IMPROVING SOFTWARE SECURITY VIA THE USE OF PRE-TRAINED CODE LARGE LANGUAGE MODELS IN VULNERABILITIES DETECTION

OLANREWAJU EBUN, OLADOKUN
Master of Science in Information Technology
National Open University of Nigeria, 2019

A thesis submitted
in partial fulfilment of the requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

Department of Mathematics and Computer Science
University of Lethbridge
LETHBRIDGE, ALBERTA, CANADA

IMPROVING SOFTWARE SECURITY VIA THE USE OF PRE-TRAINED CODE LARGE LANGUAGE MODELS IN VULNERABILITIES DETECTION

OLANREWAJU EBUN, OLADOKUN

Date of Defence: August 14, 2025

Dr. Jackie Rice Thesis Supervisor	Professor	Ph.D.
Dr. John Anvik Thesis Examination Committee Member	Assoc. Professor	Ph.D.
Dr. Andrew Fiori Thesis Examination Committee Member	Assoc. Professor	Ph.D.
Dr. Wendy Osborn Chair of the Thesis Examination	Assoc. Professor	Ph.D.

Dedication

This thesis is specially dedicated to the almighty GOD who gave mankind, the most important, efficient and intelligent biological machine - the brain, through which scientists, engineers and philosophers across thousands of generations have progressively contributed to the development of various civilizations.

Abstract

The ubiquity and dependence on software systems by people, businesses and organizations in the 21st century has resulted in an upsurge in cyber-attacks in recent times. These attacks are generally characterized by different levels of sophistication, occurrence and complexity that makes it difficult for conventional cybersecurity approaches to adequately mitigate them. Although cybercriminals, including hackers, are usually blamed for most cyber-attacks, the fundamental cause is, however, typically associated with the inherent security weaknesses. These weaknesses are the loopholes in the software source code programs through which hackers exploit systems in ways that constitute cybercrimes. Hence, in recent years, various AI-based approaches have been proposed or explored in studies to address this challenge. These innovative methods are aimed at complementing the conventional approaches (including awareness training, malware scanning, and manual code inspection) that have been adopted over the years. In our research, we experimented with the use of emerging AI models called Large Language Models (LLMs) in the detection of vulnerabilities in a software system. As a case study, I used Android software since current statistics reveal that over 71 percent of all mobile phones across the world are based on this software. In my experiment, I utilized LVDAndro: a recently released open-source Android vulnerabilities-dataset for training my selected LLMs, which were CodeBERT and GraphCodeBERT. The goal was to detect vulnerabilities in Android code bases. Overall, my approach achieved better performance (0.99 Accuracy, 0.99 F1) in Android vulnerability detection compared to the classical Machine Learning (ML) (0.94 Accuracy, 0.94 F1) model used in the previous study.

Acknowledgments

Let me start by greatly appreciating Dr. Jackie Rice for the golden opportunity of studying under her supervision. Based on my observation, graduate students at the University of Lethbridge crave the special privilege of studying with her. Her decision to accept me into her renowned research group came at a critical moment in my life when I decided to briefly leave the industry for academia. Furthermore, I am also grateful to all our professors, especially members of the ISEDAM research lab for exposing aspiring researchers like me to the world of critical thinking, problem analysis, paper writing and scientific collaboration. Furthermore, I am immensely grateful to my mum - Mrs. Caroline Oladokun for her support and prayers throughout my study. I am also very grateful to my late dad - Mr. Micheal Oladokun for investing so much on my academic success from childhood. Finally, this acknowledgment will not be complete without greatly appreciating the impeccable support from my better half and lovely wife, Elizabeth Oladokun, our two daughters - Christlove and Christlife, and son - Christlead. Leaving them for two years to study abroad was one of the most challenging periods of our life but we are grateful to GOD for his mercies, protection, guidance and blessings.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Since the advent of the Information Age in the 20th century to the current era, the ubiquity and dependence on software systems has changed the world [12]. Almost all critical infrastructure powering every sector of the economy including transportation, commerce, banking and energy are managed by modern software applications [13]. The recent Cloudstrike IT outage which affected the daily operations of thousands of businesses, organizations and governments across the world underscores the critical role software plays in driving the economy of the modern world [14]. Furthermore, due to their widespread usage, software systems have become the targets of cyber-attacks from various threat actors including state-sponsored hackers [15]. Consequently, people and businesses continue to incur both financial and non-monetary losses including ransom payments, regulatory fines and reputational damage [16]. Several factors are frequently attributed to the occurrence and prevalence of modern cyber-attacks. One of the leading factors is the exploitation of vulnerabilities in software systems [17].

Vulnerabilities are inherent weaknesses in the software source code that compromise its security [18]. Weaknesses are caused by unintended errors or flaws in the design and development of a piece of software. If a weakness could be exploited by hackers, it is termed a vulnerability. As exploitation approaches become more sophisticated, conventional approaches for mitigating vulnerabilities are no longer adequate. For instance, Figure 1.1 shows how common vulnerabilities have increased progressively over a period of 10 years [7].

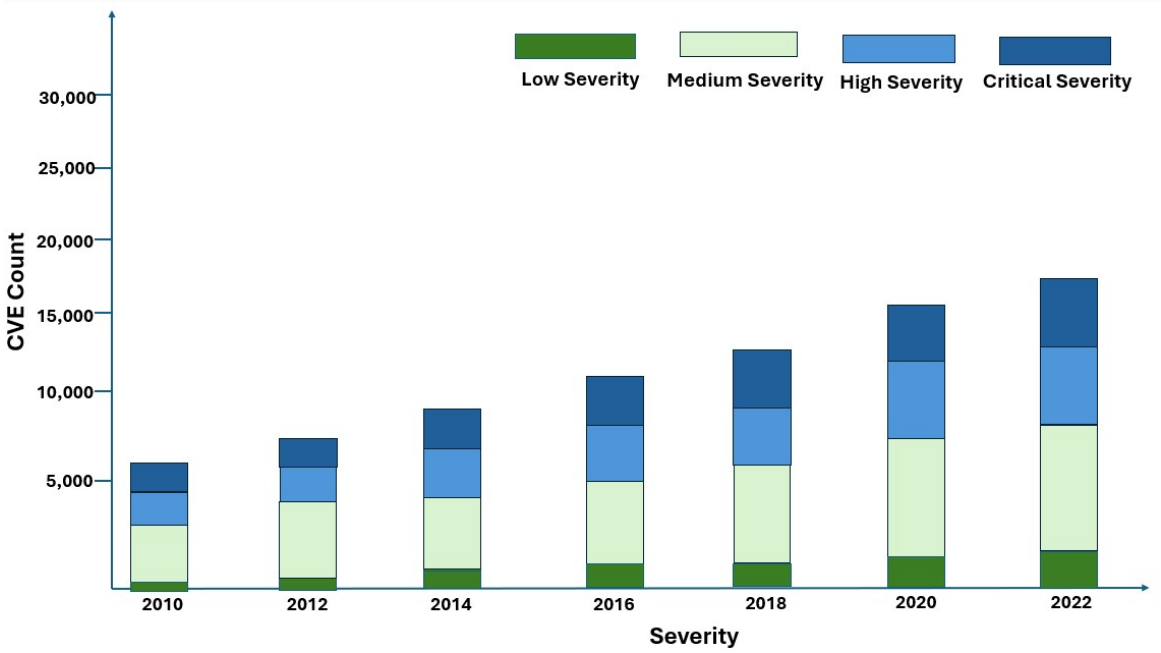

Figure 1.1: Increasing vulnerabilities over a period of 12 years [7].

1.1 Motivation

Recent trends in industry and academia include the accelerated adoption of various Artificial Intelligence (AI) approaches in complementing existing strategies for addressing software vulnerabilities [19, 20, 21]. The classical AI approach typically involves the use of Machine Learning (ML) and Deep Learning (DL) [20, 21]. However, the outcomes of previous studies show that further work may be required to improve the performance of existing AI-based methods [5, 19]. For instance, the source code datasets used in some studies are largely synthetic and therefore do not reflect real-life software systems. An example is the Software Assurance Reference Dataset (SARD) [22], which is commonly used in many AI-based vulnerability detection studies. In certain cases where real-life datasets are available, such as Big-Vul [23], they contain vulnerabilities pertaining only to C/C++ source code and are therefore not optimal for training models for other software systems written in other programming languages. Although LLMs could generalize to other languages that weren't used in their pre-training, their understanding of such languages may be limited due to nuances specific to those languages. Hence, for optimal results especially in tasks such as

vulnerabilities detection, fine-tuning LLMs on specific languages is better. In another study, [19] observed that vulnerabilities detection results on real datasets such as Big-Vul are not as good as synthetic datasets such as SARD and therefore recommended that future work should use more real vulnerabilities datasets. We hypothesize that one of the reasons for the poor performance in their approach could be due to the use of only models which were originally designed for natural language and not programming languages. Similarly, in other cases, the models adopted for developing vulnerability detection tools are still sub-optimal despite their promising performance when used on some realistic datasets such as the recently released Android-based datasets called LVDAndro [5]. For instance, in their experiments [5] utilized three traditional ML models to develop vulnerability detection tools using the LVDAndro datasets. While their results were acceptable, our hypothesis was that Large Language Models (LLMs) specifically optimized for programming languages will perform better than [5]'s classical ML approach when trained on the same real-life source code datasets. This is because LLMs are specifically designed for performing language or text-based processing tasks such as text comprehension, generation, classification, summarization and sentiment analysis, just to mention a few. Furthermore, LLMs are pre-trained on vast amounts of text-based data from various sources including the Internet, books and articles. In addition to these, all LLMs contain a special feature in their architecture called transformers which enables them to better understand the semantics and context in natural language unlike classical ML models that lack this capability. Since both natural language and programming language are text-based, the capabilities of LLMs in the former can be practically extended to the latter. Another reason is that vulnerability detection in software source code is related to text classification in human natural language. Thus, the potential improvements in software vulnerabilities detection through the adoption of code-based LLMs is the major motivation for this work. Our work will involve the selection and use of appropriate models and real-life source code datasets. In this study, LLMs that are pre-trained in programming languages are referred to as Code Large Language Models (CLLMs). This study is aimed

at exploring how these models can be used to detect vulnerabilities in source code. Our implementation uses the Android Operating System (OS) as a case study. The following outlines our reasons for choosing vulnerabilities within the Android OS as our dataset:

- First, current statistics reveal that Android is the most widely used mobile OS in the world [24].
- Android OS is used across a broad range of devices including smartphones and embedded systems [25].
- Various Android applications which are released to end users everyday are known to have vulnerabilities that could be exploited by hackers and other cyber criminals [5].
- Finally, the availability of the recently released real life Android-based dataset called LVDAndro allows for the training of LLM models to detect vulnerabilities in this software [5].

1.2 Contributions

Overall, this study aims at improving software security via the use of code LLMs for detection of vulnerabilities in source codes. By using these LLM-based approaches, we intend to achieve the following Research Objectives (RO):

- RO1: Determine the effectiveness of LLM approach in software vulnerability detection using Android-based source code as a case study.
- RO2: Determine the extent to which emerging LLM approaches are better than the conventional ML/DL approaches.
- RO3: Determine the extent to which code-based LLMs are better than natural language-based LLMs in vulnerability detection.

- RO4: Ascertain whether graph-based code LLMs are better than pure code-based LLMs in vulnerability detection.
- RO5: Utilize the outcome of the study to recommend how code-based LLMs can be used to improve software security through the detection of vulnerabilities in source code.

1.3 Organization of Thesis

- In Chapter 1, we discussed the motivation and hypothesis of this research.
- Chapter 2 focuses on a literature review of core concepts upon which this study is based. This includes the overview of various language models, transformer architecture, steps for language model processing and software security vulnerabilities.
- In Chapter 3, we discuss the methodology adopted for our work. This includes implementation design, compute infrastructure, data acquisition, data preparation and data processing. We also discussed model selection, fine-tuning and evaluation.
- Chapter 4 describes the experiments that were conducted and their associated results.
- Chapter 5 offers some discussion about the results presented in Chapter 4.
- We conclude this thesis by summarizing the work done and offering recommendations for future study.

Chapter 2

Background and Literature Review

2.1 Language Models

Language models (LMs) are machine learning and deep learning models specially designed to understand and process human language [**?**]. These models use various techniques for processing textual information. Language models are the foundation of all Natural Language Processing (NLP) models, including Large Language Models (LLMs). They are also used to develop various AI applications such as chat bots, language translators and text summarizers. The recent success of popular tools such as ChatGPT, Deepseek and other similar chat bots are attributed to language models. Historically, these models evolved over the years from the area of AI called Natural Language Processing (NLP) [**?**].

2.2 How Language Models Evolved

In 1913, Andrey Markov pioneered the concept of probability distribution of words in a text by examining the distribution of vowels in a book called Eugene Onegin [26]. This basic idea is the foundation of all n-gram language models which are based on the assumption (Markov assumption) that the probability of a word depends only on the previous word. Claude Shannon expanded this statistical technique in 1948 by demonstrating how to quantify information and randomness (entropy) [27]. This statistical principle is the foundation of other popular models like Bag of Words (BOW) and Term Frequency - Inverse Document Frequency (TF-IDF) that are used in classical NLP for processing texts. Another major milestone in language models was the development of rule-based models which were

pioneered with the introduction of the first chat bot called ELIZA in 1967 [28]. In the 1980s, Recurrent Neural Networks (RNNs) [8, 29] emerged as another major improvement in the development of language models. RNNs are designed to process sequential data (speech, text) by using recurrent connections to form a memory in a hidden state. However, due to the limitations of RNN, an improved model called Long Short-Term Memory (LSTM) [8, 29] was introduced in 1997. LSTMs require high computational costs, high training time and large data requirements for training [8]. The next major leap in language models was the introduction of vector space models such as Word2Vec which was developed by Google in 2013 [30]. An improved model called Gated Recurrent Units (GRU) was developed in [31]. It performed faster and had a less complex architecture compared to the LSTM. These gates also help GRU capture long range dependencies (improved contextual understanding) in text sequences unlike RNN models [31]. However, the development of the first transformer model by a team of researchers at Google in 2017 [32] brought the most significant improvement in the evolution of all language models. Transformer models are designed based on the concept of the attention mechanism and encoder-decoder architecture [32]. The encoder component of transformers is the foundation of all Bidirectional Encoder Representation from Transformers (BERT) models [10] while the decoder component is the basis of all Generative Pre-trained Transformer (GPT) models [33].

2.3 Type of Language Models

There are various types of language models. These include N-gram, rule-based, vector space, neural network, sequence to sequence, transformer-based, large language models and code language models [8].

2.3.1 N-gram Language Models

N-gram language models are models in which probabilities are assigned to upcoming words or word sequences in order to predict them [34]. These include bigrams and trigrams.

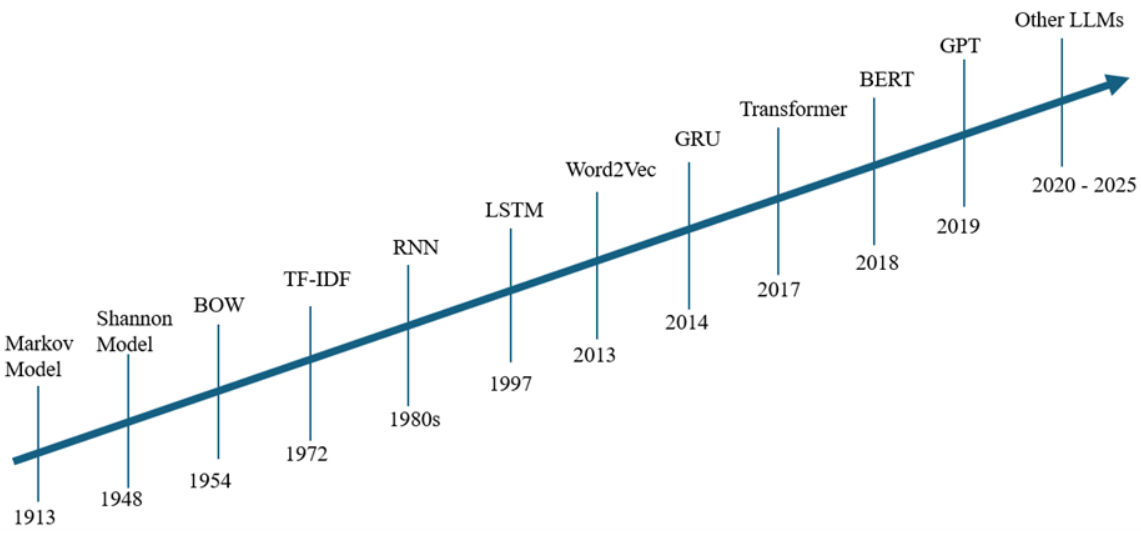

Figure 2.1: Chronological evolution of Language Models [8].

They are typically based on Markov's assumption which states that the probability of the next event in a sequence (e.g. next word in a sentence) depends only on the current event (word) but not past events (words) [35]. In NLP, this can be expressed as follows:

$$s = (w_1, w_2 \dots w_N) \qquad Sentence\ of\ words$$
$$P(s) = P(w_1, w_2 \dots w_N) \qquad Probability\ distribution\ of\ sentence\ of\ words$$
$$P(w) = P(w_t | w_1 \dots w_{t-1}) \qquad Probability\ of\ next\ word$$
$$w_t = \ Discrete\ words\ in\ a\ sentence\ sequence$$

2.3.2 Rule-based Language Models

Rule-based language models are based on handcrafted words or sentence patterns which have been pre-described in scripts provided to the system. Their performance is limited to the scripts called the knowledge base [8, 36]. Thus, they are domain specific. Examples include PARRY, ALICE and ELIZA [8, 28].

2.3.3 Vector Space Language Models

Vector space models are based on the concept of representing words in space dimensions called vectors or embeddings. In these models, the closer the words are together in the vector space, the closer their meanings and vice versa. Examples include Word2Vec and GloVe [8].

- **Word2Vec**: is a vector space model developed in 2013 by researchers at Google as a method to create vector representations of words called embeddings [26]. The model uses the cosine similarity between words to capture some level of semantic similarity. Word2Vec uses two types of architectures to produce word embedding. The first is Continuous Bag of Words (CBOW) while the second is Continuous Skip-Gram Model [37]. CBOW predicts the target word based on its surrounding context while Continuous Skip-Gram predicts the surrounding context words based on the target word. Word2Vec uses these two mechanisms as strategies for converting ordinary texts to numeric vectorized forms that are more meaningful.

- **GloVe**: is a vector space model developed at Stanford University in 2014 to create vector representations of words. Unlike Word2Vec that uses predictions for generating word representations, GloVe uses co-occurence counts to produce vector representations of the words in a corpus [38]. Co-occurrence count refers to the number of times two words appear together within a particular context in a corpus. GloVe also uses unsupervised learning of the word representations [38].

2.3.4 Sequence to Sequence Language Models

The development of the Sequence to Sequence (Seq2Seq) language models was required by the need to translate words and sentences from one language to another [39]. Seq2Seq utilizes an encoder-decoder architecture which are designed using earlier models such as LSTM and GRU. However, this architecture is different from the encoder-decoder architecture of transformer models which incorporates an improved technique called an attention mechanism as explained in Section 2.4. The LSTM/GRU encoder is responsible

for reading the input sequence and generating its equivalent vectors while the LSTM/GRU decoder converts these vectors to the corresponding output sequence [39].

2.3.5 Recurrent Neural Network Language Models

Recurrent Neural Network (RNN) language models are implemented using a type of Neural Network (NN) called Recurrent Neural Network (RNN) [8]. A NN consists of an interconnected network of nodes called neurons which work together to process information [40]. An RNN consists of layers of NNs that are connected in a recurring manner to enable them to process sequential time series information. However, due to its architectural (memory) limitations, these models are not effective in processing long term interdependence between words in sentences [8].

2.3.6 Transformer-based Language Models

The Transformer model was introduced through a research paper called "Attention is All You Need" in 2017 [9]. The model contains key design components including self-attention, encoder-decoder stacks, word embedding and positional embedding. This innovative design gives it the ability to outperform the Seq2Seq model by processing token embeddings in parallel. Most of the current popular pre-trained language models are based on a transformer architecture [9]. We describe this in further detail in the following section.

2.4 Transformer Architecture

Unlike the Seq2Seq models, a transformer model does not utilize LSTM or GRU in its encoder-decoder architecture. Rather, it consists of an encoder stack, a decoder stack, a positional encoding layer, a feed forward layer, an input embedding layer, a linear layer, and a softmax layer [9]. Figure 2.2 shows the basic architecture of a transformer model.

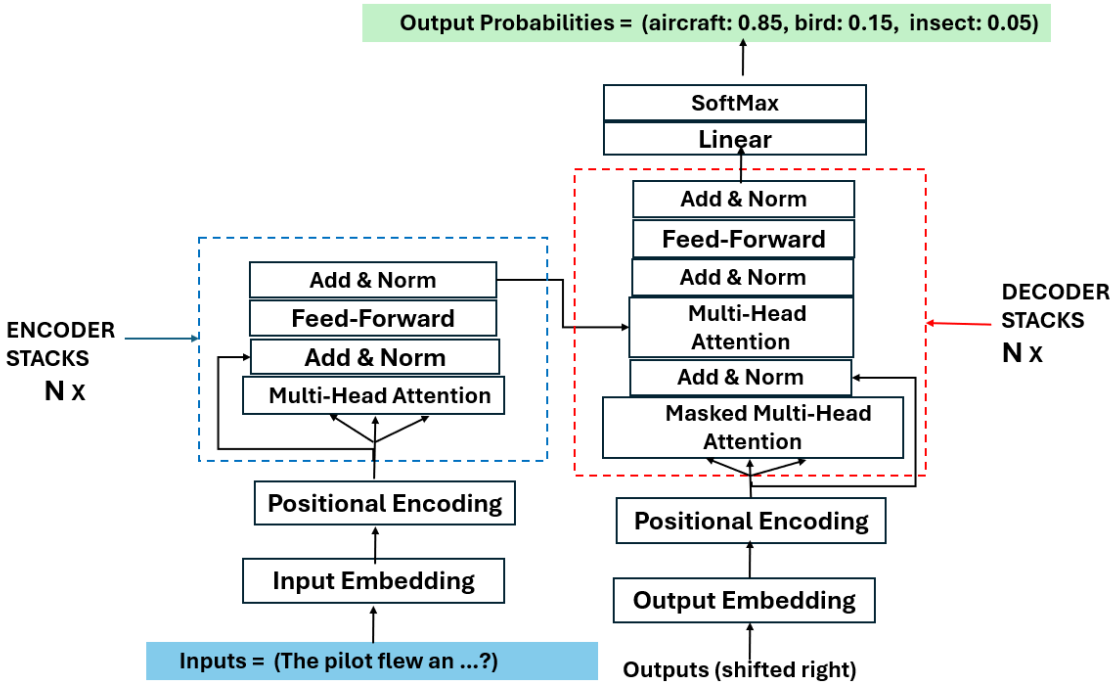

Figure 2.2: Transformer architecture [9].

2.4.1 Encoder Stack

The encoder stack processes any input sequence by breaking it down into meaningful representations. This is achieved by creating semantic or contextualized representations of the tokens in which each token also contains information from other tokens through a process called self-attention [9]. Self-attention enables a transformer to determine the relevance of different parts of an input sequence whenever it is processing a specific part of the sequence. Self-attention is implemented in the encoder stack using a layer called multi-head attention. In addition to the multi-head attention layer, the encoder stack consists of two additional layers called the feed-forward layer and the add & nom layer.

- **Multi-Head Self Attention Layer**: This layer captures the relationships and interdependencies between words in any sequence [9]. It achieves this through the use of three important components called the query (Q), key (K) and value (V) vector sets. These components collectively help the transformer to focus on the most relevant parts of any input sequence during text processing. The query vector for any given

element (word or phrase) represents what the element is searching for (focused on) in the entire sentence. The key acts as a label to represent information that can be pulled from other elements. The value is the actual information that each element contains based on their attention scores whenever there is a match between query and value [9].

- **Feed-forward Layer:** This is a Neural Network (NN) layer that is connected to the output of the attention layer. It performs non-linear transformations for learning complex patterns in textual data and position-wise processing so that each token can be processed independently [9]. The NN layers are trained on textual datasets using multiple parameters. Parameters are the internal variables that the model uses to achieve learning during training. Parameters include weights and biases, and are adjusted iteratively during training to minimize loss and control the model's behaviour [41]. Weights and biases are usually initiated with random values at the commencement of training. Weights are numerical values, which help in determining the strength of the neural network connections. In other words, weights determine the influence an input has on the neuron's output. A neuron is a computational unit in each layer of a neural network that accepts inputs and returns an output [42]. Weights are automatically adjusted during training to minimize loss [42]. Biases are numerical constants that are added to each neuron to provide additional flexibility to the model during training [42]. Collectively, both weight and bias help in controlling the neuron's output. The relationship between weight and bias required for a model's prediction can be expressed as follows:

  $$Y = w * X + b$$

  where Y is the predicted value, X is the input, w is weight and b is bias. Weights corresponds to the gradient while bias represents the intercept.

- **Add & Norm Layer:** This layer helps to combine residual connections and layer normalization in order to achieve stability during training [9].

2.4.2 Decoder Stack

The decoder stack is responsible for receiving the representations and converting them into output sequences. Similar to the encoder stack, the decoding stack also contains a multi-head self attention layer, a feed-forward layer, and an Add & Norm layer. However, the decoder stack also contains another layer called the masked multi-head attention layer [9].

- **Masked Multi-Head Attention Layer** is a multi-head attention layer with a masked position. The masking is included to prevent positions from focusing on subsequent positions [9]. Using this strategy, predictions for a position depend only on the known outputs before that position.

2.4.3 Softmax Layer

The softmax layer is the final output layer. Its purpose is to convert the raw attention scores (logits) to a probability distribution by using the softmax function [9]. Softmax is a mathematical function that converts a vector of real numbers into a probability distribution where every output lies between 0 and 1. For instance, supposing the input to the transformer is an incomplete sentence such as the following: ”The pilot flew an .....” . In order for the transformer to predict the best word that completes the sentence, it will first convert the entire sentence to contextualized embeddings (vectorized tokens with semantic information).

After processing through the transformer’s layers, the contextualized embedding is then converted to logits at the linear layer. At the output, the softmax layer will convert the logits to probabilities ranging fron 0 to 1. These could be potential words such as (aircraft : 0.85, bird: 0.10, insect: 0.05). The highest probability will be selected by the transformer as the best output to complete the sentence. In this case, this would be 0.85.

2.5 Types of Transformer-based Language Models

Transformer-based language models can be classified into three general types based on the layer of the transformer architecture (encoder, decoder or combination of both) that is primarily used for their operation [9]. The following sections describe the three types:

2.5.1 Encoder Transformer Models

Encoder-based language models are designed to use only the encoder component of transformers [43]. These models are optimized for Natural Language Understanding (NLU) tasks such as text classification and sentiment analysis. The most popular encoder-based model is Bidirectional Encoder Representations from Transformers (BERT) [10].

- **BERT**: This encoder model was designed by Google in 2018 to improve the understanding of contextual information in unlabeled texts. BERT achieves this by predicting text that may come either before the text or after the text. This is the reason why it is bidirectional [10]. Figure 2.3 shows the official architecture of BERT during pre-training. Internally, it contains transformer features such as embedding

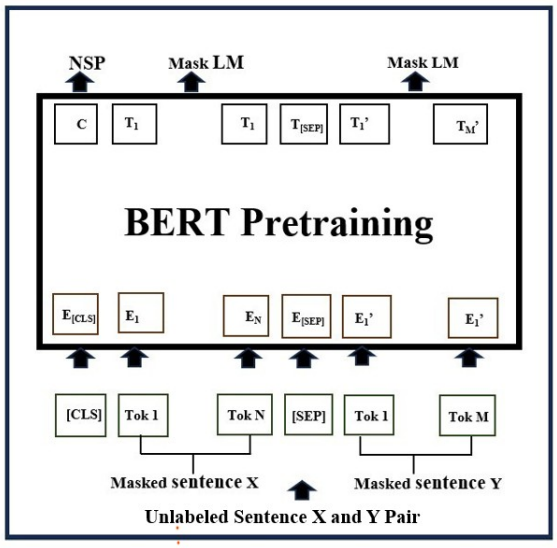

Figure 2.3: Architecture of BERT [10].

and encoder stacks. It also has a tokenizer which converts human language text to

token sequences. The embedding component converts the token sequences to vector array representations. The model uses the same architecture and parameters for both the pre-training and fine-tuning. Pre-training and fine-tuning are explained in detail in sub-sections 2.7.4 and 2.7.5. Using text data from Wikipedia and BookCorpus, BERT is pre-trained through two main techniques called Masked Language Modelling (MLM) and Next Sentence Prediction (NSP) [10]. MLM is used to randomly mask and predict some percentage of input token. Through MLM, BERT is able to achieve word level comprehension. A classification symbol (CLS) is added in front of every input sample and a separator token (SEP) is added to separate the questions and answers. Masked language modelling is a technique used in Natural Language Processing (NLP) where a model is trained to predict a masked (missing) word in a given sentence based on the surrounding context [10]. In MLM, the model learns a bidirectional context. This is because words on the left and right sides of the masked token are used to derive the context. In this way, the relationship between two sentences can be captured. Next Sentence Prediction (NSP) is used to predict whether two sentences follow each other. Through NSP, BERT is able to achieve sentence level comprehension. BERT automatically carries these two processing tasks internally using its multiple transformers and neural networks. In Figure 2.3, a two sentence pair X and Y are fed into the BERT model. Some tokens (Tok1...TokN) are randomly masked. The entire sequences are converted to embeddings E_1 ... E_M that are passed through BERT's multiple internal transformer layers and neural networks. BERT then uses MLM and NSP algorithms to predict those masked tokens and next sentences at the output.

2.5.2 Decoder Transformer Models

Decoder-based models are designed using only the decoder component of transformers. Decoder models consist of layers of decoders optimized for Natural Language Generation (NLG) tasks. The most popular decoder-based model is Generative Pre-trained Transformer

(GPT) [33]. Decoder models are autoregressive because they can predict the next word in a sentence based on the previously generated word in the sequence. Decoder models are also unidirectional because they process text in one direction only. Decoder models such as GPT are the reasons for the rapid growth of generative AI technologies.

- **Generative Pre-trained Transformer**: The first decoder-based model was developed by OpenAI in 2018 and was called GPT-1 [1]. This decoder-based transformer model uses self-supervised learning to train on large amounts of textual data in order to produce various natural language-based outputs [33]. Subsequently, other GPT models have been released yearly until 2023 [1] as shown in Table 2.1.

Table 2.1: GPT-based models [1].

GPT Model	Parameters	Year
GPT-1	117M	2018
GPT-2	1.5B	2019
GPT-3	175B	2020
GPT-3.5	1.3B, 6B, 175B	2022
GPT-4	100T	2023

2.5.3 Encoder-Decoder Models

Encoder-decoder-based models are designed by incorporating both encoders and decoders in their architecture. They are optimized for text processing tasks that involve understanding sequences of input texts as well as the generation of sequences of output texts. They are good at capturing interrelationships between the elements of input and output sequences. These include tasks such as text translation and text summarization. The most popular encoder-decoder model is Bidirectional and Auto-Regressive Transformer (BART) [44].

- **Bidirectional and Auto-Regressive Transformer (BART)**: This encoder-decoder transformer model combines the bidirectional encoder method of BERT and the

unidirectional technique of GPT. Hence it is capable of performing both natural language comprehension like BERT and language (text) generation like GPT [45].

Table 2.2: Uses of encoder, decoder and encoder-decoder models.

Model Type	Use Cases
Encoder (BERT)	Sentiment Analysis, Text Classification
Decoder (GPT)	Text Generation, Language Translation
Encoder-Decoder (BART)	Machine Translation, Summarization

2.5.4 Large Language Models

Large Language Models (LLMs) are extensions of the transformer-based pre-trained language models which have been trained on massive (gigabytes and petabytes) corpora of textual data from various sources including Wikipedia, books and the Internet. LLMs are capable of performing general purpose human language tasks such as question answering, text generation, text classification and document summarization [8]. LLMs also use billions of parameters. Parameters are the variables or underlying relationships that LLMs learn from their training data in order to perform prediction. The availability of faster parallel processors (GPUs, TPUs), massive datasets and storage devices are some of the reasons for the recent accelerated improvement in the development of various LLMs [46]. Popular LLMs include ChatGPT by Open AI, Llama by Meta and Gemini by Google [8]. LLMs have the ability to learn various statistical interrelationships from very large textual datasets through self-supervised and semi-supervised training. In self-supervised learning, a model learns from unlabeled data (such as text or audio) during pre-training by automatically generating its own labels directly from those data [32]. However, in semi-supervised learning, the model is trained with labelled and unlabelled data simultaneously [47]. A LLM architecture consists of several layers of feed-forward neural networks and transformers specifically optimized for processing vast amounts of text datasets from various sources.

2.5.5 Challenges Associated with Language Models

Despite their increasing popularity in recent years, language models including LLMs still face several challenges, some of which are highlighted below.

- Language models could amplify or reinforce biases associated with some races, ethnic groups or cultures across the world [8]. Hence, human agents may be required to fact check and correct misinformation associated with some LLM outputs.

- The massive and prolonged computations associated with LLMs contribute to global warming through their carbon footprints [48].

- Training of language models, especially LLMs, is expensive and have huge hardware requirements. Hence, only large companies have the capacity to train LLMs [49].

2.5.6 Code Language Models

Code Language Models are specialized language models that are pre-trained on the source code of various computer programming languages including C++, C, Python, Java, JavaScript and Go. Pre-training is the process whereby the model is first trained on massive unlabelled data. Pre-training is discussed later in detail in Section 2.7.4. Examples of code language models include CuBERT [50], CodeBERT [2], GraphCodeBERT [11] and PLBART [51].

2.6 Overview of selected Code Language Models

Before the introduction of code language models, various types of source code embedding techniques were developed. Source code embedding is a representation of software code with high-dimensional vectors created by using deep neural networks to train on large volumes of software source code samples [52, 53]. Unlike raw texts of source code which cannot be processed directly by ML/DL models, embeddings provide the numeric vector equivalents of the text which the models take as inputs. An example of a source code

embedding is code2vec which was developed by researchers at Facebook and Technion [54]. However, the development of code language models such as CuBERT, CodeBERT and GraphCodeBERT helps to achieve the task of generating improved contextual source code embedding without having to manually craft the required features from the source code [2, 50, 11].

2.6.1 CuBERT

CuBERT (Code Understanding BERT) is the first transformer-based model that is specifically trained to understand software source code [50]. This was achieved by pre-training the original BERT architecture on Python code samples which were collected from GitHub. The corpus that was used contained 9.3 billion tokens in 7.4 million Python files. CuBERT was evaluated on a benchmark of five classification tasks, one localization task and one repair task. The five classification tasks are: a variable-misuse, a wrong binary operator, an exception type, a swapped operand, and a function-docstring [50]. The purpose of a localization task is to detect where a bug lies in the sequence, while the goal of a repair task is to find a suitable replacement for the bug [50]. CuBERT performance was found to outperform the LSTM model, the BiLTSM-Word2Vec model and the first transformer model [50].

2.6.2 CodeBERT

CodeBERT is a BERT-based code language model developed by researchers at Microsoft in 2020. It is the first bimodal model for multiple programming languages including Python, Java, JavaScript, PHP, Ruby and Go. CodeBERT is considered bimodal because it is trained to understand the semantic relationship between human natural language (NL) and programming language (PL) [2]. CodeBERT is a general-purpose code model because it can perform varieties of NL-PL understanding tasks, including code search. It can also perform NL-PL generation tasks, including code documentation. CodeBERT can also generalize (extend) its capabilities to other programming languages that are not included in

its pre-training [2]. Like all transformer models, CodeBERT architecture consists of several stacked layers of transformers. It is designed to achieve two training tasks called Masked Language Modelling (MLM) and Replaced Token Detection (RTD). MLM was explained earlier in Section 2.5.1. Replaced Token Detection is the process of intentionally corrupting an input text sequence by replacing some tokens with sampled possible alternatives from a generator and predicting whether or not each token in the corrupted input was replaced by a generator sample [10]. The goal of RTD is not to predict the original corrupted token but to determine if each token was replaced. Unlike CuBERT, which was trained on a single programming language (Python), CodeBERT is trained on six programming languages. Table 2.3 shows the information about data used in training CodeBERT. The bimodal data consists of natural language and programming language code pairs while the unimodal data consists of only programming language samples. CodeBERT has a total of 125 million parameters.

- **CodeBERT Architecture**: CodeBERT is based on an optimized BERT architecture called RoBERTa [55] which uses a multi-stack layer of transformers. Because it requires bimodal data for training, CodeBERT has two dedicated generators as shown in Fig 2.4. The first generator is the Natural Language (NL) Generator while the second is the Code Generator. The two generators are specialized language models for generating probable tokens for the masked positions depending on the surrounding contexts [2]. Another important component is the NL-Code Discriminator which is the targeted pre-trained model. It is trained by detecting potential alternative tokens that are randomly sampled from the NL and PL generators.

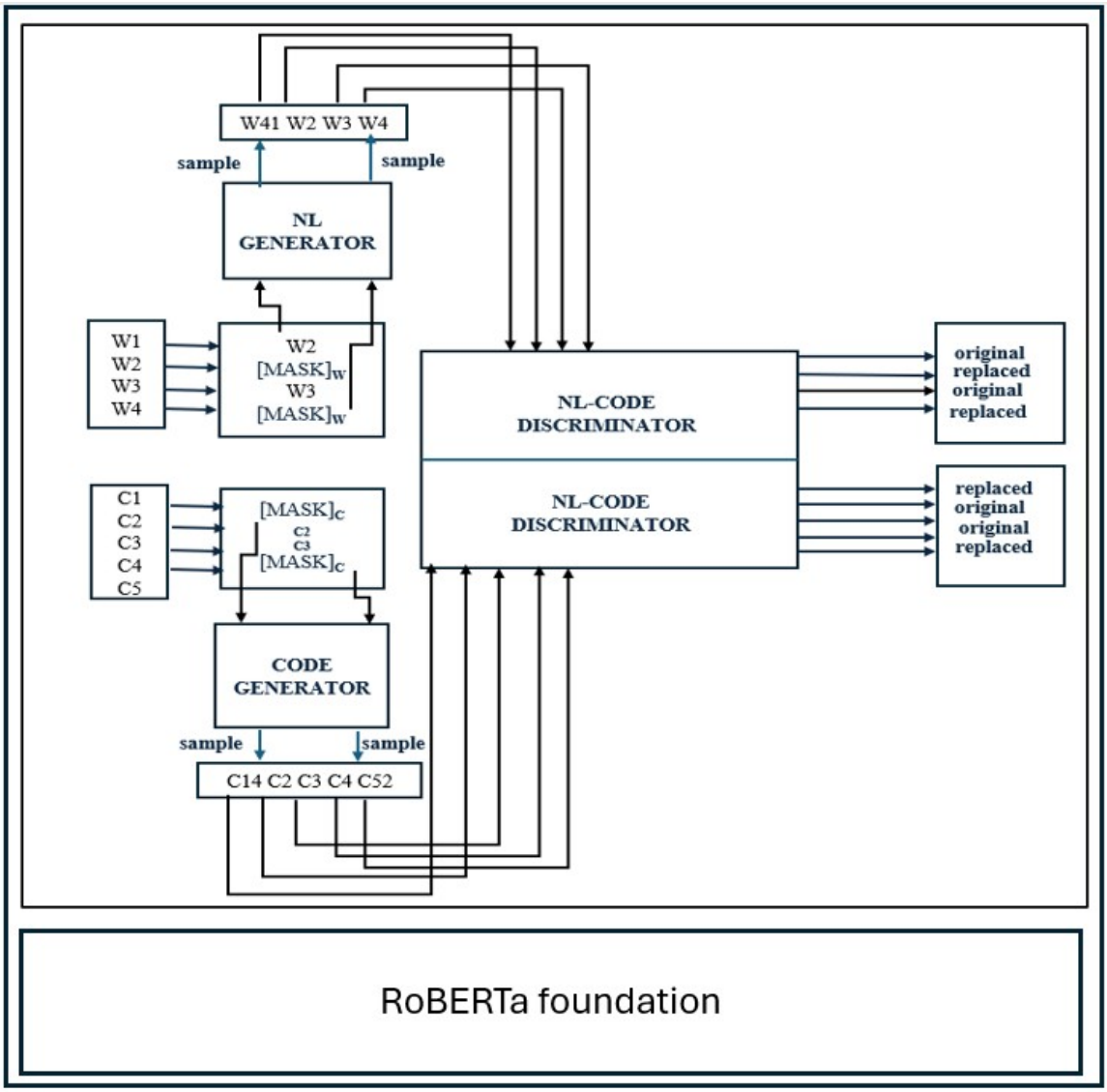

Figure 2.4: Architecture of CodeBERT including its RoBERTa foundation [2].

- **Components of CodeBERT**

  1. RoBERTa model: This is the natural language optimized BERT model from which CodeBERT was derived. RoBERTa has more parameters (355M) than the classical BERT model (340M).

  2. NL Generator: This component generates potential possible tokens for the masked positions of human text.

  3. PL Generator: This component generates potential possible tokens for the masked positions of code text.

4. NL-Code Generator: This component detects potential replacement (alternative) tokens sampled from NL and PL generators to achieve training.

Table 2.3: Datasets used in training CodeBERT [2].

Training Data	Bimodal Data (Code Pairs)	Unimodal Code Samples
PHP	662,907	977,821
Java	500,754	500,754
Python	458,219	1,156,085
Go	319,256	726,768
JavaScript	143,252	1,857,835
Ruby	52,905	164,048
Total	2,137,293	6,452,446

2.6.2.3 Applications of CodeBERT

Because of its encoder-based architecture and pre-training on programming and natural language, CodeBERT is capable of the following tasks.

- **Code Search**: This involves searching for code statements that are semantically related within a collection of code space [2].
- **Code Generalization**: CodeBERT is capable of generalization to programming languages that were not utilized in its pre-training [2]. However, this capability is limited. Hence, fine-tuning with additional datasets is ideal to improve this capability.
- **Clone Detection**: CodeBERT can be optimized to detect code clones (identical code or copied code). This is because a code clone will produce a contextual embedding identical to or closely similar to the original code even if it is slightly modified [56].
- **Unified Code Encoder**: CodeBERT is capable of representing both natural language and code in the same shared embedding space thereby allowing natural language to code translation and vice versa [2].

- **Defect Prediction**: CodeBERT is capable of predicting defects in software codes [57].

2.6.3 GraphCodeBERT

GraphCodeBERT is a transformer-based model for programming languages. Like CodeBERT, it was derived from an optimized BERT model called RoBERTa by a team of Microsoft researchers [11]. However, GraphCodeBERT differs from CodeBERT because it leverages the structure of code through data flow to learn code representation.

- **GraphCodeBERT Architecture**: As shown in Figure 2.5, GraphCodeBERT uses the concept of data flow in its architecture. A data flow is a graph which represents the interdependence between variables [11]. The variables are represented by the nodes while their origin is represented by the edges. According to Guo [11], data flow is always the same even if a source code is parsed under different abstract grammars. This is unlike Abstract Syntax Tree (AST) which varies with different abstract grammars. This is because the data flow feature of GraphCodeBERT enables it to focus on code semantics and not just the syntactic-level structure of the code. In the following sets of formulas, GraphCodeBERT takes the source code C, comments W, and the data flow G (C) as necessary input to its pre-training [11].

$$W = (w_1, w_2, w_3, \ldots, w_m) \quad \text{Comment}$$

$$C = (c_1, c_2, c_3, \ldots, c_n) \quad \text{Source Code}$$

$$G(C) = (V, E) \quad \text{Data Flow}$$

$$V = (v_1, v_2, v_3, \ldots v_k) \quad \text{Set of variables}$$

$$E = (E_1, E_2, E_3, \ldots E_L) \quad \text{Set of edges}$$

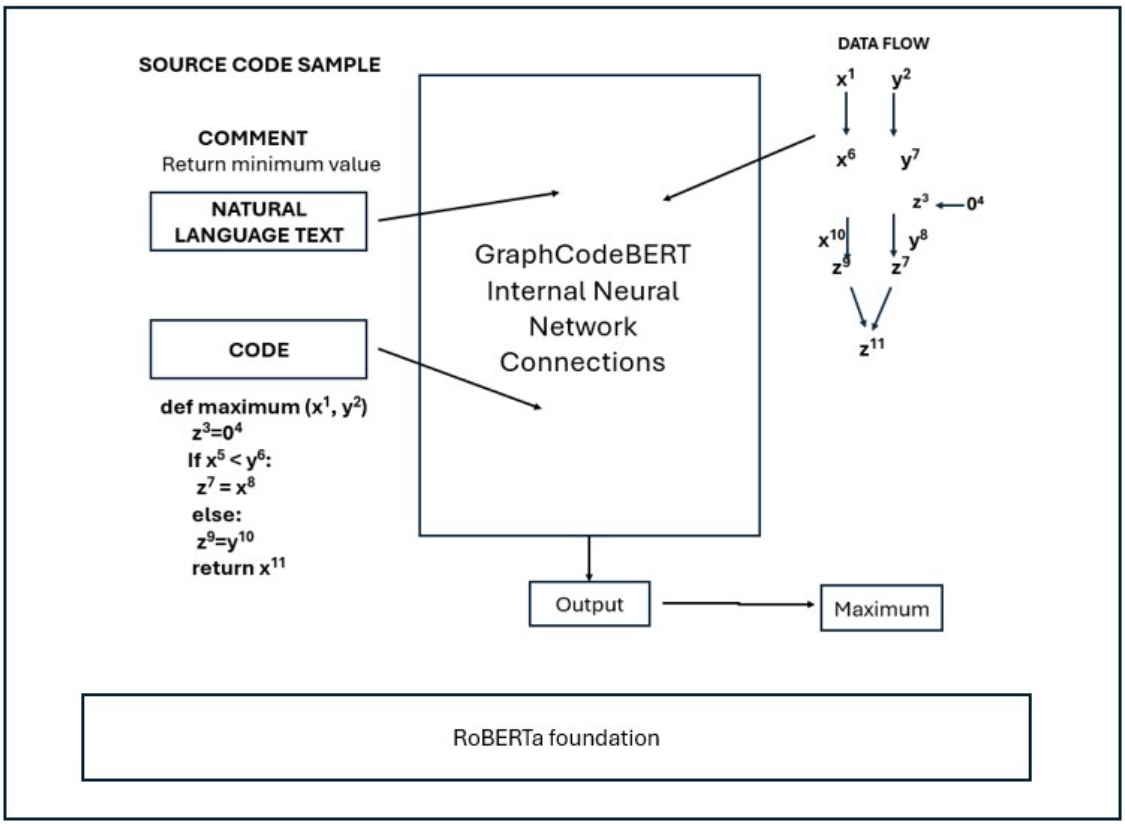

Figure 2.5: Architecture of GraphCodeBERT [11].

2.7 Steps in Code Language Model Processing

There are four stages involved in preparing a code language model for specific tasks.

2.7.1 Dataset Acquisition

Dataset acquisition is the first step in the code LLM workflow. It involves obtaining relevant datasets from appropriate sources. These include public repositories such as GitHub, Gitlab and Kaggle.

2.7.2 Dataset Preparation

Dataset preparation refers to all preliminary activities that are carried out to make the data suitable for model training, including data cleaning, data transformation and data validation. Data cleaning includes identifying and removing duplicates, errors and inconsistencies while data transformation includes conversion of data to appropriate formats using techniques

such as normalization, and feature engineering. Other dataset preparation activities include Exploratory Data Analysis (EDA) for pattern searching or statistical inferences [58, 59].

2.7.3 Tokenization

Tokenization is the conversion of the dataset texts into smaller units called tokens. These tokens could be unit characters, subwords or words. Common techniques include byte pair encoding (BPE), wordpiece and n-gram [1]. In LLMs, tokens are further converted into integer units called numerical IDs that directly map those token indexes to the embedding table of that model. In this way, the vocabulary of the model is derived. Embeddings are the arrays of dense vectors that encode syntactic information (structure and rules) and semantic information (meaning or interpretation) about the tokens [9, 10]. Embeddings are not the final output of LLMs but are used to build the internal vocabulary of the LLM which are later used for various forms of language processing with contextualized meaning.

2.7.4 Pre-training

During pre-training, self-supervised learning is used to train the model on a large corpus of textual data. Self-supervised learning is the process by which the model learns directly from the data without any additional need for labeling the data. The goal of self-supervised learning is to automatically learn useful representations (labels) within the data. Self-supervised learning is usually carried out in LLMs through Masked Language Modelling (MLM) and Next Sentence Prediction (NSP) [10]. In code LLMs, Replaced Token Detection (RTD) is used instead of NSP [2]. Pre-training enables the model to acquire general understanding of the language.

2.7.5 Transfer Learning and Fine-tuning

Transfer learning is the process of enabling a pre-trained model to transfer some aspects of their general knowledge to a specific but related task. This is achieved through a technique called fine-tuning [60]. Fine-tuning is achieved by selectively training (optimizing) an

already trained model to perform a specific task. This is usually done by using a smaller dataset, usually in a supervised manner. The supervised manner refers to the training of a model with labelled data. During fine-tuning, the relevant weights in the neural network layers of the LLM are automatically updated to improve its performance on a particular downstream task. In LLMs, the term ”downstream” is typically used to refer to a specific practical application of a trained model in the real world such as text classification, sentiment analysis, or question answering [61].

2.8 Source Code Weaknesses and Vulnerabilities

Vulnerabilities in source code continue to pose a significant challenge across the world. According to the globally recognized not-for-profit organization called MITRE Corporation, a weakness is a condition in a software that, under certain circumstances could contribute to the introduction of vulnerabilities, and are in many cases caused by the developer during the software development process [62]. Similarly, the US National Vulnerability Database (NVD) defines a vulnerability as a weakness in the computational logic of software that if exploited results in a negative impact to confidentiality, integrity, or availability [63].

2.8.1 Software Defects

Software defect is a generic term used to refer to unintended flaws in a program that causes it to behave incorrectly such as giving a wrong output. Defects are usually loosely referred to as bugs, errors or faults [64]. A defect is not necessarily or automatically a security issue unlike vulnerabilities which are mainly security risks. However, if uncorrected, defects could serve as starting points for weaknesses that lead to vulnerabilities.

2.8.2 Reporting Frameworks for Software Vulnerabilities

There are various standard vulnerability reporting frameworks that are used across the world. These include: Common Weakness Enumeration (CWE) [62], Common

Vulnerabilities and Exposures (CVE) [65], Open Worldwide Application Security Project (OWAPS) Top 10 [3], and the Common Vulnerabilities Scoring System (CVSS) [66].

2.8.3 Common Weakness Enumeration (CWE)

CWE provides a uniform measuring technique both for software security analysis tools as well as a generic baseline to identify weaknesses and their mitigation procedures. CWE contains thousands of identified software weaknesses. Each weakness is assigned a unique numeric identifier (ID) which contains specific information about the weaknesses. Table 2.4 shows the CWE Weaknesses (CWE-ID) for 2023.

Table 2.4: 2023 CWE Weaknesses for 2023 [41].

CWE-ID	**Name**	**Rank**
CWE-787	Out-of-bounds Write	1
CWE-79	Cross-site Scripting	2
CWE-89	SQL Injection	3
CWE-416	Use After Free	3
CWE-20	Improper Input Validation	4
CWE-125	Out-of-bounds Read	5
CWE-78	OS Command Injection	6

2.8.4 Common Vulnerabilities and Exposures (CVE)

CVE is a standard security database of known cybersecurity vulnerabilities and exposures [67]. CVE provide standard names, identification numbers and other parameters for identifying known vulnerabilities. This makes it faster and easier for IT security professionals to track them and collaborate with others.

2.8.5 Open Worldwide Application Security Project (OWAPS)

OWAPS Top 10 is a continually updated list of top 10 web application security risks maintained by an online IT and research community called OWAPS [3]. Table 2.5 shows the list of the top 10 critical web application security risk and vulnerabilities [3].

Table 2.5: OWAPS Top 10 List [3].

Rank	ID	Name
1	A01:2021	Broken Access Control
2	A02:2021	Cryptographic Failures
3	A03:2021	Injection
4	A04:2021	Insecure Design
5	A05:2021	Security Misconfiguration
6	A06:2021	Vulnerable and Outdated Components
7	A07:2021	Identification and Authentication Failures
8	A08:2021	Software and Data Integrity Failures
9	A09:2021	Security Logging and Monitoring Failures
10	A10:2021	Server-Side Request Forgery (SSRF)

2.8.6 Common Vulnerabilities Scoring System (CVSS)

Common Vulnerabilities Scoring System (CVSS) is a standard metric used to qualitatively measure the severity of vulnerabilities [4]. CVSS provides a uniform, consistent and accurate measurement of vulnerability severities across organizations, industries and governments, thereby enabling them to prioritize remediation activities based on the severity of the vulnerability. There are five vulnerability severity ratings (none, low, medium, high, and critical) with their corresponding severity score range as shown in Table 2.6. The higher

the severity score range of a particular vulnerability, the higher the severity rating of that vulnerability.

Table 2.6: CVSS Severity Rating Scale [4].

Rating	**Severity Score Range**
None	0.0
Low	0.1 – 3.9
Medium	4.0 – 6.9
High	7.0 – 8.9
Critical	9.0 – 10.0

2.9 LVDAndro Dataset

The LVDAndro dataset we used in our study contains some of the standard vulnerability reporting metrics including CWE, CVSS and OWAPS (detailed in sections 2.8.3, 2.8.5 and 2.8.6).

2.9.1 Data Fields in the LVDAndro Dataset

The LVDAndro dataset contains several data fields as shown in Table 2.7. The most important fields required for training a model for our classification task are the processed code and the vulnerability status fields. The processed code field is derived from the code field (raw source code samples) after performing data cleaning. Data cleaning includes identification, correction or removal of duplicates, missing values and other data inconsistencies.

Table 2.7: Data fields in the LVDAndro dataset [5].

Field	**Meaning**
Index	Numeric identifier generated automatically
Type	Vulnerability type
Pattern	Vulnerable code pattern
Code	Unprocessed source code
Severity	Vulnerability severity
CVSS	Common Vulnerability Scoring System
CWE_ID	Common Weakness Enumeration ID
CWE_Desc	Common Weakness Enumeration Description
OWASP_Mobile	OWAPS for mobile security threats.
OWAPS_MASVS	OWAPS for Mobile Security Verification Standards
Reference	Reference URL for the CWE vulnerability
Processed_Code	Pre-processed source code
Vulnerability Status	Safe code (0), Unsafe code (1)

2.9.2 CWE-IDs of the Dataset

Several CWE-IDs are available in the LVDAndro dataset. CWE-ID was explained earlier in section 2.8.3. The CWE-IDs assist in identifying the type of security weaknesses based on MITRE Corporation's standard and the open-source community guidance. Table 2.8 lists some of these CWE-IDs and their description.

Table 2.8: Examples of CWE-IDs in the LVDAndro dataset [5].

CWE-ID	**Description**
CWE-926	Improper exploit of Android application components
CWE-327	Use of broken or risky cryptographic algorithm
CWE-200	Exposure of sensitive information to an unauthorized actor
CWE-299	Improper check for certificate revocation
CWE-312	Cleartext storage of sensitive information
CWE-532	Insertion of sensitive information into log file
CWE-921	Storage of sensitive data in a mechanism without access control
CWE-749	Exposed dangerous method or function

2.9.3 Examples of Vulnerable Code Samples

Some examples of vulnerable code samples based on their standard CWE-IDs (available in the LVDAndro dataset) are as follows:

1. **CWE-327 (Use of Broken or Risky Cryptographic Algorithm) [68]**.

   The following code is vulnerable due to its use of risky encryption algorithm called the Data Encryption Standard (DES).

   ```
   Cipher des = Cipher.getInstance("DES...");
   des.initEncrypt(key2);
   ```

2. **CWE-532 (Insertion of Sensitive Information into Log File) [69]**.

   The following code is vulnerable due to the inclusion of very sensitive personal information such as a credit card number in the log file.

   ```
   logger.info("Username: " + usernme + ", CCN: " + ccn);
   ```

3. **CWE-749 (Exposed Dangerous Method or Function) [70]**.

   The following code is vulnerable because a critical database operation (DROP) is written inside a Java method that is already declared public. This sensitive operation is supposed to be restricted and not exposed to every class in that application.

   ```
   public void removeDatabase(String databaseName) {try
   { Statement stmt = conn.createStatement();
   stmt.execute("DROP DATABASE " + databaseName);
   } catch (SQLException ex) {...}
   }
   ```

2.10 Related Work

Due to the unique ability of LLMs to understand the contextual, semantic and syntactic inter-relationships in source code, various research projects have explored how these models could be optimized to perform various static analysis tasks on code.

Pan *et al.* explored the use of CodeBERT in 2021 for the prediction of software defects using standard metrics such as naturalness, size and complexity [66]. However, Pan's study was not focused on software security or vulnerabilities detection.

Huang *et al.* carried out a study in 2022 on the detection of vulnerabilities in C/C++ source code using natural language-based BERT models including RoBERTa, DistillBERT and MobileBERT and recommended the use of more real-world datasets to improve the effectiveness of the models [19]. Moreover, this study by Huang's team adopted only natural language-based models and C/C++ datasets.

In 2019, Zhou *et al.* developed a graph neural network-based model called Devign for software vulnerabilities detection by training the model on manually curated datasets which were obtained from large-scale open-source C projects [71]. However, this model lacked transformers in its architecture which limited the models' contextual and semantic understanding of source code.

In 2023, Senanayake *et al.* explored the use of conventional ML models such as Support Vector Classifier (SVC), Random Forest (RF) and Multilayer Perceptron (MLP) as proof of concept for the detection of Android vulnerabilities [5]. Senanayake *et al.* also produced a dataset called LVDAndro from thousands of Android applications for training ML/DL models. A limitation in this work was their use of conventional ML models which cannot be optimized for language (text) processing or code understanding.

Chapter 3

Methodology

3.1 Methodology

This chapter is divided into three main sections. In the first section, we discuss the infrastructure, computation resources and libraries used in the implementation of the project. In the second section we focus on dataset related activities including dataset acquisition, dataset preparation and various dataset preprocessing stages necessary for ensuring optimal training of the selected model. In the third section, we discuss the selected models, and the strategies used in re-training (fine-tuning) them for the downstream task of code vulnerabilities detection via binary classification. The high-level flow chart of the implementation is shown in Figure 3.1. In order to have access to state-of-the-art computational resources such as high-performance GPUs and storage, Google Colab [72] was adopted for the implementation of the project. Overall, the design is subdivided into the infrastructure provisioning, data acquisition, data preparation, model selection, model fine-tuning, model inference and model evaluation phases.

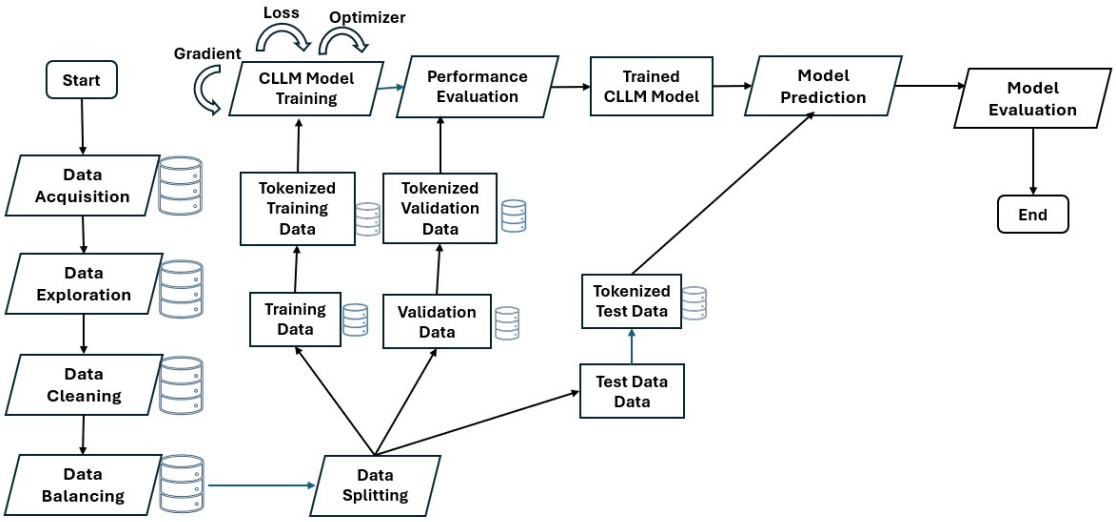

Figure 3.1: Implementation flow chart.

3.2 Infrastructure Provisioning

In order to take advantage of the high-performance computing infrastructure and storage necessary for implementing the project, two Google cloud services were adopted. These are Google Colab, Google Drive [73] and Hugging Face [74]. Google Colab is a cloud-based Jupyter [3] notebook service, while Google Drive is a file storage service where the datasets used in the project were stored. Hugging Face is a cloud service for LLM training.

3.2.1 Computation Resources

The hardware accelerator used in the project is the A100 Graphic Processing Unit (GPU) provided via the paid subscription platform of Google Colab. The A100 GPU is a Tensor Core GPU developed by NVIDIA [75]. This GPU uses the NVIDIA Ampere Architecture providing about 20 times performance improvement over previous GPUs. It also has a fast memory bandwidth of over 2 terabytes per second. This gives the GPU the capability of running the huge datasets and models [75].

3.2.2 ML/DL Libraries and Frameworks

Several data analysis and ML/DL libraries were used in the project. These include: Pandas [76], Mathplotlib [77] and scikit-learn [78]. Pandas was used to perform various exploratory data analysis, data cleaning and data cleaning tasks on the datasets. Mathplotlib was used in performing data visualization on the datasets such as plotting graphs. scikit-learn and Hugging Face were used for performing the deep learning tasks on the models and datasets.

3.3 Dataset Acquisition and Overview

The recently released LVDAndro dataset was acquired for training the selected models [5]. This dataset contains real-life Android source codes curated by scanning more than 15,000 apps and classifying the results into safe or vulnerable based on the Common Weakness Enumeration. Android OS and applications are generally developed using a combination of several languages including Java, C and C++ [79]. However, in 2017 another programming language called Kotlin which compiles to Java bytecode was announced by Google as its official programming language for Android [80]. The LVDAndro dataset contains more than 2 million unique code samples which are provided for open-source usage on GitHub by the curator and publisher [5]. Figure 3.2 shows how the LVDAndro dataset was synthesized by Senanayake *et al.* [5]. It involves scraping of Android APK (Application Package) and source files from various repositories, scanning and annotating them with Android vulnerability scanners (Mobile Security Framework (MobSF) [81] and Quick Android Review Kit [82]), and finally aggregating them into a dataset of safe and vulnerable code. We believe the reason Senanayake *et al.* used two separate Android vulnerability scanners during the dataset preparation rather than one was to reduce the margin for errors because no single scanner is 100% perfect. By training our code LLMs on the LVDAndro datasets, the model should be able to learn the vulnerability detection capabilities of these scanners.

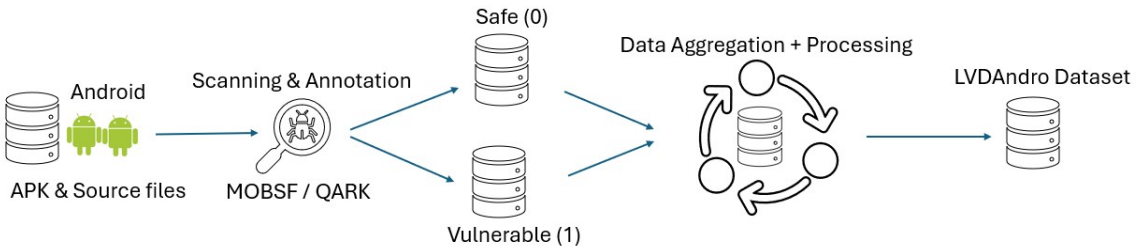

Figure 3.2: How LVDAndro dataset was created [5].

3.3.1 Overview of Scanners used in producing LVDAndro dataset

MobSF and QARK are the two vulnerabilities scanning tools originally used in producing the LVDAndro datasets [5].

- **MobSF** is an open-source automated vulnerabilities scanner for mobile apps including Android and iOS [83]. According to [84], MobSF is one of the security assessment tools recommended by the OWAPS Mobile Security Testing Guide [83]. MobSF uses static and dynamic analysis for the identification of vulnerabilities. Static analysis involves examination of a code without executing it [85]. Dynamic analysis focuses on the behaviour of the software during runtime. MobSF also ranks the identified vulnerabilities based on the severity levels [84].

- **QARK** is a vulnerabilities scanner that uses static code analysis technique to identify vulnerabilities in Android applications. According to [86], the interpretation of QARK findings is more difficult for users that are not familiar with Android application development platform [86]. Based on the study conducted by [6], a comparison of the performance of MobSF and QARK is shown in Table 3.1.

Table 3.1: Comparison of MobSF and QARK vulnerabilities detection performance [6].

Metrics	MobSF	QARK
Accuracy	0.91	0.89
Precision	0.93	0.92
Recall	0.95	0.93
F1	0.94	0.92

3.3.2 Dataset Composition

The LVDAndro dataset consists of three main datasets: Dataset 1, Dataset 2, and Dataset 3. We have maintained the same names for our adopted datasets and subdatasets to ensure consistency with their original names and to easily compare our model's performance with theirs. Datasets 1 and 2 are further sub-divided into QARK, MobSF and Combined subdatasets based on the scanning method that was used in synthesizing and annotating them. Figure 3.3 shows the composition of the LVDAndro dataset.

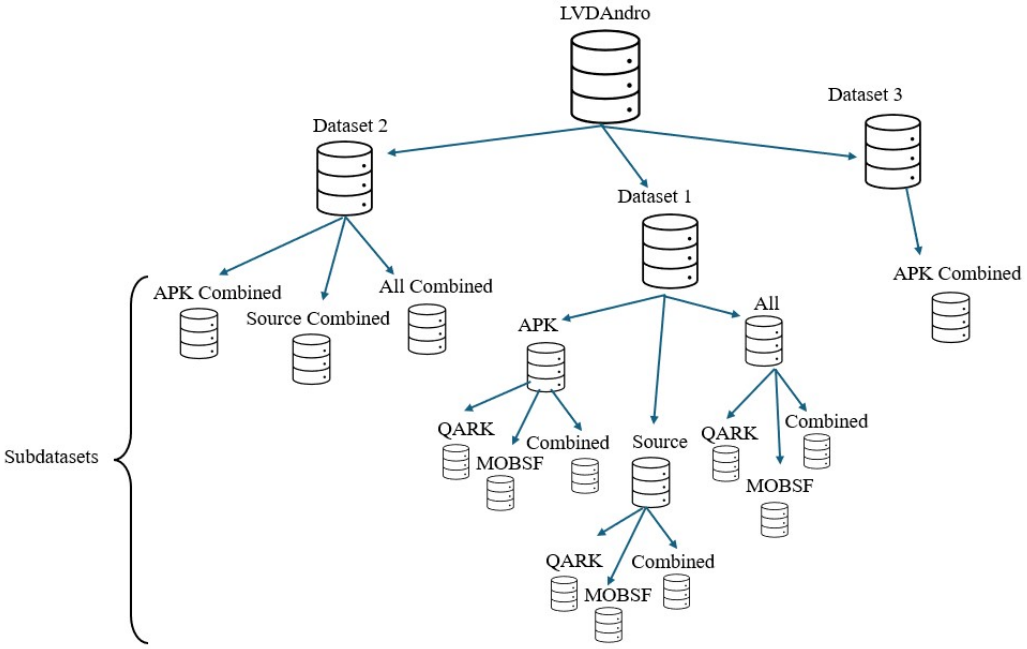

Figure 3.3: LVDAndro dataset composition [5].

The Combined dataset was produced from both the QARK and MobSF scanner. Each subdataset is further divided into three smaller subdatasets APK, Source files and All (APK and Source). The APK subdatasets contain compiled Android installation files, the Source subdatasets contain Android source code, and the All subdatasets contain both [5]. All of the subdatasets contain various types of Android vulnerabilities which will enable our selected models to learn a wide variety of vulnerabilities.

Dataset 1 consists of 511 Android apps from all of the major open-source APK and Android projects in FossDroid repository [5]. Dataset 2 consists of 5,503 open Android APK and Android projects in the FossDroid app repository across 17 project categories, including the internet, systems, games, and multimedia [5]. Dataset 3 consists of 15,021 Android APK from three major repositories including FossDroid, AndroVul and Android. Dataset 3 is composed of only APK Combined subdatasets [5].

3.4 Data Preparation

We performed various data preparation activities to ensure the LVDAndro dataset is well suited for training our models. These data preparation activities are explained in the following sections.

3.4.1 Exploratory Data Analysis

The first step in the data preparation phase involves exploratory data analysis where Pandas functions were used to read the dataset from CSV format into a data frame in memory. The data was stored as data frames to allow the presentation of the data in tabular rows and columns. Various Pandas functions such as head, tail, shape, describe, column, nunique, and value counts were used in exploring the dataset further to extract preliminary information about its structure and composition.

3.4.2 Data Cleaning

The second task in the data preparation phase is the data cleaning. This is typically necessary for removing inaccuracies, errors, incomplete data or empty rows in the dataset. Rows with missing values and unnecessary columns were removed appropriately. Data cleaning generally helps in mitigating undesirable conditions such as overfitting and underfitting [87]. Overfitting is an undesirable condition in which models perform very well on training data but poorly on unseen data such as test data [87]. Conversely, underfitting is the situation where the model performs poorly on both the training and unseen data [87].

3.4.3 Data Balancing

Another vital step in the data preparation process is the data balancing. This is used for addressing cases of class imbalance in machine learning and deep learning. Class imbalance is common in most software vulnerability datasets because the majority class (safe code) typically has a higher proportion than the vulnerable code. This is because a standard or good quality software application is expected to have fewer vulnerabilities, bugs or weaknesses in real life environments. Consequently, each of the subdatasets in the LVDAndro dataset contains a significantly higher proportion of safe Android code samples (labelled as 0) than vulnerable code (labelled as 1). This is shown in Figure 3.4 where one of the subdatasets in the LVDAndro contains more than 5,000,000 safe code samples but just 61,241 vulnerable code samples.

Class imbalance in datasets can lead to inaccuracies, bias or poor generalization in the performance of any machine learning or deep learning model [88]. To avoid this, data balancing was performed once on each of the subdatasets in the LVDAndro dataset. Data balancing was also performed in the previous study [5]. We used a standard technique called random undersampling [89]. This technique involves matching the minority class by randomly deleting rows from the majority class. Afterwards, both the majority class and the minority class have a balanced data proportion in each subdataset. An advantage of this

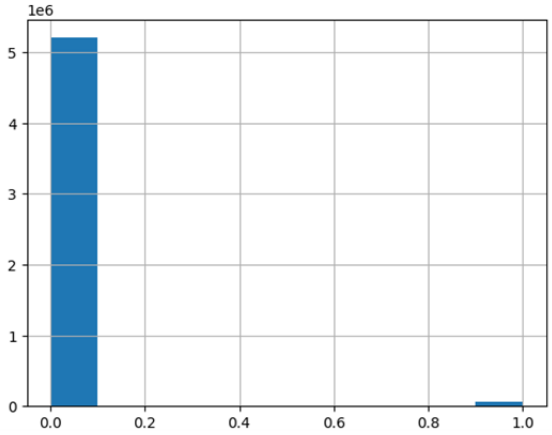

Figure 3.4: Proportion of Android safe code (0) and vulnerable code (1).

approach is that the size and data of the minority class (vulnerable code samples) always remain the same even if random undersampling is performed multiple times. This is vital because vulnerable code samples are the main target of our models. Figure 3.5 shows the outcome of the data balancing via random undersampling. Both the majority and minority class now have equal proportion. Table 3.2 shows the outcome after data balancing has been performed on each of the imbalanced subdatasets of the LVDAndro dataset.

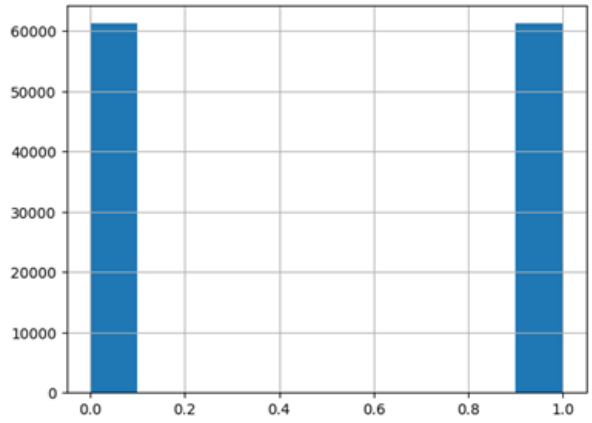

Figure 3.5: Balanced proportion of Android safe (0) and vulnerable samples (1).

Table 3.2: Proportion of safe and vulnerable code samples before and after data balancing.

	Before Data Balancing			**After Data Balancing**		
DATASET 1	**Safe**	**Vul**	**Total**	**Safe**	**Vul**	**Total**
APK QARK	2754233	22594	2776827	22594	22594	45188
Source QARK	589317	5745	595062	5745	5745	11490
All QARK	3310569	28141	3338710	28141	28141	56282
APK MobSF	1875229	28221	1903450	28221	28221	56442
Source MobSF	609594	10916	620510	10916	10916	21832
All MobSF	2411087	37916	2449003	37916	37916	75832
APK Combined	4441089	49853	4490942	49853	49853	99706
Source Combined	847721	12738	860459	12738	12738	25476
All Combined	5201348	61241	5262589	61241	61241	122482
TOTAL	**22040187**	**257365**	**22297552**	**257365**	**257365**	**514730**
DATASET 2						
APK Combined	12228925	127366	12356291	127366	127366	254732
Source Combined	4324827	52286	4377113	52286	52286	104572
All Combined	14481215	162649	14643864	162649	162649	325298
TOTAL	**31034967**	**342301**	**31377268**	**342301**	**342301**	**684602**
DATASET 3						
APK Combined	15279029	191949	**15470978**	191949	191949	**383898**

3.5 Data Conversion to Hugging Face format

The LVDAndro dataset was originally provided in a CSV format which cannot be used directly for training with Hugging Face LLMs. Hence, the dataset library within Hugging Face was used to load and convert each of the subdatasets of the LVDAndro dataset into the standard dataset dictionary format of Hugging Face by using the load dataset method.

3.6 Dataset Splitting

Each of the subdatasets in the LVDAndro dataset was split once into training, test and validation data by using a 60:20:20 ratio respectively [5, 90]. The training data is used for training the model by adjusting its weights and biases to minimize loss. The validation data is used for adjusting the model's hyperparameters, guides its performance and prevent

overfitting. The testing data is used to test the performance of the trained model on unseen data. The division of the dataset into different sections helps to ensure that the selected models are not trained on the same data points which could lead to bias, overfitting or other inaccuracies.

3.7 Data Tokenization

Similar to Natural Language Processing, the Android source code datasets cannot be used directly in their raw text formats in training the code LLMs. Hence, they were converted into suitable tokens and numeric IDs by using the Hugging Face AutoTokenizer class and model checkpoint. A checkpoint is a saved snapshot of a specific model type and state during its initial pre-training by its developer [74]. A checkpoint typically contains unique parameters such as weights and logs which allow other users to utilize the model for fine-tuning later. Since some models may have several versions, a checkpoint is used to identify different versions available for pre-training [74]. The checkpoints that we used for CodeBERT and GraphCodeBERT in this study are codebert-base and graphcodebert-base respectively. Both models were developed by Microsoft [11]. Similarly, bert-base-uncased is the official checkpoint of the BERT version that was selected for our experiment [74]. The output of the tokenization is automatically stored in PyTorch tensor format.

3.8 Model Selection

Two code-based LLMs (CodeBERT and GraphCodeBERT) and one natural language LLM (BERT) were selected for the experiments. Unlike the previous work that adopted three classical ML models (SVC, RF and MLP) [5], we chose CodeBERT and GraphCodeBERT for the following reasons:

- Both CodeBERT and GraphCodeBERT are transformer-based models. Hence, we hypothesize that both models will comprehend the syntax and semantics in text-based

data (such as source code) better than classical ML models. This is enabled by their attention mechanism explained earlier in section 2.4.3.

- Both CodeBERT and GraphCodeBERT were pre-trained on source code from six programming languages and thus possess a general understanding of structure for these languages.

3.9 Model Re-training (Fine-tuning)

As described in Chapter 2, the selected code-based LLMs (CodeBERT, GraphCodeBERT) have already been trained on source code from six programming languages (Go, Java, JavaScript, PHP, Python and Ruby) by their developers. The computational cost and other technical resources required to train another LLM from scratch would have made it impossible to achieve within the scope of this work. Thus, the typical standard in most LLM-based projects is to retrain the models on a smaller dataset optimized for a particular domain. In this work, the datasets for retraining are the Android datasets containing both the safe and vulnerable code samples. This process of re-training a model for a specialized task in a supervised manner is called fine-tuning. Figure 3.6 and Figure 3.7 show a high-level description of the fine-tuning process for CodeBERT and GraphCodeBERT respectively.

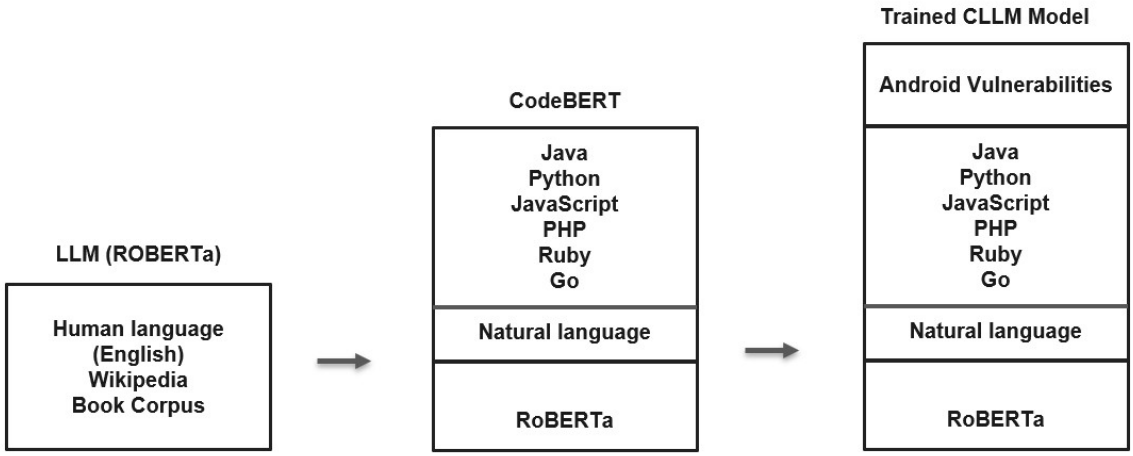

Figure 3.6: Fine-tuning CodeBERT with Android source code datasets.

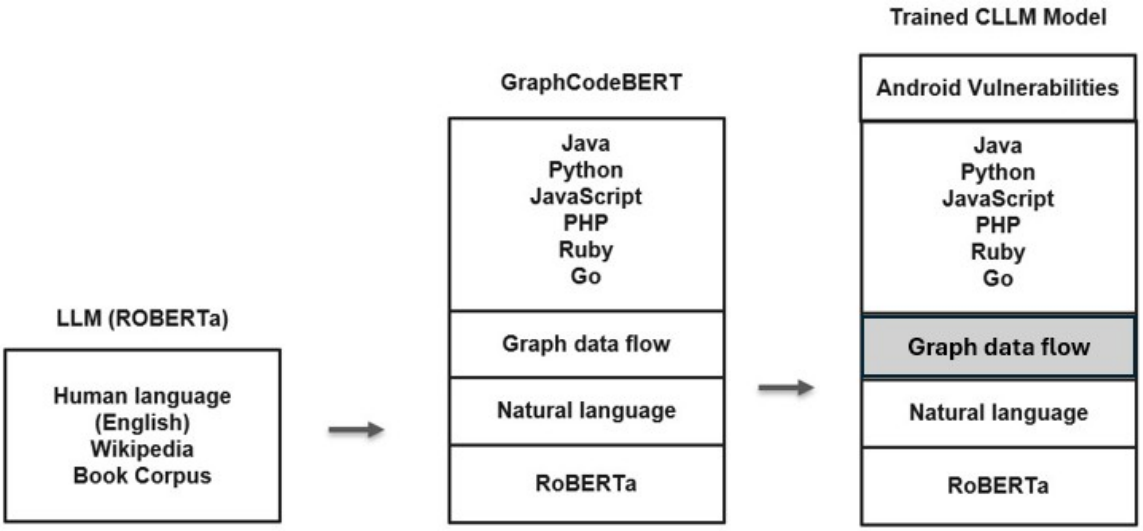

Figure 3.7: Fine-tuning GraphCodeBERT with Android source code datasets.

3.9.1 Trainer Class

A special class called Trainer in the Hugging Face library was used to perform the fine-tuning of the code LLMs for this task. It is an Application Programming Interface (API) that is available in the Hugging Face library to simplify the training, evaluation and saving of the transformer models used during fine-tuning. The trainer class uses PyTorch in the backend for its processing [74]. The trainer class accepts the parameters in Table 3.3 as input for fine-tuning of CodeBERT, GraphCodeBERT and BERT.

Table 3.3: Parameters for fine-tuning CodeBERT, GraphCodeBERT and BERT.

Type	**Parameter**
Tokenizer	AutoTokenizer from pre-trained CodeBERT AutoTokenizer from pre-trained GraphCodeBERT AutoTokenizer from pre-trained BERT
Model Checkpoint	Microsoft/codebert-base Microsoft/graphcodebert-base Microsoft/bert-base-uncased
Training dataset	Tokenized training dataset
Validation dataset	Tokenized validation dataset
Test dataset	Tokenized validation dataset
Computation metrics	Compute metrics

These parameters are described below:

- **Model Checkpoint**: A model checkpoint in the Hugging Face library represents the particular LLM model version that is used to perform the fine-tuning task. This is explained in detail in Section 3.7.

- **Tokenizer**: Tokenizer is a function in the Hugging Face library that performs two processing steps. It first converts the raw text into tokens before converting those tokens into numerical IDs that represent information in the model's vocabulary. There are several types of tokenizers, but the selection of each depends on the type of model being used for a fine-tuning task. In this study, we used the AutoTokenizer which automatically selects the appropriate tokenizer for each of our models (RoBERTaTokenizer for CodeBERT and GraphCodeBERT and BERTTokenizer for BERT). [74].

- **Training Arguments:** The training arguments are discussed in Subsection 3.9.2.
- **Tokenized Training Dataset:** This is the subset of the tokenized dataset used for training purposes.
- **Tokenized Validation Dataset:** This is the subset of the tokenized dataset that is used for validation purposes.
- **Tokenized Testing Dataset:** This is the subset of the tokenized dataset that is used for testing purposes.
- **Compute Metrics:** These are discussed in Section 3.11 (e.g. accuracy, precision, recall, and F1).

3.9.2 Training Arguments

The fine-tuning of CodeBERT, GraphCodeBERT and BERT was performed using the standard training arguments shown in Table 3.4.

Table 3.4: Training arguments for fine-tuning CodeBERT, GraphCodeBERT and BERT.

Type	**Actual**	**Description**
Output directory	Training directory	These are standard values typically used in Hugging Face for fine-tuning these models.
Evaluation strategy	Epoch	
Number of training epochs	4	
Per device training batch size	16	
Per device eval batch size	64	

- **Output Directory**: The output directory represents the path to the directory where the training outputs are saved. The outputs include the training logs, model checkpoints and other vital information [74].

- **Evaluation Strategy:** Evaluation strategy is a technique used to control how often evaluation is performed during fine-tuning. For instance, when the evaluation strategy is epoch, evaluation will be done at the end of each epoch. An epoch is a complete pass through the entire training dataset by the model during the training. Evaluation is crucial during fine-tuning because it helps to measure the performance of the model with unseen data [74].

- **Number of Training Epochs:** This is an integer number that determines the number of epochs that will be executed by the model [74].

- **Per Device Training Batch Size:** Per device training batch size is used to determine the number of training samples from the dataset that would be processed simultaneously in the processing unit (GPU or CPU) during each training step [74].

- **Per Device Eval Batch Size:** Per device eval batch size is used to determine the number of evaluation samples [74].

3.10 Model Prediction Task

The goal of using the selected models is to perform the task of differentiating vulnerable code from safe code. This is the model prediction task. Predictions can be safe code (0) or vulnerable (1). After fine-tuning CodeBERT, GraphCodeBERT and BERT models with the tokenized training and validation datasets, we used each of our trained models to perform vulnerability prediction on the unseen (test) data.

3.11 Model Evaluation

To evaluate the performance of the trained code LLMs, four standard metrics were used: accuracy, precision, recall and F1 [91]. The compute metrics function in the Hugging Face library was used to calculate each of these metrics [74]. Accuracy is used to determine the proportion of predictions that were correct out of all the predictions performed by the

model [40]. Both true correct positives and true negatives are vital in the calculation of the metrics as shown in Equation 3.1, Equation 3.2, and Equation 3.3. True positives are predictions of 1 that are correct while false positives are predictions of 1 that should be 0. Similarly, true negatives are predictions of 0 that are correct while false negatives are predictions of 0 that should be 1.

$$Accuracy = \frac{True\ Positive + True\ Negative}{True\ Positive + False\ Positive + True\ Negative + False\ Negative} \tag{3.1}$$

Precision measures the proportion of a model's positive classifications that are true positives [40].

$$Precision = \frac{True\ Positive}{True\ Positive + False\ Positive} \tag{3.2}$$

Recall is used to measure the proportion of the correct predictions out of all the actual true predictions [40].

$$Recall = \frac{True\ Positive}{True\ Positive\ +\ False\ Negative} \tag{3.3}$$

F1 combines both precision and recall as a harmonic mean [40].

$$F1 = \frac{2 * Precision * Recall}{Precision + Recall} \tag{3.4}$$

Figure 3.8 shows a screenshot of one of our model's output during training.

Run data is saved locally in /content/wandb/run-20250102_043654-qdidbhg5
Syncing run training_dir to Weights & Biases (docs)
View project at https://wandb.ai/oladokunbox-university-of-lethbridge/huggingface
View run at https://wandb.ai/oladokunbox-university-of-lethbridge/huggingface/runs/qdidbhg5
[24500/24500 2:15:18, Epoch 4/4]

Epoch	Training Loss	Validation Loss	Accuracy	Recall	Precision	F1
1	0.133700	0.136576	0.963096	0.961920	0.964431	0.963174
2	0.113200	0.128921	0.965954	0.969406	0.962981	0.966183
3	0.099100	0.130139	0.970771	0.971196	0.970564	0.970880
4	0.070700	0.127983	0.971179	0.974288	0.968457	0.971364

```
TrainOutput(global_step=24500, training_loss=0.11136809711067044, metrics={'train_runt
41.657, 'train_steps_per_second': 2.604, 'total_flos': 1.031237470377984e+17, 'train_l
```

Figure 3.8: Fine-tuning selected code LLMs with Android source code datasets.

Chapter 4

Experiments and Results

In this chapter, we present the various experiments and their outcomes. These experiments are divided into three main sections: Experiment 1, Experiment 2 and Experiment 3. As mentioned earlier in Chapter 3, there are three main datasets (Dataset 1, Dataset 2, and Dataset 3) used in this study. Dataset 1 contains nine subdatasets. These are APK QARK, Source QARK, All QARK, APK MobSF, Source MobSF, ALL MobSF, APK Combined, Source Combined and All Combined. Dataset 2 comprises three subdatasets. These are APK Combined, Source Combined, and All Combined. Dataset 3 contains one subdataset called APK Combined. The subdatasets have similar names based on the manual vulnerability scanning tools used in preparing and annotating them by the dataset creator [5], but they differ in sizes and in the type of Android software repositories from which they were prepared [5]. These datasets of Android code samples were used to train the selected LLMs for code vulnerability detection through binary classification where a safe code sample is classified as 0 and vulnerable code as 1.

Since these models were originally trained on much larger datasets of six programming languages, we aim to extend their generic understanding of programming languages to the related task of comprehending specifically Android source code. This process of enabling a pre-trained model to transfer some aspects of their general "knowledge" to a specific but related area is the transfer learning earlier mentioned in Subsection 2.7.5. To optimize the models for the detection of vulnerabilities in Android source code, we further trained the selected models on a labelled dataset of safe and vulnerable Android code. This is the

supervised fine-tuning process. In this way, the pre-trained models are optimized through their transformer attention mechanism to learn the unique patterns (syntax, semantics or structure) associated with the safe and vulnerable code samples.

Unlike the previous work [5] which used only two metrics (accuracy and F1), we used four standard metrics (accuracy, recall, precision and F1) across the three experiments to quantitatively measure the performances of the code LLM models. Since F1 is a harmonic average of precision and recall, obtaining precision and recall separately helps to get a more comprehensive result for the model's performance. The results of the fine-tuned LLM models (CodeBERT and GraphCodeBERT) in the vulnerabilities detection tasks were compared with the AutoML models (Random Forest – RF, Support Vector Classifier – SVC and Multi-layer Perceptron -MLP) used in the previous study.

4.1 Model Training, Validation and Testing

In the three experiments, each of the subdatasets were split into training, validation and testing datasets. We used a split ratio of 60:20:20 on each of the subdatasets to obtain 60% training, 20% validation and 20% test data sets. The training datasets were used in fine-tuning the pre-trained LLMs to adjust its parameters (neural weights and biases) for the required downstream binary classification task (vulnerability detection) through supervised learning. This was achieved by comparing the result of the code LLM's prediction with the target and correspondingly adjusting its parameters automatically during the training process to produce a trained model for vulnerability detection. The validation dataset was used to evaluate the performance of the code LLM during each epoch of the training process to prevent overfitting and underfitting. An overly complex model and inadequate training data can cause overfitting [87]. Overly complex models are associated with bias and variance problems and can be caused by the use of incorrect training parameters such as learning rate, regularization and drop-out [92]. Underfitting can be caused by factors including using an overly simple model, poor training data or poor parameter tuning [87].

An overly simple model may lack adequate parameters, neurons, proper architecture or algorithms necessary for capturing complex patterns in the data during training. We used the test dataset to measure the performance (generalization) of the trained code LLM on unseen data. To prevent overfitting and underfitting, the hyperparameter must be optimal in the training arguments before the model fine-tuning begins. We utilized the training class in the Hugging Face library to load the code LLM model, training arguments, training datasets, validation datasets, tokenizer and metrics into Google Colab notebook to achieve the entire fine-tuning workflow in each of the experiments.

4.2 Experiment 1

In Experiment 1, the nine subdatasets in Dataset 1 were used in fine-tuning the two selected code LLMs (CodeBERT and GraphCodeBERT) and one natural language LLM (BERT). The reason for the inclusion of BERT was to discover how a LLM which was originally trained on natural language only would perform in vulnerabilities detection after fine-tuning on datasets consisting of Android code. We found that the three LLMs performed well in the detection of vulnerabilities across the nine subdatasets. Based on the four metrics used in Experiment 1, the highest recorded performances (0.992 accuracy, 0.995 recall, 0.989 precision, 0.992 F1) exceeds the best performances of the previous work (0.92 accuracy, 0.99 F1) [5]. We discuss each model's performances in the following subsections.

4.2.1 Performance of CodeBERT

The performance of CodeBERT on vulnerabilities detection across each of the nine subdatasets in Dataset 1 is shown in Table 4.1. In the APK QARK, Source QARK and All QARK subdatasets, the lowest and highest accuracy are 0.950 and 0.961 respectively while the recall ranges between 0.958 and 0.966. The lowest and highest precision are 0.944 and 0.960 respectively. Similarly, the F1 also ranges between 0.951 and 0.962. In the

MobSF subdatasets, the model recorded the highest performance of three experiments. The accuracy, recall, precision and recall ranges between 0.989 and 0.992.

However, in the combined subdatasets, the accuracy dropped slightly to 0.974 while the recall varies between 0.973 and 0.980. Similarly, the precision ranges between 0.962 and 0.968 while the F1 ranges between 0.968 and 0.974. Overall, the general performance of CodeBERT as shown in Table 4.1 is better than the performance of all of the ML models (RF, SVC and MLP) used in the previous study [5] as shown in Table 4.2. For instance, the lowest recorded accuracy and F1 are 0.95 and 0.95 respectively for CodeBERT unlike the ML models (RF, SVC) which had 0.90 for accuracy and 0.90 for recall.

Table 4.1: Performance of CodeBERT on Android vulnerabilities detection in Dataset 1.

Dataset 1	**CodeBERT Binary Classification**				
Subdataset Name	**Size (MB)**	**Accuracy**	**Recall**	**Precision**	**F1**
APK QARK	305.3	0.961	0.966	0.957	0.962
Source QARK	70.5	0.950	0.958	0.944	0.951
All QARK	373.5	0.962	0.963	0.960	0.962
APK MobSF	242.4	0.993	0.996	0.989	0.992
Source MobSF	77.3	0.983	0.984	0.982	0.983
All MobSF	312.1	0.992	0.995	0.989	0.992
APK Combined	558.9	0.974	0.980	0.968	0.974
Source Combined	111.1	0.967	0.973	0.962	0.968
All Combined	661.1	0.973	0.980	0.967	0.973

Table 4.2: Performance of AutoML with Dataset 1 in previous study [5].

Dataset 1	**AutoML Binary Classification**		
Subdataset Name	**Accuracy**	**F1**	**Top Classifier**
APK QARK	0.91	0.90	RF
Source QARK	0.91	0.90	RF
All QARK	0.91	0.90	MLP
APK MobSF	0.91	0.90	RF
Source MobSF	0.91	0.90	SVC
All MobSF	0.91	0.90	MLP
APK Combined	0.92	0.91	MLP
Source Combined	0.92	0.90	MLP
All Combined	0.92	0.90	MLP

4.2.2 Performance of GraphCodeBERT

The second model we used in Experiment 1 across the nine subdatasets is GraphCodeBERT. Its performance is presented in Table 4.3. In the QARK subdatasets (APK QARK, Source QARK, and All QARK), the accuracy and F1 of the model ranges between 0.960 and 0.966. The recall ranges between 0.957 and 0.972 while the precision varies between 0.954 and 0.959. Similarly, in the MobSF subdatasets, the best results were recorded with the All MobSF subdatasets where the accuracy and F1 were 0.990 each while the recall and precision were 0.994 and 0.987 respectively. Regarding the Combined subdatasets, the accuracy varies between 0.965 and 0.976 while the recall ranges between 0.966 and 0.981. Similarly, the precision ranges between 0.971 and 0.974 while the F1 varies between 0.965 and 0.976. Overall, the best performance of GraphCodeBERT in Experiment 1 was recorded in the MobSF subdatasets while the worst performance was

recorded in the QARK subdatasets. However, these are still better than the performance (0.92 accuracy, 0.91 F1) of the AutoML models (RF, SVC and MLP) used in the previous study [5] as shown in Table 4.2.

Table 4.3: Vulnerability detection performance of GraphCodeBERT with Dataset 1.

Dataset 1		**GraphCodeBERT Binary Classification**			
Subdataset Name	**Size (MB)**	**Accuracy**	**Recall**	**Precision**	**F1**
APK QARK	305.3	0.966	0.972	0.959	0.966
Source QARK	70.5	0.956	0.957	0.954	0.956
All QARK	373.5	0.960	0.966	0.955	0.96
APK MobSF	242.4	0.992	0.995	0.989	0.992
Source MobSF	77.3	0.983	0.987	0.979	0.983
All MobSF	312.1	0.990	0.994	0.987	0.990
APK Combined	558.9	0.976	0.981	0.971	0.976
Source Combined	111.1	0.965	0.966	0.964	0.965
All Combined	661.1	0.975	0.976	0.974	0.975

4.2.3 Performance of BERT

The third model used in Experiment 1 is BERT. Its performance is presented in Table 4.4. In the QARK subdatasets (APK QARK, Source QARK, and All QARK), the accuracy, precision and F1 of the model ranges between 0.950 and 0.959 while the recall varies between 0.943 and 0.959. However, in the Combined subdatasets, the accuracy ranges between 0.965 and 0.971. The precision ranges between 0.966 and 0.973 while the recall ranges between 0.63 and 0.970. Despite being a natural language-based LLM, BERT shows a good performance which exceeds that of the AutoML models used in the previous study [5] as shown in Table 4.2. Having confirmed that BERT can be optimized through fine-tuning to perform the downstream task of vulnerability detection, it is excluded from the remaining

two experiments (Experiment 2 and Experiment 3) where the main focus of our study is the two code LLMs (CodeBERT and GraphCodeBERT).

Table 4.4: Vulnerability detection performance of BERT with Dataset 1.

Dataset 1	**BERT Binary Classification**				
Subdataset Name	**Size (MB)**	**Accuracy**	**Recall**	**Precision**	**F1**
APK QARK	305.3	0.959	0.955	0.963	0.959
Source QARK	70.5	0.950	0.943	0.953	0.948
All QARK	373.5	0.956	0.959	0.953	0.956
APK MobSF	242.4	0.992	0.995	0.989	0.992
Source MobSF	77.3	0.982	0.987	0.977	0.982
All MobSF	312.1	0.989	0.993	0.985	0.989
APK Combined	558.9	0.969	0.969	0.969	0.969
Source Combined	111.1	0.965	0.963	0.966	0.964
All Combined	661.1	0.971	0.970	0.973	0.971

4.3 Experiment 2

In Experiment 2, three subdatasets (APK Combined, Source Combined and All Combined) from Dataset 2 were used in fine-tuning the two selected code LLMs. This is to allow a common level of comparison with the previous study [5] where only these three subdatasets were used in training the AutoML models. Generally, the two code LLMs performed well in the detection of vulnerabilities across the three subdatasets. Based on the four metrics used in Experiment 2, the highest recorded performances (0.98 accuracy, 0.97 recall, 0.98 precision, 0.98 F1) exceed the best performances of the AutoML model (RF) used in the previous work (0.93 accuracy, 0.92 F1).

4.3.1 Performance of CodeBERT

The performance of CodeBERT on vulnerabilities detection across each of the three subdatasets (APK Combined, Source Combined and All Combined) in Dataset 2 are shown in Table 4.5. In the APK Combined subdataset, 0.975 was recorded for the accuracy, recall and F1 while the precision was 0.976. In the Source Combined subdataset, 0.970 was recorded for the recall while 0.967 was recorded for accuracy and F1. The precision was 0.964. Finally, in the All Combined subdataset, 0.973 was recorded for accuracy, recall, precision and F1. In summary, based on the highest recorded metric (0.975 for accuracy, recall, F1, and 0.976 precision) CodeBERT outperformed the AutoML model (RF 0.93 accuracy, 0.92 F1) used in the previous [5] study as shown in Table 4.6.

Table 4.5: Vulnerability detection performance of CodeBERT with Dataset 2.

Dataset 2	**CodeBERT Binary Classification**			
Subdataset Name	**Size (MB)**	**Accuracy**	**Recall**	**Precision**	**F1**
APK Combined	1560.7	0.975	0.975	0.976	0.975
Source Combined	538.6	0.967	0.970	0.964	0.967
All Combined	1864.7	0.973	0.974	0.973	0.973

Table 4.6: Performance of AutoML with Dataset 2 in previous study [5].

Dataset 2	**AutoML Binary Classification**		
Subdataset Name	**Accuracy**	**F1**	**Top Classifier**
APK Combined	0.93	0.92	RF
Source Combined	0.93	0.91	RF
All Combined	0.93	0.91	RF

4.3.2 Performance of GraphCodeBERT

The performance of GraphCodeBERT on vulnerabilities detection across each of the three subdatasets (APK Combined, Source Combined and All Combined) in Dataset 2 is shown in Table 4.7. In the APK Combined subdataset, we recorded 0.978 accuracy, 0.975 recall, 0.980 precision and 0.977 F1. In the Source Combined subdataset, 0.966 was recorded for the accuracy and F1. Precision was 0.969 while the recall was 0.964. Finally, in the All Combined subdataset, the 0.974 was recorded for accuracy and F1 while the precision and recall were 0.976 and 0.973 respectively. In summary, based on the highest recorded performance metric (0.978 accuracy, 0.975 recall, 0.980 precision and 0.977 F1), CodeBERT exceeds the AutoML model (RF 0.93 accuracy, 0.92 F1) used in the previous [5] study as shown in Table 4.6.

Table 4.7: Vulnerability detection performance of GraphCodeBERT with Dataset 2.

Dataset 2		**GraphCodeBERT Binary Classification**			
Subdataset Name	**Size (MB)**	**Accuracy**	**Recall**	**Precision**	**F1**
APK Combined	1560.7	0.978	0.975	0.980	0.977
Source Combined	538.6	0.966	0.964	0.969	0.966
All Combined	1864.7	0.974	0.973	0.976	0.974

4.4 Experiment 3

In Experiment 3, the APK Combined subdataset from Dataset 3 was used in fine-tuning the two selected code LLMs (CodeBERT and GraphCodeBERT). This is the only subdataset that was used in the previous work [5] and released to the public by the producer of the dataset. Generally, the two code LLMs performed well in the detection of vulnerabilities in Dataset 3. Based on the four metrics used in the experiment, the highest recorded performances (0.966 accuracy, 0.960 recall, 0.972 precision, 0.966 F1) exceed the best performances of the AutoML model (RF) used in the previous work (0.94 accuracy, 0.94

F1). Note that while we report our metrics using three significant digits, the work in [5] reported to only two significant digits.

4.4.1 Performance of CodeBERT

The performance of CodeBERT for the vulnerabilities detection task on Dataset 3 is shown in Table 4.8. Overall, 0.966 was recorded for accuracy, 0.960 for recall, 0.972 for precision and 0.966 F1. This result is again better than the performance of the AutoML model (RF 0.94 accuracy, 0.94 F1) used in the previous study [5] as presented in Table 4.9.

Table 4.8: Vulnerability detection performance of CodeBERT with Dataset 3.

Dataset 3		**CodeBERT Binary Classification**			
Subdataset Name	**Size (MB)**	**Accuracy**	**Recall**	**Precision**	**F1**
APK Combined	2220.4	0.966	0.960	0.972	0.966

Table 4.9: Performance of AutoML with Dataset 3 in previous study [5].

Dataset 3	**AutoML Binary Classification**		
Subdataset Name	**Accuracy**	**F1**	**Top Classifier**
APK Combined	0.94	0.94	RF

4.4.2 Performance of GraphCodeBERT

The performance of GraphCodeBERT on the vulnerabilities detection task on Dataset 3 is shown in Table 4.10. 0.966 was recorded for the accuracy, 0.972 for precision, 0.959 for recall and 0.966 for F1. This result is better than the performance of the AutoML model (RF 0.94 accuracy, 0.94 F1) used in the previous study [5] as shown in Table 4.9.

Table 4.10: Vulnerability detection performance of GraphCodeBERT with Dataset 3.

Dataset 3		**GraphCodeBERT Binary Classification**			
Subdataset Name	**Size (MB)**	**Accuracy**	**Recall**	**Precision**	**F1**
APK Combined	2220.4	0.966	0.959	0.972	0.966

Chapter 5

Discussion

The research objectives (RO) that we set to achieve are:

- **RO1:** Determine the effectiveness of a LLM approach in software vulnerability detection using Android source code as a case study.
- **RO2:** Find out the extent to which a LLM approach is better than the conventional ML/DL approaches.
- **RO3:** Determine the extent to which code-based LLMs are better than natural language-based LLMs in vulnerability detection.
- **RO4**: Ascertain whether pure code-based LLMs are better than graph-based code LLMs for vulnerability detection.
- **RO5:** Collectively utilize the outcome of the study to recommend how code-based LLMs can be used to improve software security through the detection of vulnerabilities in source code.

To achieve these, we conducted three experiments where we fine-tuned two pre-trained code LLMs (CodeBERT and GraphCodeBERT) and one natural language LLM (BERT) with nine different Android source code vulnerability datasets. Overall, all of the selected LLMs performed better than the AutoML approach [5]. In discussing the performance of the models, we will revisit the four metrics (accuracy, recall, precision, F1) presented earlier in Chapter 3.

- **Accuracy:** Accuracy is usually used for preliminary assessment of the model's performance. Because of its simplicity, accuracy is easy to understand. Hence it can be used to explain a model performance to non-technical people such as business leaders and non-technical audience. Accuracy also makes more sense if the datasets utilized in the experiments are already well balanced but not very useful if otherwise [93, 94].

- **Precision:** Precision focuses on the reduction of false positives. The fewer false positives, the more precise the model. In vulnerability detection, because a false positive implies wrongly identifying safe code as vulnerable code, low precision could lead to waste of time and resources for the technical team. This is because additional manual inspection or use of complementary scanning tools by subject matter experts may be required to validate the result. Consequently, this could lead to end users and developers losing confidence in a model.

- **Recall:** The goal of recall is to identify as many actual vulnerabilities as possible in the entire code base. In other words, detection systems (models) with a good recall must have very low false negatives. This is desirable because the consequences of classifying a vulnerable sample as safe might be very costly. For instance, in a vulnerability scanning and detection system used for protecting a city energy grid, if a model wrongly classified vulnerable code as safe code, the presence of the vulnerable code might adversely impact the normal operation of an energy grid system serving millions of people if bad actors were to take advantage of the vulnerability.

- **F1:** F1 is recommended as a metric when there is a need to balance precision with recall thereby ensuring a more reliable vulnerability detection model. In other words, F1 is ideal in situations where the cost of wrongly identifying a safe code segment as being vulnerable is as important as missing an actual vulnerable code sample.

5.1 Which metric is the most important?

Which metric is most important depends on the context, use case or the situation where the vulnerability detection model is deployed. For example, if the model is deployed to detect vulnerabilities in a game played for fun by teenagers, low precision (high false positives) may not be a significant concern. However, if the model is deployed to catch vulnerabilities in mission critical software used in a satellite rocket, even a single false positive detection might cause a launch to be suspended temporarily such that staff and resources are deployed to investigate the mis-identified vulnerability. The time, effort and resources wasted to scrutinize the false detection may be very expensive in this scenario. Similarly, with recall, if the model is deployed to identify vulnerabilities in military software designed to monitor incoming ballistic missiles from an enemy nation, a low recall (high false negative) implies that a real vulnerability might be wrongly classified as safe. Consequently, this unseen vulnerability might be used by threat actors to maliciously attack the normal operation of the military application. In essence, recall is very important if the goal is to identify as many actual vulnerabilities as possible existing in the software system because missing any could be very costly or disastrous. However, the use of F1 helps to achieve a balanced trade-off for both precision and recall. In the previous work [5], Senanayake *et al.* used only two metrics (accuracy, F1) to evaluate the performance of the classical machine learning models (SVM, RF, MLP) in the detection of vulnerabilities. Senanayake *et al.* noted that minimizing both false positives and false negatives is important in enhancing the performance of any machine learning model but reducing false negatives is more critical in a vulnerability detection task [4]. Other researchers including Schaad and Binder used three metrics (precision, recall and F1) in their vulnerability detection model [5].

5.2 Why we used four metrics

In this study, we believe using all four metrics will give better insights regarding the performances of the models. Because it is easier to understand, accuracy can be used to

provide a high-level explanation of the model's performance to a non-technical audience like senior managers or customers. However, for a security analyst in charge of the deployment and monitoring of a vulnerability detection model, recall, precision and F1 are crucial. A model with very high recall implies that it will detect most of the vulnerabilities present in a software system. Also, a model with very high precision will not produce many false alerts (false positives) which could lead to wastage of resources. Furthermore, the inclusion of F1 in our evaluation helps to provide a balanced assessment regarding which model performs best in minimizing both false positives and false negatives (undetected actual vulnerabilities).

5.3 Relationship between metrics and performance

Metrics are needed to quantitatively express the effectiveness of a model in the execution of a task. The optimal performance is a reflection of two metrics (precision and recall) because the harmonic mean of these two metrics determines the F1 [40]. Table 5.1 shows a simple rubric we have derived to explain this concept. The table shows that regardless of the accuracy values, our assessment of the performance of a vulnerability detection model mainly relies on its F1 value, which in turn depends on the precision and recall. For instance, if the accuracy is high but precision and recall are both low, the F1 will automatically be low. In this case, we assess the overall performance of the model as poor.

Table 5.1: Our description rubric of the relationship between metrics and performance.

	Metrics				**Performance**
Model	**Accuracy**	**Precision**	**Recall**	**F1**	**Summary**
A	High	Low	Low	Low	Very Poor
B	High	Low	High	Medium	Fair
C	High	High	Low	Medium	Fair
D	High	High	High	High	Excellent
G	Low	High	High	High	Excellent
E	Low	High	Low	Medium	Fair
F	Low	High	Low	Medium	Fair
G	Low	Medium	Low	Low	Poor
H	Low	Low	Medium	Low	Poor
I	Low	Medium	Medium	Medium	Fair
K	Low	Low	Low	Low	Worst

5.4 Discussing Metrics Variation of the Models

Having provided some insights into the desired performance of a vulnerability detection model, we next discuss the model's metrics variations across the different datasets and how they compare with one another.

5.4.1 Discussing the Metrics Variation of CodeBERT (Observation 1)

In Experiment 1, CodeBERT was fine-tuned on nine subdatasets (APK QARK, Source QARK, All QARK, APK MobSF, Source MobSF, ALL MobSF, APK Combined, Source Combined and All Combined). As explained earlier in Subsection 3.3.2, these subdatasets of Dataset 1 contain various Android source code samples already labelled as either safe or vulnerable. Figure 5.1 shows CodeBERT's metrics variation across the subdatasets in Experiment 1.

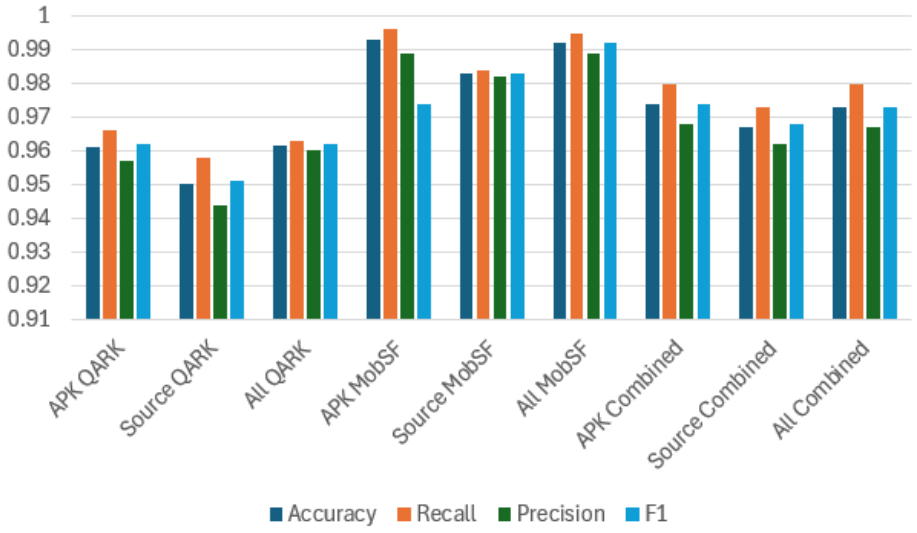

Figure 5.1: CodeBERT's metrics variation with subdatasets of Dataset 1.

Observation 1a: When utilizing the QARK subdatasets, some interesting observations were noted. First, the performance of the CodeBERT model was higher for the APK QARK subdataset (0.961 accuracy, 0.966 recall, 0.957 precision, 0.962 F1) than the Source QARK subdataset (0.950 accuracy, 0.958 recall, 0.944 precision, 0.951 F1). This implies that CodeBERT was better in the detection of vulnerabilities in the compiled version (non-human-readable) of the code (APK) than the source file (Source) which are human-readable. However, the maximum metric variation between the APK QARK and Source QARK was only 1.1%.

Observation 1b: We noted that the best for CodeBERT (0.993 accuracy, 0.996 recall, 0.989 precision, 0.992 F1) were recorded for the APK MobSF subdatasets. This also implies that CodeBERT was better at detecting vulnerabilities in the compiled version of code (APK) than source file. The maximum metric variations seen over the APK MobSF, Source MobSF, and All MobSF subdataset was only 1.2%. However, we observed about 2% and 3% improvement in the metrics (F1) of CodeBERT in the MobSF subdataset compared to the QARK subdataset.

Observation 1c: Regarding the Combined subdataset, the performance of CodeBERT with the APK Combined subdataset (0.974 accuracy, 0.980 recall, 0.968 precision, 0.974 F1) and the All Combined subdataset (0.973 accuracy, 0.980 recall, 0.967 precision, 0.973 F1) were very close but slightly better than with the Source Combined subdataset (0.967 accuracy, 0.973 recall, 0.962 precision, 0.968 F1). Nevertheless, compared to the QARK subdataset, there was about 1% improvement in CodeBERT's metrics with the Combined subdataset.

Summary: CodeBERT's performance was highest with the MobSF subdataset (APK MobSF, Source MobSF and All MobSF) and lowest with the APK subdataset (APK QARK, Source QARK and All QARK). Overall, CodeBERT performed slightly better in the detection of vulnerabilities in the APK subdataset (APK QARK, APK MobSF and APK Combined) than the Source subdataset (Source QARK, Source MobSF and All MobSF). We believe CodeBERT's transformer attention mechanism assisted by its MLM and RTD pre-training technique (discussed earlier in Subsection 2.6.2) enabled it to be effective in identifying vulnerable code samples better in the compiled Android code samples found in the APK subdataset.

5.4.2 Discussing the Metrics Variation of GraphCodeBERT (Observation 2)

In Experiment 1, GraphCodeBERT was fine-tuned with nine subdatasets (APK QARK, Source QARK, All QARK, APK MobSF, Source MobSF, ALL MobSF, APK Combined, Source Combined and All Combined). Figure 5.2 shows GraphCodeBERT's metrics variation across the subdatasets in Experiment 1.

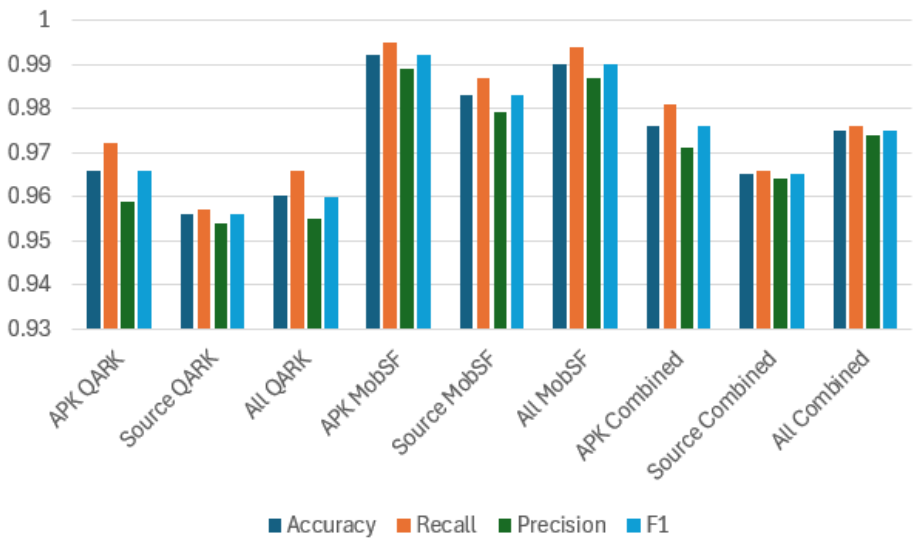

Figure 5.2: GraphCodeBERT's metrics variation with subdatasets of Dataset 1.

- **Observation 2a:** Similar to the observations with CodeBERT, the performance of GraphCodeBERT in the APK QARK subdataset (0.966 accuracy, 0.972 recall, 0.959 precision 0.966 F1) was slightly higher than the Source QARK subdataset (0.956 accuracy, 0.957 recall, 0.954 precision, 0.956 F1). Like the case with CodeBERT, we observed about 1% metrics variation of GraphCodeBERT in each of the QARK, MobSF and Combined subdatasets.

- **Observation 2b**: The highest performances of GraphCodeBERT were seen with the MobSF subdatasets. For instance, in the APK MobSF and All MobSF subdatasets, we recorded 0.992 accuracy, 0.995 recall, 0.989 precision and 0.992 F1. Similar to CodeBERT, we observed about 2% and 3% improvement in the metrics of GraphCodeBERT in the MobSF subdatasets compared to the QARK subdatasets.

- **Observation 2c**: In the Combined subdataset, GraphCodeBERT's performance was higher in the APK Combined (0.976 accuracy, 0.981 recall, 0.971 precision, 0.976 F1) than the Source Combined (0.965 accuracy, 0.966 recall, 0.964 precision, 0.965 F1). However, in the All Combined, the performance improved to 0.975 accuracy, 0.976 recall, 0.974 precision, 0.975 F1. Compared to the QARK subdatasets,

we observed about 1% improvement in the metrics of GraphCodeBERT with the Combined subdatasets.

- **Summary**: GraphCodeBERT's performance was highest with the MobSF subdatasets (APK MobSF, Source MobSF and All MobSF) and lowest with the QARK subdatasets (APK QARK, Source QARK and All QARK). Our hypothesis is that this is due to the GraphCodeBERT's transformer attention mechanism and data flow features discussed earlier in Subsection 2.6.3. However, since this performance is only slightly better than CodeBERT's performance, the impact of the data flow feature is not very significant.

5.4.3 Discussing the Metrics Variation of BERT (Observation 3)

In Experiment 1, BERT was fine-tuned on nine subdatasets (APK QARK, Source QARK, All QARK, APK MobSF, Source MobSF, ALL MobSF, APK Combined, Source Combined and All Combined). Figure 5.3 shows BERT's metrics variation across the subdatasets in Experiment 1.

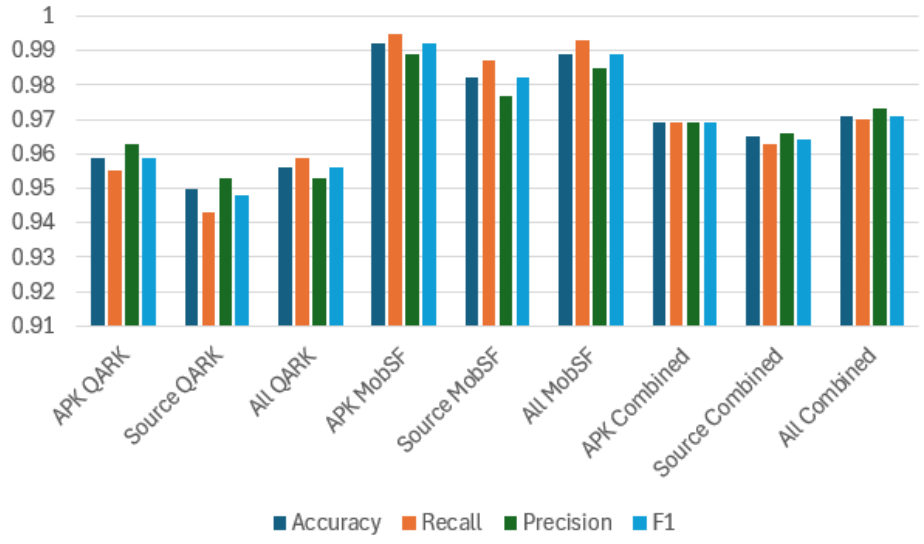

Figure 5.3: BERT's metrics variation with subdatasets of Dataset 1.

Like the observations with CodeBERT and GraphCodeBERT, the performance of BERT in the APK QARK subdataset (0.959 accuracy, 0.955 recall, 0.963 precision and 0.959

F1) was slightly higher than the Source QARK subdataset (0.950 accuracy, 0.943 recall, 0.953 precision and 0.948 F1). However, in most of the observed cases, the variation in performances is limited to about 1%.

5.4.4 Discussing the Metrics Variation of CodeBERT (Observation 4)

In Experiment 2, CodeBERT was fine-tuned on three subdatasets (APK Combined, Source Combined and All Combined). The results for the precision metric varied the most, although only a variation of 1.2% . Figure 5.4 shows CodeBERT's metric variation with the subdatasets in Experiment 2.

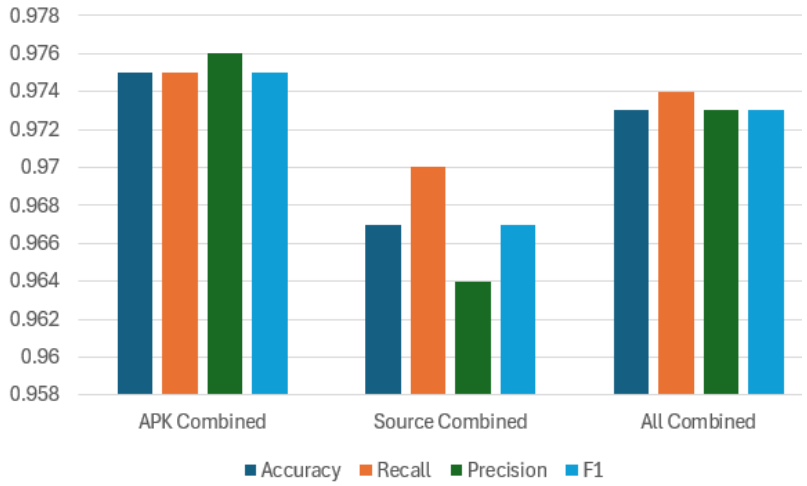

Figure 5.4: CodeBERT's metrics variation with subdatasets of Dataset 2.

- **Observation 4:** The performance of CodeBERT with the APK Combined subdataset (0.975 accuracy, 0.975 recall, 0.976 precision and 0.975 F1) was better than with the Source Combined subdataset (0.967 accuracy, 0.97 recall, 0.964 precision and 0.967 F1) and the All Combined (0.973 accuracy, 0.974 recall, 0.973 precision and 0.973 F1). However, the metrics variation was limited to a maximum of 1.2%.

- **Summary**: CodeBERT's performance was highest with the APK Combined subdataset, followed by the All Combined and the Source Combined subdatasets respectively. We believe this is due to CodeBERT's transformer attention feature,

MLM and RTD features, which enabled it to identify vulnerabilities better in the APK Combined subdataset.

5.4.5 Discussing the Performance of GraphCodeBERT (Observation 5)

In Experiment 2, GraphCodeBERT was fine-tuned on three subdatasets (APK Combined, Source Combined and All Combined). Figure 5.5 shows the metric variation of GraphCodeBERT with the subdatasets in Experiment 2.

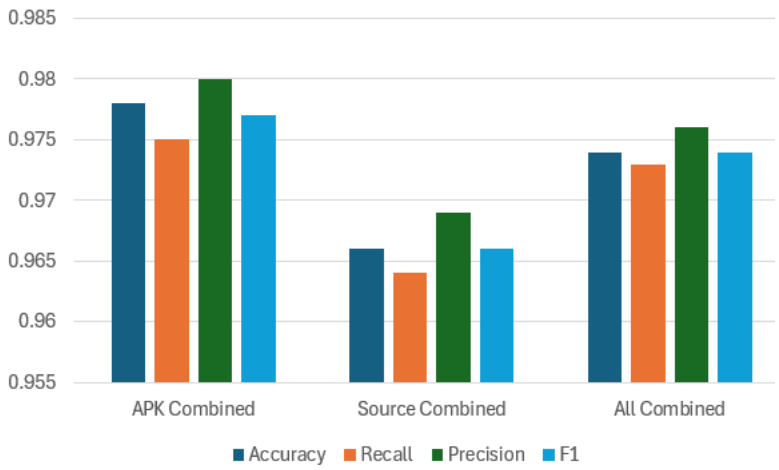

Figure 5.5: GraphCodeBERT's metrics variation with subdatasets of Dataset 2.

- **Observation 5:** Similar to the observations with CodeBERT, the performance of GraphCodeBERT was highest for the APK Combined subdataset (0.978 accuracy, 0.975 recall, 0.98 precision and 0.977 F1). This is followed by the All Combined (0.974 accuracy, 0.973 recall, 0.976 precision and 0.974 F1) and the Source Combined respectively (0.966 accuracy, 0.964 recall, 0.980 precision and 0.977 F1). The metrics variation of GraphCodeBERT across the subdataset is limited to a maximum of 1.2%.

- **Summary**: GraphCodeBERT's performance was highest with the APK Combined subdataset, followed by the All Combined and the Source Combined subdatasets respectively. This is also very similar to what we observed with CodeBERT and is also likely due to the transformer attention feature (present in both models). However,

we believe that the slight improvement in GraphCodeBERT's performance was due to its data flow feature.

5.4.6 Discussing the Performance of CodeBERT (Observation 6)

In Experiment 3, CodeBERT was fine-tuned on Dataset 3 which contains only the APK Combined subdataset. Figure 5.6 shows the metric variation of CodeBERT with the APK Combined subdataset in Experiment 3.

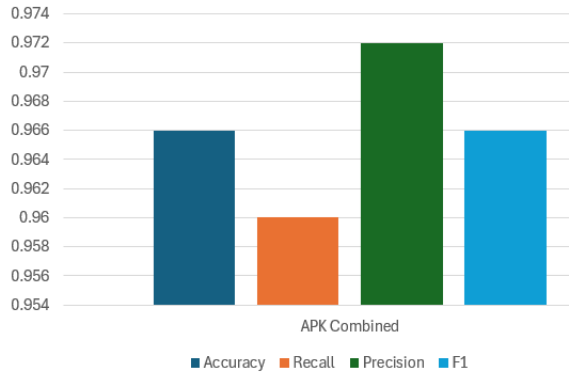

Figure 5.6: CodeBERT's metrics variation with Dataset 3.

- **Observation 6:** In the final experiment with CodeBERT, we noticed a drop in the recall value to 0.960 while the accuracy, precision and F1 were slightly higher at 0.966, 0.972 and 0.966 respectively.

- **Summary**: Precision was the highest metric observed in this experiment, likely because CodeBERT recorded more true positives than false positives due to its transformer attention, MLM and RTD features.

5.4.7 Discussing the Performance of GraphCodeBERT (Observation 7)

In Experiment 3, GraphCodeBERT was fine-tuned on Dataset 3 which contains only the APK Combined subdataset. Figure 5.7 shows the metric variation of GraphCodeBERT with the APK Combined subdataset in Experiment 3.

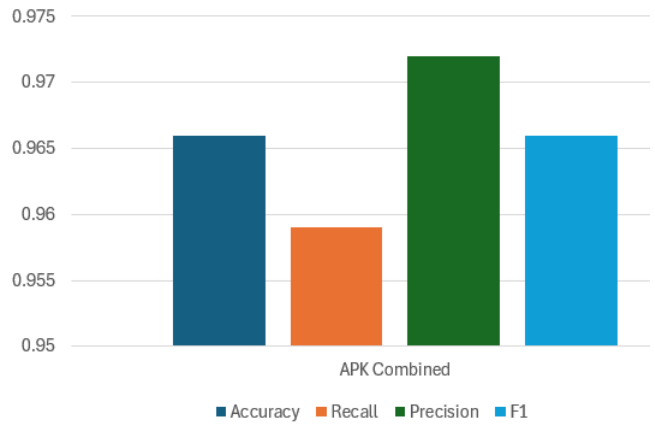

Figure 5.7: GraphCodeBERT's metrics variation with Dataset 3.

- **Observation 7:** In the last experiment with GraphCodeBERT, we noticed a drop in recall (0.959). The accuracy and F1 were both 0.966 while the precision was 0.972.
- **Summary**: Again, the highest metric for this experiment was precision, likely due to GraphCodeBERT's transformer attention and data flow features.

5.5 Summary of All Observations

In this section, we compare the performance of GraphCodeBERT, CodeBERT and BERT based on the observations recorded in this study. The details are presented in Figure 5.8, Figure 5.9, Figure 5.10, Figure 5.11 and Figure 5.12. The yellow, purple and light blue legends indicate when each of the two models' performance is better, worse, or equal to each other, respectively.

5.5.1 Experiment 1

Figure 5.8 shows a summary of the Comparison between the performances of GraphCodeBERT and BERT. Figure 5.9 shows the Comparison between the performances of CodeBERT and BERT. Figure 5.10 shows the Comparison between the performances of CodeBERT and GraphCodeBERT.

As shown in Figure 5.8, GraphCodeBERT outperformed BERT 27 times while BERT outperformed GraphCodeBERT twice. The two models performed the same 6 times (mostly on the APK MobSF subdataset).

Experiment 1								
	GraphCodeBERT				BERT			
Dataset 1	Accuracy	Recall	Precision	F1	Accuracy	Recall	Precision	F1
APK QARK	0.966	0.972	0.959	0.966	0.959	0.955	0.963	0.959
Source QARK	0.956	0.957	0.954	0.956	0.950	0.943	0.953	0.948
All QARK	0.9603	0.966	0.955	0.960	0.956	0.959	0.953	0.956
APK MOBSF	0.992	0.995	0.989	0.992	0.992	0.995	0.989	0.992
Source MOBSF	0.983	0.987	0.979	0.983	0.982	0.987	0.977	0.982
All MOBSF	0.990	0.994	0.987	0.990	0.989	0.993	0.985	0.989
APK Combined	0.976	0.981	0.971	0.976	0.969	0.969	0.969	0.969
Source Combined	0.965	0.966	0.964	0.965	0.965	0.963	0.966	0.964
All Combined	0.975	0.976	0.974	0.975	0.971	0.970	0.973	0.971

Performance Summary			
Model	**Better**	**Worse**	**Draw**
GraphCodeBERT	**28**	**2**	**6**
BERT	**2**	**28**	**6**

Figure 5.8: Comparison of the performances of BERT and GraphCodeBERT.

As shown in Figure 5.9, CodeBERT outperformed BERT 27 times while BERT outperformed CodeBERT 6 times. The two models performed the same 3 times (twice with the APK MobSF subdataset and once with the Source QARK subdataset).

Experiment 1								
	CodeBERT				BERT			
Dataset 1	Accuracy	Recall	Precision	F1	Accuracy	Recall	Precision	F1
APK QARK	0.961	0.966	0.957	0.962	0.959	0.955	0.963	0.959
Source QARK	0.950	0.958	0.944	0.951	0.950	0.943	0.953	0.948
All QARK	0.9617	0.963	0.960	0.962	0.956	0.959	0.953	0.956
APK MOBSF	0.993	0.996	0.989	0.992	0.992	0.995	0.989	0.992
Source MOBSF	0.983	0.984	0.982	0.983	0.982	0.987	0.977	0.982
All MOBSF	0.992	0.995	0.989	0.992	0.989	0.993	0.985	0.989
APK Combined	0.974	0.980	0.968	0.974	0.969	0.969	0.969	0.969
Source Combined	0.967	0.973	0.962	0.968	0.965	0.963	0.966	0.964
All Combined	0.973	0.980	0.967	0.973	0.971	0.970	0.973	0.971

Performance Summary			
Model	**Better**	**Worse**	**Draw**
CodeBERT	**27**	**6**	**3**
BERT	**6**	**27**	**3**

Figure 5.9: Comparison of the performances of BERT and CodeBERT.

As shown in Figure 5.10, GraphCodeBERT outperformed CodeBERT 17 times while CodeBERT outperformed GraphCodeBERT 15 times. Both performed the same 4 times. It is worth nothing that GraphCodeBERT consistently outperformed CodeBERT in all four metrics for the APK QARK subdataset, and the APK Combined subdataset. This suggests that GraphCodeBERT is more sensitive to APK subdatasets which contain mainly compiled Android code samples. It seems that the GraphCodeBERT's data flow feature enables it to identify vulnerable code samples better in this APK subdataset (APK QARK and APK Combined). On the other hand, CodeBERT outperformed GraphCodeBERT in all four metrics for the All MobSF subdataset. It appears that the data flow feature of GraphCodeBERT did not provide an advantage over CodeBERT in the All MobSF subdataset.

Experiment 1

	CodeBERT				GraphCodeBERT			
Dataset 1	Accuracy	Recall	Precision	F1	Accuracy	Recall	Precision	F1
APK QARK	0.961	0.966	0.957	0.962	0.966	0.972	0.959	0.966
Source QARK	0.950	0.958	0.944	0.951	0.956	0.957	0.954	0.956
All QARK	0.9617	0.963	0.960	0.962	0.9603	0.966	0.955	0.960
APK MOBSF	0.993	0.996	0.989	0.992	0.992	0.995	0.989	0.992
Source MOBSF	0.983	0.984	0.982	0.983	0.983	0.987	0.979	0.983
All MOBSF	0.992	0.995	0.989	0.992	0.990	0.994	0.987	0.990
APK Combined	0.974	0.980	0.968	0.974	0.976	0.981	0.971	0.976
Source Combined	0.967	0.973	0.962	0.968	0.965	0.966	0.964	0.965
All Combined	0.973	0.980	0.967	0.973	0.975	0.976	0.974	0.975

Performance Summary

Model	Better	Worse	Draw
GraphCodeBERT	17	15	4
CodeBERT	15	17	4

Figure 5.10: Comparison of the performances of CodeBERT and GraphCodeBERT.

5.5.2 Experiment 2

As shown in Figure 5.11, GraphCodeBERT outperformed CodeBERT 7 times while CodeBERT outperformed GraphCodeBERT 4 times. Both performed the same once. The most consistent performance increase was for the APK Combined subdataset and the All Combined subdataset.

Experiment 2

	CodeBERT				GraphCodeBERT			
Dataset 2	Accuracy	Recall	Precision	F1	Accuracy	Recall	Precision	F1
APK Combined	0.975	0.975	0.976	0.975	0.978	0.975	0.980	0.977
Source Combined	0.967	0.970	0.964	0.967	0.966	0.964	0.969	0.966
All Combined	0.973	0.974	0.973	0.973	0.974	0.973	0.976	0.974

Performance Summary

Model	Better	Worse	Draw
GraphCodeBERT	7	4	1
BERT	4	7	1

Figure 5.11: Comparison of the performances of CodeBERT and GraphCodeBERT.

5.5.3 Experiment 3

Figure 5.12 shows the summary of the Comparison between the performances of CodeBERT and GraphCodeBERT. Both GraphCodeBERT and CodeBERT performed the same 3 times while CodeBERT outperformed GraphCodeBERT once. However, the difference was only 0.01%.

Experiment 3

	CodeBERT				GraphCodeBERT			
Dataset 3	Accuracy	Recall	Precision	F1	Accuracy	Recall	Precision	F1
APK Combined	0.966	0.960	0.972	0.966	0.966	0.959	0.972	0.966

Performance Summary

Model	Better	Worse	Draw
GraphCodeBERT	0	1	3
CodeBERT	1	0	3

Figure 5.12: Comparison of the performances of GraphCodeBERT and CodeBERT.

Based on the results obtained in Chapter 4, all three LLMs utilized in this study performed significantly better than the conventional ML/DL models used in the previous study [5]. An increase of up to about 8% at the maximum was achieved. We had expected that the graph-based (data flow) feature of GraphCodeBERT would provide a significant performance improvement over CodeBERT, but the results showed that this is not the case as there was only improvement of up to 1%. Since both CodeBERT and GraphCodeBERT are primarily based on the attention architecture of transformers, we believe that the impact of this feature is more significant than the additional graph data flow feature of GraphCodeBERT in this task. However, in future studies, we would like to further explore why the graph-based feature did not provide larger improvements for GraphCodeBERT in this type of code classification task [5].

5.5.4 Does the 1% Performance Variation Really Matter?

Does this 1% variation in performance really matter? The answer is yes! This is because software security is a continuous improvement process. In actual practice, 1% improvement translates to a capability to identify an additional 1% of vulnerable code samples. It could also translate to 1% reduction of false alarms which would mean fewer unnecessary security incidents for software security teams, thereby allowing them to focus on more pressing incidents. Over millions of code samples used in the real world, a 1% increase in effectiveness would be very significant.

5.6 Comparing code LLMs to Conventional Scanners and ML Models

We believe that a code LLM vulnerability detection approach also has the potential to detect previously unknown software vulnerabilities better than popular conventional scanners such as Nessus and Qualys which are largely signature-based [95]. Signature-based means such scanners are designed to detect security vulnerabilities based on existing queries or rules already registered in a database [95]. Thus, they are ineffective in detecting any new,

unknown or zero-day vulnerabilities. It is possible that the conventional ML models such as RF, SVC and MLP used in the previous study [5] could detect zero-day vulnerabilities because they are AI approaches which by design are inference-based. Inference-based analysis is typically used to describe AI-based approaches where decisions or conclusions are inferred on unseen data based on training on previous data rather than relying on explicit rules [96]. However, unlike code LLMs, the capability of classical ML models for zero-day vulnerability detection in source code is limited because they lack the attention-based transformer features necessary to fully understand source code syntax, structure and semantics. This means that if a vulnerable code sample is maliciously modified by threat actors, a classical ML model might not fully detect it. Table 5.2 shows the comparison between conventional vulnerability scanners, ML models, and code LLM approach.

Table 5.2: Comparison of code LLMs to Conventional Scanners and ML Models.

Vulnerability Scanner	**Signature-based**	**Inference-based**	**Zero-Day Detection**
Nessus	Yes	Limited	No
Qualys	Yes	Limited	No
ML Models	No	Yes	Future Work
Code LLMs	No	Yes	Future Work

5.7 Assessment of Our Research Objectives

Based on the general assessments of the three experiments in this study, the following is the summary of the outcomes of this study:

- **RO1**: Based on the results of our three experiments, we believe the LLM approach is effective in software vulnerability detection.

- **RO2:** The recorded higher performances (accuracy, precision, recall, F1) of CodeBERT, GraphCodeBERT and BERT compared to the performances of the ML models used in the previous study [5] confirm that the LLM approach performs better. We believe the transformer architecture of the LLMs (especially their attention-based mechanism) allows them to better understand the syntax, semantic and other structural inter-relationships in the source code samples. Also, we believe the fine-tuning of these models with labelled Android code vulnerabilities allows them to outperform their conventional ML/DL models counterparts in vulnerabilities detection.

- **RO3:** The moderately higher performances of CodeBERT and GraphCodeBERT compared to BERT's performance, indicate that code-based LLMs appear to be slightly better than natural language-based LLM (BERT) in the vulnerability detection task. We had expected far less performance for BERT since it was never pre-trained on programming languages like the code LLMs. We assume the fine-tuning with Android vulnerability code samples enabled its performance to be close to the code LLMs. Hence, we believe datasets used for fine-tuning LLMs for classification tasks are as important as the dataset used for their initial pre-training. We also believe the similar attention-based transformer features present in both natural language LLMs and code-based LLMs may also play a role in this outcome. Further study is needed to determine the relationship between these factors (pre-training data, fine-tuning data and transformer features) of both code-based LLMs and natural language-based LLMs.

- **RO4:** We hypothesized that graph-based code LLMs would perform better than pure code-based LLMs in the task of vulnerability detection. However, their performance increase was minimal in our study.

- **RO5**: Overall, we believe code-based LLMs offer promising opportunities in the improvement of software security via the detection of common software code vulnerabilities. Furthermore, because these LLMs can be optimized through fine-tuning to understand unique patterns associated with various types of vulnerable codes, we believe they will be potentially capable of detecting newer vulnerabilities, especially zero-day vulnerabilities that are usually difficult to detect by conventional methods. Further study is required to validate this.

Chapter 6

Conclusion

In this work, our main goal was to improve software security through improved vulnerability detection-based approaches. As a case study, we chose Android software based on the availability of a recently released real-life vulnerabilities dataset - LVDAndro. An additional factor in our selection of a dataset based on Android was the wide adoption of the Android operating system across the world. We believe the outcome of our study may help developers of Android software in improving protection from various security threats, particularly to mobile software. To achieve our aim, we proposed the use of code-based Large Language Models (LLMs) as a better alternative to the classical ML techniques which were adopted in the previous study. We also used the entire range of the subdatasets used in the previous study to provide a common ground of comparison with our approach. To adapt our models to the Android vulnerability detection task, we performed supervised fine-tuning using several labelled subdatasets from the LVDAndro dataset. The trained models were then used to perform vulnerability detection on the test data. To derive more insights into the model's performance, we extended the metrics used in the previous study from two (accuracy, F1) to four (accuracy, recall, precision and F1). Based on our results, our selected LLMs (BERT, CodeBERT, GraphCodeBERT) performed well in the detection of Android source code vulnerabilities. These performances significantly exceed those of the three classical ML models (SVC, RF, MLP) used in the previous study. Furthermore, we also observed that the code LLMs (CodeBERT, GraphCodeBERT) outperformed the natural language LLM (BERT). GraphCodeBERT slightly outperformed CodeBERT in the vulnerability detection

task. However, this performance is marginal contrary to our expectations. We believe the attention-based feature of GraphCodeBERT is more significant than its graph data-flow feature in the detection of patterns in the code samples. Hence, its performance is closer to CodeBERT, which also possesses an attention-based feature. However, further work is needed to fully ascertain this.

6.1 Threat to Validity

Based on our knowledge, we believe we are the first to explore the use of code LLMs on the newly curated Android-based LVDAndro datasets for vulnerability analysis research. Previous studies had largely focused on the popular synthetic datasets such as SARD [22] and the generic C++ based dataset called Big-Vul [23], neither of which are designed for Android vulnerability research. The adoption of the LVDAndro datasets enabled us to compare the code LLM approach to the classical ML models used in the previous study. Although this dataset was produced from real-life Android source code, the effectiveness of our chosen approach is still subject to the extent to which real-world Android vulnerabilities are fully or accurately represented in the datasets.

6.2 Recommendations

Software development security teams could utilize our code LLM approach to enhance their secure coding strategy. Conventional vulnerability scanners such as Nessus and Qualys typically use signature-based vulnerability detection mechanisms which are known to be ineffective for zero-day (new or unknown) vulnerabilities unlike the inference-based approach of code LLMs. Hence, we believe software security analysts could be advised to adopt a hybrid approach where both conventional scanners and code LLM scanners are used together. However, since newer vulnerabilities are continually emerging at a fast rate, the models would require continuous re-training in production environment through the use of software automation and configuration tools such as Ansible [97]. This would also

enable the model to be up to date in detecting newer or previously unknown or zero-day vulnerabilities. To avoid the huge costs associated with re-training the entire model, a triggered re-training approach could be adopted. In this way, re-training would only occur when certain conditions are met such as performance degradation or data drift. Re-training could also be manually triggered in response to the announcement of newer vulnerabilities by established cyber security agencies, software vendors or independent security researchers.

6.3 Future Research Directions

- Only Android based software vulnerabilities were analyzed in this work. We would like to extend the work to Apple iOS vulnerabilities.

- Our current work focused more on the effectiveness of code LLMs in vulnerability detection. In future work, we would be interested in a detailed study of why the graph-based data flow feature of GraphCodeBERT does not provide a significant improvement over the pure attention-based transformer feature of CodeBERT in the task of vulnerable code classification.

- Lastly, we are interested in studying whether code LLMs are more sensitive to the detection of specific types of vulnerabilities among the various CWE and CVE categories in the Android datasets.

Bibliography

[1] K. Kalyan, "A Survey of GPT-3 family Large Language Models including ChatGPT and GPT-4," *Natural Language Processing Journal*, vol. 6, p. 100048, 12 2023.

[2] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang, and M. Zhou, "CodeBERT: A pre-trained model for programming and natural languages," in *Findings of the Association for Computational Linguistics: EMNLP 2020*, T. Cohn, Y. He, and Y. Liu, Eds. Association for Computational Linguistics, Nov. 2020, pp. 1536–1547. [Online]. Available: https://aclanthology.org/2020.findings-emnlp.139/

[3] OWAPS, "OWASP Top Ten 2025," 2025, Accessed: 2025-09-02. [Online]. Available: https://owasp.org/www-project-top-ten/

[4] Forum of Incident Response and Security Teams, "Common vulnerability scoring system v3.1: Specification document," 2019, accessed: 2025-09-01. [Online]. Available: https://www.first.org/cvss/v3-1/specification-document

[5] J. Senanayake, H. Kalutarage, M. O. Al-Kadri, L. Piras, and A. Petrovski, "Labelled Vulnerability Dataset on Android Source Code (LVDAndro) to Develop AI-Based Code Vulnerability Detection Models." SCITEPRESS – Science and Technology Publications, Lda. Under CC license (CC BY-NC-ND 4.0), 2023. [Online]. Available: https://www.scitepress.org/Papers/2023/120604/120604.pdf

[6] J. Senanayake, H. Kalutarage, A. Petrovski, L. Piras, and M. O. Al-Kadri, "Defendroid: Real-time Android code vulnerability detection via blockchain federated neural network with XAI," *Journal of Information Security and Applications*, vol. 82, p. 103741, 03 2024.

[7] Communications Security Establishment Canada, "National cyber threat assessment 2025–2026," Canada Centre for Cyber Security, Tech. Rep., 2024, accessed July 10, 2025. [Online]. Available: https://www.cyber.gc.ca/sites/default/files/national-cyber-threat-assessment-2025-2026-e.pdf

[8] M. Raeini, "The Evolution of Language Models: From N-Grams to LLMs, and Beyond," *SSRN Electronic Journal*, Jan. 1. [Online]. Available: https://papers.ssrn.com/sol3/papers.cfm?abstract_id=4625356

[9] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Kaiser, and I. Polosukhin, "Attention is all you need." Long Beach, California, USA: Curran Associates Inc., 2017, pp. 6000–6010.

[10] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding," 2019. [Online]. Available: https://api.semanticscholar.org/CorpusID:52967399

[11] D. Guo, S. Ren, S. Lu, Z. Feng, D. Tang, S. Liu, L. Zhou, N. Duan, J. Yin, D. Jiang, and M. Zhou, "GraphCodeBERT: Pre-training Code Representations with Data Flow," *ArXiv*, vol. abs/2009.08366, 2020. [Online]. Available: https://api.semanticscholar.org/CorpusID:221761146

[12] T. Misa, "Understanding 'How Computing Has Changed the World'," *Annals of the History of Computing, IEEE*, vol. 29, pp. 52–63, Jan. 11. [Online]. Available: https://ieeexplore.ieee.org/document/4407445/

[13] G. Ekşi, B. Tekinerdogan, and C. Catal, "Software security management in critical infrastructures: a systematicliterature review," *Turkish Journal of Electrical Engineering and Computer Sciences*, vol. 30, pp. 1142–1161, Jan. 5. [Online]. Available: https://journals.tubitak.gov.tr/elektrik/vol30/iss4/1/

[14] J. C. Morris and M. K. Mayer, "Canaries in coal mines and normal accidents: The crowdstrike outage and its lessons for critical infrastructure," *Journal of Critical Infrastructure Policy*, vol. 5, no. 2, pp. 61–64, 2024. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/jci3.12021

[15] A. Bendovschi, "Cyber-Attacks – Trends, Patterns and Security Countermeasures," *Procedia Economics and Finance*, vol. 28, pp. 24–31, 12. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S2212567115010771?via%3Dihub

[16] A. Kokaji and A. Goto, "An analysis of economic losses from cyberattacks: based on input–output model and production function," *Journal of Economic Structures*, vol. 11, 12. [Online]. Available: https://journalofeconomicstructures.springeropen.com/counter/pdf/10.1186/s40008-022-00286-4.pdf

[17] B. O. Omoyiola, "An Overview of Root Causes of Cybersecurity Breaches in Organizations," *SSRN Electronic Journal*, May 2. [Online]. Available: https://papers.ssrn.com/sol3/papers.cfm?abstract_id=4348319

[18] I. Krsul, E. Spafford, and M. Tripunitara, "An Analysis of Some Software Vulnerabilities," *proceedings of the 21st nissc 1998*, 1998. [Online]. Available: https://csrc.nist.gov/files/pubs/conference/1998/10/08/proceedings-of-the-21st-nissc-1998/final/docs/paperd6.pdf

[19] W. Huang, S. Lin, and C. Li, "BBVD: A BERT-based Method for Vulnerability Detection," *International Journal of Advanced Computer Science and Applications*, vol. 13, no. 12, 2022. [Online]. Available: http://dx.doi.org/10.14569/IJACSA.2022.01312103

[20] J. Wu, "Literature review on vulnerability detection using NLP technology," *arXiv e-prints*, p. arXiv:2104.11230, Apr. 2021.

[21] J. Akram and P. Luo, "Sqvdt: A scalable quantitative vulnerability detection technique for source code security assessment," *Software: Practice and Experience*, vol. 51, no. 2, pp. 294–318, 2021. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/spe.2905

[22] National Institute of Standard and Technology, "SARD: A Software Assurance Reference Dataset (SARD)," 2017. [Online]. Available: https://samate.nist.gov/SARD/

[23] J. Fan, Y. Li, S. Wang, and T. N. Nguyen, "A C/C++ Code Vulnerability Dataset with Code Changes and CVE Summaries," in *Proceedings of the 17th International Conference on Mining Software Repositories*, ser. MSR '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 508–512. [Online]. Available: https://doi.org/10.1145/3379597.3387501

[24] Statista, "Android - statistics facts," https://www.statista.com/topics/876/android/#topicOverview, 2024, accessed: 2025-08-22. [Online]. Available: https://www.statista.com/topics/876/android/#topicOverview

[25] S. Hudson, "Not Just For Phones And Tablets: What Other Devices Run Android?" https://www.makeuseof.com/tag/phones-tablets-devices-run-android/, 2014, accesssed: 2025-08-22. [Online]. Available: https://www.makeuseof.com/tag/phones-tablets-devices-run-android/

[26] A. A. Markov, "An example of statistical investigation of the text eugene onegin concerning the connection of samples in chains," *Science in Context*, vol. 19, pp. 591 – 600, 2006. [Online]. Available: https://api.semanticscholar.org/CorpusID:144854176

[27] C. E. Shannon, "A mathematical theory of communication," *The Bell System Technical Journal*, vol. 27, no. 3, pp. 379–423, 1948. [Online]. Available: https://ieeexplore.ieee.org/stampPDF/getPDF.jsp?tp=&arnumber=6773024&ref=

[28] J. Weizenbaum, "ELIZA—a computer program for the study of natural language communication between man and machine," *Commun. ACM*, vol. 9, no. 1, pp. 36–45, 1966. [Online]. Available: https://doi-org.uleth.idm.oclc.org/10.1145/365153.365168

[29] A. Sherstinsky, "Fundamentals of recurrent neural network (RNN) and long short-term memory (LSTM) network," *Physica D: Nonlinear Phenomena*, vol. 404, p. 132306, 2020.

[30] T. Mikolov, K. Chen, G. Corrado, and J. Dean, "Efficient estimation of word representations in vector space," in *1st International Conference on Learning Representations, ICLR 2013, Scottsdale, Arizona, USA, May 2-4, 2013, Workshop Track Proceedings*, Y. Bengio and Y. LeCun, Eds., 2013. [Online]. Available: http://arxiv.org/abs/1301.3781

[31] G. Shen, Q. Tan, H. Zhang, P. Zeng, and J. Xu, "Deep Learning with Gated Recurrent Unit Networks for Financial Sequence Predictions," *Procedia*

Computer Science, vol. 131, pp. 895–903, Jan. 1. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1877050918306781?via%3Dihub

[32] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, "Attention is all you need," in *Proceedings of the 31st International Conference on Neural Information Processing Systems*, ser. NIPS'17. Red Hook, NY, USA: Curran Associates Inc., 2017, p. 6000–6010.

[33] G. Yenduri, R. Murugan, C. Govardanan, Y. Supriya, G. Srivastava, P. Reddy, D. Raj, R. Jhaveri, P. B, W. Wang, A. Vasilakos, and T. Gadekallu, "Gpt (generative pre-trained transformer)— a comprehensive review on enabling technologies, potential applications, emerging challenges, and future directions," *IEEE Access*, vol. PP, pp. 1–1, 01 2024.

[34] E. Shareghi, D. Gerz, I. Vulić, and A. Korhonen, "Show some love to your n-grams: A bit of progress and stronger n-gram language modeling baselines," in *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, J. Burstein, C. Doran, and T. Solorio, Eds. Minneapolis, Minnesota: Association for Computational Linguistics, Jun. 2019, pp. 4113–4118. [Online]. Available: https://aclanthology.org/N19-1417/

[35] D. Jurafsky, *Speech & language processing*. Pearson Education India, 2000.

[36] V. Vianu, "Rule-Based Languages," *Annals of Mathematics and Artificial Intelligence*, vol. 19, 10. [Online]. Available: https://link.springer.com/content/pdf/10.1023/A:1018907806177.pdf

[37] T. Mikolov, K. Chen, G. S. Corrado, and J. Dean, "Efficient estimation of word representations in vector space," in *International Conference on Learning Representations*, 2013. [Online]. Available: https://api.semanticscholar.org/CorpusID:5959482

[38] J. Pennington, R. Socher, and C. Manning, "GloVe: Global vectors for word representation," in *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, A. Moschitti, B. Pang, and W. Daelemans, Eds. Doha, Qatar: Association for Computational Linguistics, Oct. 2014, pp. 1532–1543. [Online]. Available: https://aclanthology.org/D14-1162/

[39] I. Sutskever, "Sequence to Sequence Learning with Neural Networks," *arXiv preprint arXiv:1409.3215*, 2014.

[40] A. Geron, *Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow: Concepts, Tools, and Techniques to Build Intelligent Systems*. O'Reilly Media, Inc., 2019.

[41] J. Mugaanyi, L. Cai, S. Cheng, C. Lu, and J. Huang, "Evaluation of large language model performance and reliability for citations and references in scholarly writing:

Cross-disciplinary study," *J Med Internet Res*, vol. 26, p. e52935, Apr 2024. [Online]. Available: https://www.jmir.org/2024/1/e52935

[42] Jurafsky, Daniel and Martin, James H, *Speech and Language Processing: An Introduction to Natural Language Processing, Computational Linguistics, and Speech Recognition*, 1st ed. USA: Prentice Hall PTR, 2000.

[43] X. Amatriain, A. Sankar, J. Bing, P. K. Bodigutla, T. J. Hazen, and M. Kazi, "Transformer models: An Introduction and Catalog," *arXiv e-prints*, p. arXiv:2302.07730, Feb. 2023.

[44] M. Lewis, "Bart: Denoising sequence-to-sequence pre-training for natural language generation, translation, and comprehension," *arXiv preprint arXiv:1910.13461*, 2019.

[45] M. Lewis, Y. Liu, N. Goyal, M. Ghazvininejad, A. Mohamed, O. Levy, V. Stoyanov, and L. Zettlemoyer, "BART: Denoising Sequence-to-Sequence Pre-training for Natural Language Generation, Translation, and Comprehension," *arXiv e-prints*, p. arXiv:1910.13461, Oct. 2019.

[46] J. Kaplan, S. McCandlish, T. Henighan, T. B. Brown, B. Chess, R. Child, S. Gray, A. Radford, J. Wu, and D. Amodei, "Scaling laws for neural language models," *arXiv preprint arXiv:2001.08361*, 2020.

[47] D.-H. Lee, "Pseudo-label : The simple and efficient semi-supervised learning method for deep neural networks," *ICML 2013 Workshop : Challenges in Representation Learning (WREPL)*, 07 2013.

[48] Y. Wu, I. Hua, and Y. Ding, "Unveiling environmental impacts of large language model serving: A functional unit view," in *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, W. Che, J. Nabende, E. Shutova, and M. T. Pilehvar, Eds. Vienna, Austria: Association for Computational Linguistics, Jul. 2025, pp. 10 560–10 576. [Online]. Available: https://aclanthology.org/2025.acl-long.519/

[49] N. Kandpal and C. Raffel, "Position: The most expensive part of an llm should be its training data," *arXiv preprint arXiv:2504.12427*, 2025.

[50] A. Kanade, P. Maniatis, G. Balakrishnan, and K. Shi, "Learning and evaluating contextual embedding of source code," in *International conference on machine learning*. PMLR, 2020, pp. 5110–5121.

[51] W. U. Ahmad, S. Chakraborty, B. Ray, and K.-W. Chang, "Unified pre-training for program understanding and generation," *arXiv preprint arXiv:2103.06333*, 2021.

[52] M. R. I. Rabin, A. Mukherjee, O. Gnawali, and M. A. Alipour, "Towards demystifying dimensions of source code embeddings," in *Proceedings of the 1st ACM SIGSOFT International Workshop on Representation Learning for Software Engineering and Program Languages*, ser. RL+SEPL 2020. New York, NY,

USA: Association for Computing Machinery, 2020, p. 29–38. [Online]. Available: https://doi.org/10.1145/3416506.3423580

[53] U. Alon, M. Zilberstein, O. Levy, and E. Yahav, "code2vec: Learning Distributed Representations of Code," p. arXiv:1803.09473, Mar. 2018. [Online]. Available: https://ui.adsabs.harvard.edu/abs/2018arXiv180309473Ahttp://arxiv.org/pdf/1803.09473

[54] Uri, Alon and Meital, Zilberstein and Omer, Levy and Eran, Yahav, "code2vec: learning distributed representations of code," *Proc. ACM Program. Lang.*, vol. 3, no. POPL, Jan. 2019. [Online]. Available: https://doi.org/10.1145/3290353

[55] L. Zhuang, L. Wayne, S. Ya, and Z. Jun, "A Robustly Optimized BERT Pre-training Approach with Post-training," in *Proceedings of the 20th Chinese National Conference on Computational Linguistics*, S. Li, M. Sun, Y. Liu, H. Wu, K. Liu, W. Che, S. He, and G. Rao, Eds. Huhhot, China: Chinese Information Processing Society of China, Aug. 2021, pp. 1218–1227. [Online]. Available: https://aclanthology.org/2021.ccl-1.108/

[56] S. N. Pinku, D. Mondal, and C. K. Roy, "On the use of deep learning models for semantic clone detection," in *2024 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2024, pp. 512–524.

[57] C. Pan, M. Lu, and B. Xu, "An empirical study on software defect prediction using codebert model," *Applied Sciences*, vol. 11, no. 11, 2021. [Online]. Available: https://www.mdpi.com/2076-3417/11/11/4793

[58] V. B. Parthasarathy, A. Zafar, A. Khan, and A. Shahid, "The ultimate guide to fine-tuning llms from basics to breakthroughs: An exhaustive review of technologies, research, best practices, applied research challenges and opportunities," *arXiv preprint arXiv:2408.13296*, 2024.

[59] K. Wongsuphasawat, Y. Liu, and J. Heer, "Goals, process, and challenges of exploratory data analysis: An interview study," *arXiv preprint arXiv:1911.00568*, 2019.

[60] F. Zhuang, Z. Qi, K. Duan, D. Xi, Y. Zhu, H. Zhu, H. Xiong, and Q. He, "A comprehensive survey on transfer learning," *Proceedings of the IEEE*, vol. 109, no. 1, pp. 43–76, 2020.

[61] N. Saunshi, S. Malladi, and S. Arora, "A mathematical exploration of why language models help solve downstream tasks," *arXiv preprint arXiv:2010.03648*, 2020.

[62] MITRE Corporation, "CWE Version 4.15 Now Available," 2024, accessed: 2025-6-26. [Online]. Available: https://cwe.mitre.org/news/archives/news2024.html#july16_CWE_Version_4.15_Now_Available

[63] N. V. Database, "Vulnerabilities," 2022, publisher: National Institute of Standards and Technology. [Online].

Available: https://nvd.nist.gov/vuln#:~:text=%22A%20weakness%20in%20the%20computational,/ResourcesSupport/AllResources/CNARules.

[64] I. Bojanova and C. E. C. Galhardo, "Bug, fault, error, or weakness: Demystifying software security vulnerabilities," *IT Professional*, vol. 25, no. 1, pp. 7–12, 2023.

[65] National Institute of Standards and Technology, "Vulnerability Metrics," Tech. Rep., 2022, accessed: 2025-06-26. [Online]. Available: https://nvd.nist.gov/vuln-metrics/cvss

[66] C. Pan, M. Lu, and B. Xu, "An Empirical Study on Software Defect Prediction Using CodeBERT Model," *Applied Sciences*, vol. 11, p. 4793, 5. [Online]. Available: https://mdpi-res.com/d_attachment/applsci/applsci-11-04793/article_deploy/applsci-11-04793-v2.pdf?version=1621848391

[67] P. Mell and T. Grance, "Use of the common vulnerabilities and exposures (cve) vulnerability naming scheme," 2002-09-01 2002, accessed: 2025-03-09. [Online]. Available: https://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-51.pdf

[68] MITRE Corporation, "CWE-327: Use of a Broken or Risky Cryptographic Algorithm," 2025, accessed: 2025-08-12. [Online]. Available: https://cwe.mitre.org/data/definitions/327.html

[69] ——, "CWE-532: Insertion of Sensitive Information into Log File," 2025, accessed: 2025-08-12. [Online]. Available: https://cwe.mitre.org/data/definitions/532

[70] ——, "CWE-749: Exposed Dangerous Method or Function," 2025, accessed: 2025-08-12. [Online]. Available: https://cwe.mitre.org/data/definitions/749.html

[71] Y. Zhou, S. Liu, J. Siow, X. Du, and Y. Liu, "Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks," *Advances in neural information processing systems*, vol. 32, 2019.

[72] M. J. Nelson and A. K. Hoover, "Notes on using google colaboratory in ai education," in *Proceedings of the 2020 ACM Conference on Innovation and Technology in Computer Science Education*, ser. ITiCSE '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 533–534. [Online]. Available: https://doi.org/10.1145/3341525.3393997

[73] Google Research, "Welcome to Colab," https://colab.research.google.com, 2024, accessed:2025-01-01.

[74] T. Wolf, L. Debut, V. Sanh, J. Chaumond, C. Delangue, A. Moi, P. Cistac, T. Rault, R. Louf, M. Funtowicz, J. Davison, S. Shleifer, P. von Platen, C. Ma, Y. Jernite, J. Plu, C. Xu, T. Le Scao, S. Gugger, M. Drame, Q. Lhoest, and A. Rush, "Transformers: State-of-the-art natural language processing," in *Proceedings of the 2020 Conference on Empirical Methods in Natural Language*

Processing: System Demonstrations, Q. Liu and D. Schlangen, Eds. Online: Association for Computational Linguistics, Oct. 2020, pp. 38–45. [Online]. Available: https://aclanthology.org/2020.emnlp-demos.6/

[75] NVIDIA, "NVIDIA A100 Tensor Core GPU," https://www.nvidia.com/en-us/data-center/a100/, 2025, accessed: 2025-05-10. [Online]. Available: https://www.nvidia.com/en-us/data-center/a100/

[76] W. McKinney, "Data Structures for Statistical Computing in Python," in *Proceedings of the 9th Python in Science Conference*, 2010, pp. 56 – 61.

[77] J. Hunter, "Matplotlib: A 2D Graphics Environment," *Computing in Science & Engineering*, vol. 9, pp. 90–95, Jan. 6. [Online]. Available: https://ieeexplore.ieee.org/document/4160265/

[78] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and Duchesnay, "Scikit-learn: Machine Learning in Python," *J. Mach. Learn. Res.*, vol. 12, no. null, pp. 2825–2830, 2011. [Online]. Available: https://dl.acm.org/doi/pdf/10.5555/1953048.2078195

[79] R. Cohen and T. Wang, *Android Application Development Processes and Tool Chains for Intel® Architecture*. Berkeley, CA: Apress, 2014, pp. 47–84. [Online]. Available: https://doi.org/10.1007/978-1-4842-0100-8_3

[80] TechCrunch. (2019) Kotlin is now Google's preferred language for Android app developmen. Accessed on 2025-08-21. [Online]. Available: https://techcrunch.com/2019/05/07/kotlin-is-now-googles-preferred-language-for-android-app-development/

[81] A. Abraham, M. Dobrushin, N. Vincent, and Magaofei, "Mobile Security Framework (MobSF)," {https://github.com/MobSF/Mobile-Security-Framework-MobSF},, 2023, accessed:2025-06-04. [Online]. Available: https://github.com/MobSF/Mobile-Security-Framework-MobSF

[82] T. Tony, "Introducing QARK An Open Source Tool to Improve Android Application Security," 2015, 2025-03-09. [Online]. Available: https://security.linkedin.com/content/security/global/en_us/index/posts/2015/introducing-qark

[83] C. Anwar, C. Sumerli, N. Rahayu, and K. Kraugusteeliana, "The application of mobile security framework (mobsf) and mobile application security testing guide to ensure the security in mobile commerce applications," *Jurnal Sistim Informasi dan Teknologi*, vol. 5, pp. 97–102, 06 2023.

[84] V. Kouliaridis, G. Karopoulos, and G. Kambourakis, "Assessing the security and privacy of android official id wallet apps," *Information*, vol. 14, no. 8, 2023. [Online]. Available: https://www.mdpi.com/2078-2489/14/8/457

[85] P. Thomson, "Static analysis," *Commun. ACM*, vol. 65, no. 1, p. 50–54, Dec. 2021. [Online]. Available: https://doi.org/10.1145/3486592

[86] C. Titiakarawongse, S. Taksin, J. Ruangsawat, K. Deeduangpan, and S. Boonkrong, "Comparative vulnerability analysis of thai and non-thai mobile banking applications," *Journal of Cybersecurity and Privacy*, vol. 4, no. 3, pp. 650–662, 2024. [Online]. Available: https://www.mdpi.com/2624-800X/4/3/31

[87] C. Aliferis and G. Simon, *Overfitting, Underfitting and General Model Overconfidence and Under-Performance Pitfalls and Best Practices in Machine Learning and AI*. Cham: Springer International Publishing, 2024, pp. 477–524. [Online]. Available: https://doi.org/10.1007/978-3-031-39355-6_10

[88] K. Ghosh, C. Bellinger, R. Corizzo, P. Branco, B. Krawczyk, and N. Japkowicz, "The class imbalance problem in deep learning," *Machine Learning*, vol. 113, no. 7, pp. 4845–4901, 2024.

[89] S. Dowlagar and R. Mamidi, "DepressionOne@LT-EDI-ACL2022: Using machine learning with SMOTE and random UnderSampling to detect signs of depression on social media text." in *Proceedings of the Second Workshop on Language Technology for Equality, Diversity and Inclusion*, B. R. Chakravarthi, B. Bharathi, J. P. McCrae, M. Zarrouk, K. Bali, and P. Buitelaar, Eds. Dublin, Ireland: Association for Computational Linguistics, May 2022, pp. 301–305. [Online]. Available: https://aclanthology.org/2022.ltedi-1.45/

[90] N. Buhl, "Training, Validation, Test Split for Machine Learning Datasets," 2024. [Online]. Available: https://encord.com/blog/train-val-test-split/#:~:text=The%20optimal%20split%20ratio%20depends,10%2D20%25%20test%20data.

[91] A. Geron, *Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow: Concepts, Tools, and Techniques to Build Intelligent Systems*, 2nd ed. O'Reilly Media, Inc., 2019.

[92] S. Geman, E. Bienenstock, and R. Doursat, "Neural networks and the bias/variance dilemma," *Neural Computation*, vol. 4, no. 1, pp. 1–58, 01 1992. [Online]. Available: https://doi.org/10.1162/neco.1992.4.1.1

[93] H. He and E. A. Garcia, "Learning from imbalanced data," *IEEE Transactions on Knowledge and Data Engineering*, vol. 21, no. 9, pp. 1263–1284, 2009.

[94] F. Provost and T. Fawcett, "Robust classification for imprecise environments," *Machine learning*, vol. 42, no. 3, pp. 203–231, 2001.

[95] H. Holm, T. Sommestad, J. Almroth, and M. Persson, "A quantitative evaluation of vulnerability scanning," *Inf. Manag. Comput. Security*, vol. 19, 10 2011.

[96] N. Alonso, B. Millidge, J. Krichmar, and E. O. Neftci, "A theoretical framework for inference learning," *Advances in Neural Information Processing Systems*, vol. 35, pp. 37 335–37 348, 2022.

[97] Jeff Geerling, *Ansiblefor DevOps Server and configuration management for humans*, 2nd ed. Leanpub, 2023. [Online]. Available: https://public.tiozaodolinux.com/pdf/ansible-for-devops.pdf