclusions

With the development of technology and the reliance of modern societies on digital systems, software, and networks have become an essential part of the daily infrastructure. With these systems' increasing size and complexity, two fundamental problems have emerged: software vulnerabilities and illegal intrusion into the network. Detecting software vulnerabilities is of utmost importance because their occurrence can lead to significant losses and threaten user privacy, as well as intrusion detection to protect networks from increasing attacks. Although many tools are available to analyze software and detect intrusion, they suffer from some shortcomings, especially in dealing with massive data, so more effective techniques must be used. Therefore, this research uses significant language and traditional machine learning models to find innovative solutions to detect vulnerabilities and intrusion. The most important question remains: How can these models achieve accurate and reliable results?

This thesis aims to evaluate the performance of large language models to predict software vulnerabilities and the performance of machine learning models in detecting network intrusions. We obtained results that reveal these models' capabilities.

In the first experiment, we tested the performance of large language models GPT, Llama, and CodeBERT using training and testing data of different sizes (1000,5000 and 20000 samples). The first experiment showed that the GPT model outperformed the Llama model in zero and few-shot cases on all data sizes. However, the performance of both models declined in the few-shot case. This shows the challenges of using a limited number of examples with the models, which led to their dispersion in examples.

The CodeBERT-5000 model proved to be the most efficient among the three models, as the accuracy reached 79.2% when tested on a data size of 20000. This indicates that increasing the size of the training data improves performance. Also, this result confirms the superiority of CodeBERT due to its unique design in processing software data. During the experiment to measure the inference time to predict software vulnerabilities for each model, the results showed that the GPT model was the fastest, followed by the Llama model, then the CodeBERT-5000, which took longer to process the same data size. This is due to the complex calculations that the CodeBERT performs to process the scripts.

Our main contribution to this study was to conduct three main experiments aimed at studying the performance of the models and providing deeper insights into their efficiency and reliability: (1) Consistency check to ensure the stability of the models by repeating the experiment multiple times and the result was that the models achieved high stability (2) Analyzing the ability of the GPT and Llama models to explain their predictions and understand the logic behind the predictions, where we explained how the models were able to explain behind their predictions and (3) measure the temporal performance (latency) of models to predict 1000 records and the GPT model was the fastest in performance among the three models. These experiments aimed to analyze the models in depth from multiple aspects, enhancing the results' reliability.

In the second section, we used machine learning models (LR, RF, SVM, and DT) to detect network intrusion. The results showed ideal performance for the DT model, which had an accuracy of 100%, followed by the RF model, which had an accuracy of 99.8%.

The LR and SVM models' performance was good but less than that of the tree models.

This study notes that choosing the appropriate model depends on the nature of the task. The CodeBERT model trained on sufficient data proved its superiority in software vulnerabilities, while GPT and Llama performed well in zero-shot scenarios. Traditional models, such as Decision Trees and Random Forests, showed ideal performance for intrusion detection tasks.

Finally, this study enhances our understanding of the performance of different models in cybersecurity, providing more efficient solutions for the future.

Future Works

As future work, we would also like to finetune the GPT and LLAMA models in case there are sufficient resources, training these on data that belongs strictly to the task at hand predicting software vulnerabilities, taking advantage of techniques such as LoRA (Low-Rank Adaptation) or PEFT (Parameter Efficient Fine-Tuning) to save time and resources. In addition, the performance of the Codebert models can be improved by increasing the size of the training data to improve their ability to generalize with more extensive test data. Moreover, processors such as (GPU/TPU) can be used to speed up the prediction processes and analyze the impact of using these processors on resources and speed. Finally, newer models can be used and compared with existing models to benefit from new updates in the field.

References

Ahmad, I., Basheri, M., Iqbal, M. J., & Rahim, A. (2018). Performance comparison of support vector machine, random forest, and extreme learning machine for intrusion detection. *IEEE Access*, *6*, 33789–33795.

Ahmad, Z., Shahid Khan, A., Wai Shiang, C., Abdullah, J., & Ahmad, F. (2021). Network intrusion detection system: A systematic study of machine learning and deep learning approaches. *Transactions on Emerging Telecommunications Technologies*, *32*(1), e4150.

Alrowaily, M., Alenezi, F., & Lu, Z. (2019). Effectiveness of machine learning based intrusion detection systems. *Security, Privacy, and Anonymity in Computation, Communication, and Storage: 12th International Conference, SpaCCS 2019, Atlanta, GA, USA, July 14–17, 2019, Proceedings 12*, 277–288.

Asmiddin, A. M., Nurhuda, P., & Megawati, R. (2023). Advantages and Disadvantages of Chatgpt in Science Learning: A Systematic Literature Review. *Jurnal Penelitian Pendidikan IPA*, *9*(12), 1335–1341.

Awajan, A. (2023). A novel deep learning-based intrusion detection system for IOT networks. *Computers*, *12*(2), 34.

Beltagy, I., Cohan, A., Logan IV, R., Min, S., & Singh, S. (2022). Zero-and few-shot nlp with pretrained language models. *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics: Tutorial Abstracts*, 32–37.

Bengio, Y., Simard, P., & Frasconi, P. (1994). Learning long-term dependencies with gradient descent is difficult. *IEEE Transactions on Neural Networks*, *5*(2), 157–166.

Berabi, B., Gronskiy, A., Raychev, V., Sivanrupan, G., Chibotaru, V., & Vechev, M. (2024). Deepcode AI fix: Fixing security vulnerabilities with large language models. *ArXiv Preprint ArXiv:2402.13291*.

BERT Model - NLP - GeeksforGeeks. (n.d.). Retrieved December 23, 2024, from https://www.geeksforgeeks.org/explanation-of-bert-model-nlp/

Bhatti, D. G., & Virparia, P. V. (2020). Soft computing-based intrusion detection system with reduced false positive rate. *Design and Analysis of Security Protocol for Communication*, 109–139.

Brown, T. B. (2020a). Language models are few-shot learners. *ArXiv Preprint ArXiv:2005.14165*.

Brown, T. B. (2020b). Language models are few-shot learners. *ArXiv Preprint ArXiv:2005.14165*.

Brown, T. B. (2020c). Language models are few-shot learners. *ArXiv Preprint ArXiv:2005.14165*.

Chang, K., Xu, S., Wang, C., Luo, Y., Xiao, T., & Zhu, J. (2024). Efficient Prompting Methods for Large Language Models: A Survey. *ArXiv Preprint ArXiv:2404.01077*.

Charbuty, B., & Abdulazeez, A. (2021). Classification based on decision tree algorithm for machine learning. *Journal of Applied Science and Technology Trends*, *2*(01), 20–28.

Chefer, H., Gur, S., & Wolf, L. (2021). Generic attention-model explainability for interpreting bi-modal and encoder-decoder transformers. *Proceedings of the IEEE/CVF International Conference on Computer Vision*, 397–406.

Chen, Y., Ding, Z., Alowain, L., Chen, X., & Wagner, D. (2023). Diversevul: A new vulnerable source code dataset for deep learning based vulnerability detection. *Proceedings of the 26th International Symposium on Research in Attacks, Intrusions and Defenses*, 654–668.

Cruz, D. B., Almeida, J. R., & Oliveira, J. L. (2023). Open source solutions for vulnerability assessment: A comparative analysis. *IEEE Access*, *11*, 100234–100255.

CWE - CWE-352: Cross-Site Request Forgery (CSRF) (4.16). (n.d.). Retrieved November 29, 2024, from https://cwe.mitre.org/data/definitions/352.html

Das, S. S., Serra, E., Halappanavar, M., Pothen, A., & Al-Shaer, E. (2021a). V2w-bert: A framework for effective hierarchical multiclass classification of software vulnerabilities. *2021 IEEE 8th International Conference on Data Science and Advanced Analytics (DSAA)*, 1–12.

Das, S. S., Serra, E., Halappanavar, M., Pothen, A., & Al-Shaer, E. (2021b). V2w-bert: A framework for effective hierarchical multiclass classification of software vulnerabilities. *2021 IEEE 8th International Conference on Data Science and Advanced Analytics (DSAA)*, 1–12.

Dashboard – Replicate. (n.d.). Retrieved December 31, 2024, from https://replicate.com/

Devlin, J. (2018). Bert: Pre-training of deep bidirectional transformers for language understanding. *ArXiv Preprint ArXiv:1810.04805*.

Dosovitskiy, A. (2020). An image is worth 16x16 words: Transformers for image recognition at scale. *ArXiv Preprint ArXiv:2010.11929*.

Edwards, A., & Camacho-Collados, J. (2024). Language Models for Text Classification: Is In-Context Learning Enough? *ArXiv Preprint ArXiv:2403.17661*.

Feng, Z., Guo, D., Tang, D., Duan, N., Feng, X., Gong, M., Shou, L., Qin, B., Liu, T., & Jiang, D. (2020). Codebert: A pre-trained model for programming and natural languages. *ArXiv Preprint ArXiv:2002.08155*.

Ferrag, M. A., Ndhlovu, M., Tihanyi, N., Cordeiro, L. C., Debbah, M., & Lestable, T. (2023). Revolutionizing cyber threat detection with large language models. *ArXiv Preprint ArXiv:2306.14263*.

Feurer, M., & Hutter, F. (2019). Hyperparameter optimization. *Automated Machine Learning: Methods, Systems, Challenges*, 3–33.

Gillioz, A., Casas, J., Mugellini, E., & Abou Khaled, O. (2020). Overview of the Transformer-based Models for NLP Tasks. *2020 15th Conference on Computer Science and Information Systems (FedCSIS)*, 179–183.

Giray, L. (2023). Prompt engineering with ChatGPT: a guide for academic writers. *Annals of Biomedical Engineering*, *51*(12), 2629–2633.

Gulati, A., Qin, J., Chiu, C.-C., Parmar, N., Zhang, Y., Yu, J., Han, W., Wang, S., Zhang, Z., & Wu, Y. (2020). Conformer: Convolution-augmented transformer for speech recognition. *ArXiv Preprint ArXiv:2005.08100*.

Guo, W., Fang, Y., Huang, C., Ou, H., Lin, C., & Guo, Y. (2022a). HyVulDect: a hybrid semantic vulnerability mining system based on graph neural network. *Computers & Security*, *121*, 102823.

Guo, W., Fang, Y., Huang, C., Ou, H., Lin, C., & Guo, Y. (2022b). HyVulDect: a hybrid semantic vulnerability mining system based on graph neural network. *Computers & Security*, *121*, 102823.

Hafeez, M. A., Rashid, M., Tariq, H., Abideen, Z. U., Alotaibi, S. S., & Sinky, M. H. (2021). Performance improvement of decision tree: A robust classifier using tabu search algorithm. *Applied Sciences*, *11*(15), 6728.

Hnamte, V., & Hussain, J. (2023). DCNNBiLSTM: An efficient hybrid deep learning-based intrusion detection system. *Telematics and Informatics Reports*, *10*, 100053.

Hu, Z., Zhang, J., & Ge, Y. (2021). Handling vanishing gradient problem using artificial derivative. *IEEE Access*, *9*, 22371–22377.

Javaid, A., Niyaz, Q., Sun, W., & Alam, M. (2016). A deep learning approach for network intrusion detection system. *Proceedings of the 9th EAI International Conference on Bio-Inspired Information and Communications Technologies (Formerly BIONETICS)*, 21–26.

Jin, H., Papadimitriou, G., Raghavan, K., Zuk, P., Balaprakash, P., Wang, C., Mandal, A., & Deelman, E. (2024). Large Language Models for Anomaly Detection in Computational Workflows: from Supervised Fine-Tuning to In-Context Learning. *ArXiv Preprint ArXiv:2407.17545*.

Kaplan, J., McCandlish, S., Henighan, T., Brown, T. B., Chess, B., Child, R., Gray, S., Radford, A., Wu, J., & Amodei, D. (2020). Scaling laws for neural language models. *ArXiv Preprint ArXiv:2001.08361*.

Kaur, A., & Nayyar, R. (2020). A comparative study of static code analysis tools for vulnerability detection in c/c++ and java source code. *Procedia Computer Science*, *171*, 2023–2029.

Kenton, J. D. M.-W. C., & Toutanova, L. K. (2019a). Bert: Pre-training of deep bidirectional transformers for language understanding. *Proceedings of NaacL-HLT*, *1*, 2.

Kenton, J. D. M.-W. C., & Toutanova, L. K. (2019b). Bert: Pre-training of deep bidirectional transformers for language understanding. *Proceedings of NaacL-HLT*, *1*, 2.

Kersting, N. S., Rahman, M., Vedala, S., & Wang, Y. (2024). Harmonic LLMs are Trustworthy. *ArXiv Preprint ArXiv:2404.19708*.

Kuang, F., Xu, W., & Zhang, S. (2014). A novel hybrid KPCA and SVM with GA model for intrusion detection. *Applied Soft Computing*, *18*, 178–184.

Lan, Z. (2019). Albert: A lite bert for self-supervised learning of language representations. *ArXiv Preprint ArXiv:1909.11942*.

LeCun, Y., Bengio, Y., & Hinton, G. (2015). Deep learning. *Nature*, *521*(7553), 436–444.

Lewis, M. (2019). Bart: Denoising sequence-to-sequence pre-training for natural language generation, translation, and comprehension. *ArXiv Preprint ArXiv:1910.13461*.

Limits - OpenAI API. (n.d.). Retrieved December 31, 2024, from https://platform.openai.com/settings/organization/limits

Lin, G., Wen, S., Han, Q.-L., Zhang, J., & Xiang, Y. (2020). Software vulnerability detection using deep neural networks: a survey. *Proceedings of the IEEE*, *108*(10), 1825–1848.

Lin, X., Wang, W., Li, Y., Yang, S., Feng, F., Wei, Y., & Chua, T.-S. (2024). Data-efficient Fine-tuning for LLM-based Recommendation. *Proceedings of the 47th International ACM SIGIR Conference on Research and Development in Information Retrieval*, 365–374.

Liu, Y. (2019). Roberta: A robustly optimized bert pretraining approach. *ArXiv Preprint ArXiv:1907.11692*, *364*.

Maree, M., & Shehada, W. (2024). Optimizing Curriculum Vitae Concordance: A Comparative Examination of Classical Machine Learning Algorithms and Large Language Model Architectures. *AI*, *5*(3), 1377–1390.

Mateo Tudela, F., Bermejo Higuera, J.-R., Bermejo Higuera, J., Sicilia Montalvo, J.-A., & Argyros, M. I. (2020). On combining static, dynamic and interactive analysis security testing tools to improve owasp top ten security vulnerability detection in web applications. *Applied Sciences*, *10*(24), 9119.

Medeiros, I., Neves, N., & Correia, M. (2016). DEKANT: a static analysis tool that learns to detect web application vulnerabilities. *Proceedings of the 25th International Symposium on Software Testing and Analysis*, 1–11.

Mnih, V., Heess, N., & Graves, A. (2014). Recurrent models of visual attention. *Advances in Neural Information Processing Systems*, *27*.

Mohammadi, S., Mirvaziri, H., Ghazizadeh-Ahsaee, M., & Karimipour, H. (2019). Cyber intrusion detection by combined feature selection algorithm. *Journal of Information Security and Applications*, *44*, 80–88.

Nisioti, A., Mylonas, A., Yoo, P. D., & Katos, V. (2018). From intrusion detection to attacker attribution: A comprehensive survey of unsupervised methods. *IEEE Communications Surveys & Tutorials*, *20*(4), 3369–3388.

Noever, D. (2023). Can large language models find and fix vulnerable software? *ArXiv Preprint ArXiv:2308.10345*.

NSL-KDD. (n.d.-a). Retrieved November 30, 2024, from https://www.kaggle.com/datasets/hassan06/nslkdd

NSL-KDD. (n.d.-b). Retrieved December 25, 2024, from https://www.kaggle.com/datasets/hassan06/nslkdd

Omar, M., & Shiaeles, S. (2023). VulDetect: A novel technique for detecting software vulnerabilities using Language Models. *2023 IEEE International Conference on Cyber Security and Resilience (CSR)*, 105–110.

OpenAI's GPT-3 Language Model: A Technical Overview. (n.d.). Retrieved December 22, 2024, from https://lambdalabs.com/blog/demystifying-gpt-3?srsltid=AfmBOoqk2i8AxFj0aZ-wF217QgsEwdEtB4kbklUjfCtaiBif1n6yoiXa&utm_source=chatgpt.com

Pang, G., Shen, C., Cao, L., & Hengel, A. Van Den. (2021). Deep learning for anomaly detection: A review. *ACM Computing Surveys (CSUR)*, *54*(2), 1–38.

Pearce, H., Tan, B., Ahmad, B., Karri, R., & Dolan-Gavitt, B. (2023). Examining zero-shot vulnerability repair with large language models. *2023 IEEE Symposium on Security and Privacy (SP)*, 2339–2356.

Pisner, D. A., & Schnyer, D. M. (2020). Support vector machine. In *Machine learning* (pp. 101–121). Elsevier.

Prompt engineering - OpenAI API. (n.d.). Retrieved December 28, 2024, from https://platform.openai.com/docs/guides/prompt-engineering#tactic-ask-the-model-to-adopt-a-persona

Radford, A. (2018). *Improving language understanding by generative pre-training*.

Radford, A., Wu, J., Child, R., Luan, D., Amodei, D., & Sutskever, I. (2019a). Language models are unsupervised multitask learners. *OpenAI Blog*, *1*(8), 9.

Radford, A., Wu, J., Child, R., Luan, D., Amodei, D., & Sutskever, I. (2019b). Language models are unsupervised multitask learners. *OpenAI Blog*, *1*(8), 9.

Rajamani, S. K., & Iyer, R. S. (2023). Machine Learning-Based Mobile Applications Using Python and Scikit-Learn. In *Designing and developing innovative mobile applications* (pp. 282–306). IGI Global.

Ramesh, A., Pavlov, M., Goh, G., Gray, S., Voss, C., Radford, A., Chen, M., & Sutskever, I. (2021). Zero-shot text-to-image generation. *International Conference on Machine Learning*, 8821–8831.

Ravale, U., Marathe, N., & Padiya, P. (2015). Feature selection based hybrid anomaly intrusion detection system using K means and RBF kernel function. *Procedia Computer Science*, *45*, 428–435.

Roumeliotis, K. I., & Tselikas, N. D. (2023). Chatgpt and open-ai models: A preliminary review. *Future Internet*, *15*(6), 192.

Rumelhart, D. E., Hinton, G. E., & Williams, R. J. (1986). Learning representations by back-propagating errors. *Nature*, *323*(6088), 533–536.

Sakaoglu, S. (2023). *Kartal: Web application vulnerability hunting using large language models*.

Seyyar, Y. E., Yavuz, A. G., & Ünver, H. M. (2022). An attack detection framework based on BERT and deep learning. *IEEE Access*, *10*, 68633–68644.

Shahrivar, P., & Millar, S. (2024). Detecting Web Application DAST Attacks in Large-Scale Event Data. In *Artificial Intelligence for Security: Enhancing Protection in a Changing World* (pp. 325–343). Springer.

Shaw, P., Uszkoreit, J., & Vaswani, A. (2018). Self-attention with relative position representations. *ArXiv Preprint ArXiv:1803.02155*.

Shen, Z., & Chen, S. (2020). A survey of automatic software vulnerability detection, program repair, and defect prediction techniques. *Security and Communication Networks*, *2020*(1), 8858010.

Shin, B., Park, S., Kang, K., & Ho, J. C. (2019). Self-attention based molecule representation for predicting drug-target interaction. *Machine Learning for Healthcare Conference*, 230–248.

Staudemeyer, R. C., & Morris, E. R. (2019). Understanding LSTM--a tutorial into long short-term memory recurrent neural networks. *ArXiv Preprint ArXiv:1909.09586*.

Thapa, C., Jang, S. I., Ahmed, M. E., Camtepe, S., Pieprzyk, J., & Nepal, S. (2022). Transformer-based language models for software vulnerability detection. *Proceedings of the 38th Annual Computer Security Applications Conference*, 481–496.

Valle-Perez, G., Camargo, C. Q., & Louis, A. A. (2018). Deep learning generalizes because the parameter-function map is biased towards simple functions. *ArXiv Preprint ArXiv:1805.08522*.

Van Houdt, G., Mosquera, C., & Nápoles, G. (2020). A review on the long short-term memory model. *Artificial Intelligence Review*, *53*(8), 5929–5955.

Vaswani, A. (2017). Attention is all you need. *Advances in Neural Information Processing Systems*.

wagner-group/diversevul: DiverseVul: A New Vulnerable Source Code Dataset for Deep Learning Based Vulnerability Detection (RAID 2023) https://surrealyz.github.io/files/pubs/raid23-diversevul.pdf. (n.d.). Retrieved November 27, 2024, from https://github.com/wagner-group/diversevul

walaashe/software-vulnerability-detection-using-LLM. (n.d.). Retrieved January 20, 2025, from https://github.com/walaashe/software-vulnerability-detection-using-LLM

Wang, B., Yue, X., & Sun, H. (2023). Can ChatGPT defend its belief in truth? evaluating LLM reasoning via debate. *ArXiv Preprint ArXiv:2305.13160*.

Wang, C., Li, Z., Pena, Y., Gao, S., Chen, S., Wang, S., Gao, C., & Lyu, M. R. (2023). Reef: A framework for collecting real-world vulnerabilities and fixes. *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 1952–1962.

Wang, H., Gu, J., & Wang, S. (2017). An effective intrusion detection framework based on SVM with feature augmentation. *Knowledge-Based Systems*, *136*, 130–139.

Wang, W., Wei, F., Dong, L., Bao, H., Yang, N., & Zhou, M. (2020). Minilm: Deep self-attention distillation for task-agnostic compression of pre-trained transformers. *Advances in Neural Information Processing Systems*, *33*, 5776–5788.

Weber, M., Wang, H., Qiao, S., Xie, J., Collins, M. D., Zhu, Y., Yuan, L., Kim, D., Yu, Q., & Cremers, D. (2021). Deeplab2: A tensorflow library for deep labeling. *ArXiv Preprint ArXiv:2106.09748*.

Wei, J., Bosma, M., Zhao, V. Y., Guu, K., Yu, A. W., Lester, B., Du, N., Dai, A. M., & Le, Q. V. (2021). Finetuned language models are zero-shot learners. *ArXiv Preprint ArXiv:2109.01652*.

White, J., Fu, Q., Hays, S., Sandborn, M., Olea, C., Gilbert, H., Elnashar, A., Spencer-Smith, J., & Schmidt, D. C. (2023). A prompt pattern catalog to enhance prompt engineering with chatgpt. *ArXiv Preprint ArXiv:2302.11382*.

Wisanwanichthan, T., & Thammawichai, M. (2021). A double-layered hybrid approach for network intrusion detection system using combined naive bayes and SVM. *Ieee Access*, *9*, 138432–138450.

Wolf, T., Debut, L., Sanh, V., Chaumond, J., Delangue, C., Moi, A., Cistac, P., Rault, T., Louf, R., & Funtowicz, M. (2020). Transformers: State-of-the-art natural language processing. *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, 38–45.

Xian, Y., Lampert, C. H., Schiele, B., & Akata, Z. (2018). Zero-shot learning—a comprehensive evaluation of the good, the bad and the ugly. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, *41*(9), 2251–2265.

Xu, C., Yuan, F., & Chen, S. (2022). [Retracted] BJBN: BERT-JOIN-BiLSTM Networks for Medical Auxiliary Diagnostic. *Journal of Healthcare Engineering*, *2022*(1), 3496810.

Xu, F. F., Alon, U., Neubig, G., & Hellendoorn, V. J. (2022). A systematic evaluation of large language models of code. *Proceedings of the 6th ACM SIGPLAN International Symposium on Machine Programming*, 1–10.

Xu, H., Kim, Y. J., Sharaf, A., & Awadalla, H. H. (2023). A paradigm shift in machine translation: Boosting translation performance of large language models. *ArXiv Preprint ArXiv:2309.11674*.

Yong, G., Jeon, K., Gil, D., & Lee, G. (2023). Prompt engineering for zero-shot and few-shot defect detection and classification using a visual-language pretrained model. *Computer-Aided Civil and Infrastructure Engineering*, *38*(11), 1536–1554.

Zaidi, A., & Al Luhayb, A. S. M. (2023). Two statistical approaches to justify the use of the logistic function in binary logistic regression. *Mathematical Problems in Engineering*, *2023*(1), 5525675.

Zhang, L., & Suganthan, P. N. (2014). Random forests with ensemble of feature spaces. *Pattern Recognition*, *47*(10), 3429–3437.

Zhang, R., Han, J., Liu, C., Zhou, A., Lu, P., Qiao, Y., Li, H., & Gao, P. (2024). LLaMA-adapter: Efficient fine-tuning of large language models with zero-initialized attention. *The Twelfth International Conference on Learning Representations*.

Zhang, Z., Zhang, A., Li, M., & Smola, A. (2022). Automatic chain of thought prompting in large language models. *ArXiv Preprint ArXiv:2210.03493*.

إطار عمل شامل للتنبؤ بثغرات البرامج باستخدام النماذج اللغوية الكبيرة.

ولاء يوسف عطا شحادة

د. حذيفة أشقر

د. عمر درويش

د. محمود عبيد

د. حامد عبد الحق

ملخص

مع تطور التكنولوجيا واعتماد المجتمعات الحديثة على الأنظمة الرقمية، أصبحت البرمجيات والشبكات جزءًا أساسيًا من البنية التحتية اليومية. ومع تزايد حجم وتعقيد هذه الأنظمة، ظهرت مشكلتان أساسيتان: نقاط الضعف في البرمجيات والاختراق غير القانوني للشبكة. ويعد اكتشاف نقاط الضعف في البرمجيات أمرًا بالغ الأهمية لأن حدوثها يمكن أن يؤدي إلى خسائر كبيرة ويهدد خصوصية المستخدم، بالإضافة إلى اكتشاف الاختراق لحماية الشبكات من الهجمات المتزايدة. وعلى الرغم من وجود العديد من الأدوات المتاحة لتحليل البرمجيات والكشف عن الاختراق، إلا أنها تعاني من بعض النواقص، وخاصة في التعامل مع البيانات الضخمة، لذلك يجب استخدام تقنيات أكثر فعالية. لذلك، في هذا البحث، يتم استخدام نماذج اللغة الكبيرة ونماذج التعلم الآلي التقليدية لإيجاد حلول مبتكرة للكشف عن نقاط الضعف والاختراق. ويبقى السؤال الأكثر أهمية: كيف يمكن استخدام هذه النماذج لتحقيق نتائج دقيقة وموثوقة؟

تهدف هذه الأطروحة إلى تقييم أداء نماذج اللغة الكبيرة للتنبؤ بثغرات البرمجيات بالإضافة إلى تقييم أداء نماذج التعلم الآلي في الكشف عن اختراقات الشبكة. وقد حصلنا على نتائج ذات رؤى واضحة حول قدرات هذه النماذج.

في التجربة الأولى، قمنا باختبار أداء نماذج اللغة الكبيرة جي بي تي، لاما وكود–بيرت باستخدام بيانات تدريب واختبار بأحجام مختلفة (1000 و5000 و20000 عينة). وأظهرت نتائج التجربة الأولى أن نموذج جي بي تي تفوق على نموذج لاما في حالة اللقطة الصفرية واللقطة القليلة على جميع أحجام البيانات. ومع ذلك، انخفض أداء كلا النموذجين في حالة اللقطة القليلة. وهذا يوضح تحديات استخدام عدد محدود من الأمثلة مع النماذج، مما أدى إلى تشتتها في الأمثلة.

أثبت نموذج كود–بيرت5000 أنه الأكثر كفاءة بين النماذج الثلاثة، حيث وصلت الدقة إلى 79.2٪ عند اختباره على حجم بيانات 20000. وهذا يشير إلى أن زيادة حجم بيانات التدريب يحسن الأداء. كما تؤكد هذه النتيجة تفوق كود–بيرت بسبب تصميمه الخاص في معالجة بيانات البرمجيات. خلال التجربة لقياس زمن الاستدلال للتنبؤ بالثغرات البرمجية لكل نموذج، أظهرت النتائج أن نموذج جي بي تي كان الأسرع، يليه نموذج لاما، ثم كود–بيرت5000 الذي استغرق وقتًا أطول لمعالجة نفس حجم البيانات. وذلك بسبب الحسابات المعقدة التي يقوم بها كود–بيرت لمعالجة البرامج النصية.

كانت مساهمتنا الرئيسية في هذه الدراسة هي إجراء 3 تجارب رئيسية تهدف إلى دراسة أداء النماذج وتوفير رؤى أعمق حول كفاءتها وموثوقيتها: (1) فحص الاتساق لضمان استقرار النماذج من خلال تكرار التجربة عدة مرات وكانت النتيجة أن النماذج حققت استقرارًا عاليًا (2) تحليل قدرة نموذجي جي بي تي ولاما على تفسير تنبؤاتهما وفهم المنطق وراء التنبؤات، حيث أوضحنا كيف تمكنت النماذج من التفسير وراء تنبؤاتها و(3) قياس الأداء الزمني (الزمن الكامن) للنماذج للتنبؤ بـ 1000 سجل وكان نموذج جي بي تي هو الأسرع في الأداء بين النماذج الثلاثة. هدفت هذه التجارب إلى تحليل النماذج بعمق من جوانب متعددة، مما يعزز من موثوقية النتائج.

في القسم الثاني، استخدمنا نماذج التعلم الآلي (الانحدار اللوجستي، والغابة العشوائية، وآلة الدعم المتجه، وشجرة القرار) للكشف عن التطفل على الشبكة. وأظهرت النتائج أداءً مثاليًا لنموذج شجرة القرار بدقة 100%، يليه نموذج الغابة العشوائية بدقة 99.8%.

أما بالنسبة لنماذج الانحدار اللوجستي وآلة الدعم المتجه، فقد كان أداءهما جيدًا ولكن أقل من نماذج الشجرة.

من خلال هذه الدراسة، نلاحظ أن اختيار النموذج المناسب يعتمد على طبيعة المهمة، ففي مجال ثغرات البرمجيات، أثبت نموذج كود–بيرت المدرب على كمية كافية من البيانات تفوقه، بينما حقق جي بي تي ولاما أداءً جيدًا في حالة الطلقات الصفرية. أما بالنسبة لمهام الكشف عن التطفل، فقد أظهرت النماذج التقليدية شجرة القرار والغابة العشوائية أداءً مثاليًا.

وأخيرًا، تعزز هذه الدراسة فهم أداء النماذج المختلفة في مجال الأمن السيبراني، مما يوفر حلولاً أكثر كفاءة للمستقبل.

الكلمات المفتاحية: ثغرات البرمجيات، نماذج اللغة الكبيرة، اختراق الشبكة، التعلم الآلي.