Master's Thesis

# Integrating Domain Knowledge into Transformer-based Approaches to Vulnerability Detection

Department of Mathematics, Informatics and Statistics
Ludwig-Maximilians-Universität München

**Seunghee Jeong**

Munich, November 30th, 2023


Submitted in partial fulfillment of the requirements for the degree of M. Sc.
Supervised by Dr. Clemens-Alexander Brust, Dr. Matthias Aßenmacher, Martin Binder

**Abstract**

The field of vulnerability detection in cybersecurity is critical for ensuring the security and integrity of software systems. Traditional methods like Static Application Security Testing (SAST) and Dynamic Application Security Testing (DAST) have limitations. SAST, while effective in identifying vulnerabilities early in the development cycle, often produces a high rate of false positives and struggles to understand the runtime context. DAST, on the other hand, can detect vulnerabilities in a running application but is limited by its inability to access the source code and its late detection in the software lifecycle. In contrast, the landscape of vulnerability detection has evolved significantly, embracing advanced machine learning models. Initially, the focus was on Recurrent Neural Network (RNN)-based models such as LSTM, BiLSTM, and BiGRU, along with their variants in Convolutional Neural Network (CNN)-based methodologies. However, the field has recently shifted towards transformer-based models, noted for their exceptional performance in natural language processing tasks and their proficiency in interpreting programming languages. This study leverages the strengths of transformer-based models, particularly those tailored for programming languages, to enhance vulnerability detection. By integrating domain knowledge, specifically the Common Weakness Enumeration (CWE) hierarchy, into programming language-specific Transformer-based models. In this study, we investigate the efficacy of transformer-based models through two distinct classification approaches: standard classification and hierarchical classification using a deep classifier. Our primary objective is to assess the impact of integrating domain knowledge, particularly in the context of hierarchical methods, on model performance. This exploration aims to delineate how such integration influences outcomes compared to traditional classification methods, thereby providing insights into the potential advantages of domain-specific enhancements in transformer-based models by adding a novel dimension to the semantic and syntactic analysis of source code. Our hierarchical approach using various loss weights outperformed the standard classification with Focal Loss in multiclass classification. Also, these approaches showed high performances in binary classification even though the models were fine-tuned for multiclass classification task and not for binary classification task. This represents our approaches enable broader learning of semantic and synthetic knowledge in vulnerability detection tasks using transformer-based models and suggests promising direction for future research and application in the field.

# Contents

# 1 Introduction

In an era where digital technologies permeate every aspect of our lives, the security of software systems has become paramount. Cybersecurity, particularly vulnerability detection, is crucial in safeguarding against the increasing sophistication of cyber-attacks. Traditional methods like Static Application Security Testing (SAST) and Dynamic Application Security Testing (DAST) have played pivotal roles but face inherent limitations. SAST, effective in early-stage vulnerability identification, often yields high false positives rates and lacks runtime context understanding. DAST, adept at detecting vulnerabilities in operational applications, is constrained by its inability to access source code and its late intervention in the software lifecycle (Schmitt, 2023).

This backdrop of evolving cyber threats and the inadequacies of conventional methods have catalyzed the shift towards advanced machine learning models in vulnerability detection. The adoption and evolution of various deep learning models have marked the progression of our research approach to vulnerability detection. Early stages focused on models utilizing Convolutional Neural Networks (CNNs) and Long Short-Term Memory (LSTM) networks, as demonstrated in studies like Wu et al. (2017), which employed CNN and LSTM for binary classification in vulnerability detection. Subsequent advancements were made with Bidirectional LSTM (BiLSTM) models, as seen in the works of Li et al. (2018) and Zou et al. (2019). These studies effectively applied BiLSTM-based models for binary and multiclass vulnerability detection, showcasing their versatility. The exploration then extended to Graph Neural Networks (GNNs), with studies such as Zhou et al. (2019) and Fu and Tantithamthavorn (2022) employing GNN-based models like gated graph neural networks, Graph Convolutional Networks (GCN), and Graph Attention Networks (GAT) for binary classification tasks. In more recent developments, the research has incorporated Transformer-based models. Fu et al. (2022) utilized a T5-based model for multi-class classification. In contrast, Fu et al. (2023) combined TextCNN with advanced models like CodeBERT and GraphCodeBERT, demonstrating a novel integration of CNN-based and Transformer-based methodologies in vulnerability detection. As this trajectory shows the field's recently pivoted towards transformer-based models, acclaimed for their exceptional prowess in natural language processing. These models' ability to adeptly interpret programming languages has opened new vistas in vulnerability detection.

Our research stands at the forefront of this evolution, leveraging the strengths of transformer-based models tailored for programming languages. We integrate do-

main knowledge, particularly the Common Weakness Enumeration (CWE) hierarchy, into these models to enhance their analytical depth. Our research employs state-of-the-art transformer-based models to explore the efficacy of incorporating domain knowledge into vulnerability detection. Specifically, we conduct a comparative analysis between hierarchical classification techniques and an alternative class weight-based approach for imbalanced datasets, known as focal loss. This comparison involves standard and hierarchical classification methods with a deep classifier, as detailed in Brust and Denzler (2020). The primary goal of this study is to evaluate the influence of integrating domain-specific knowledge, with a particular focus on hierarchical methodologies, on the performance of these models. This exploration aims to delineate how such integration influences outcomes compared to traditional classification methods, thereby providing insights into the potential advantages of domain-specific enhancements in transformer-based models by adding a novel dimension to the semantic and syntactic analysis of source code. This study addresses crucial research questions such as the feasibility of integrating domain knowledge into transformer-based models, the resultant impact on model performance, and the advanced approach to mitigating extreme class imbalance and efficiently fine-tuning the models for vulnerability detection tasks.

This thesis is structured methodically, beginning with a literature review that provides a comprehensive background on vulnerability detection, existing methodologies, and the role of deep learning models in this domain. Following this, we present our materials and methodology, detailing the data collection, preprocessing techniques, development of transformer-based models, and integrating domain knowledge. Subsequent chapters discuss the results of our experiments, offering a comparative analysis of model performance with and without domain knowledge integration. The discussion chapter delves into the implications of our findings, examining their significance, limitations, and potential avenues for future research. The thesis culminates with a conclusion that synthesizes our research findings and outlines directions for further exploration in this evolving field of cybersecurity.

# 2 Vulnerability Detection in Language Models

This chapter delves into the foundational understanding of vulnerability detection, examining its definition, existing methodologies, and the emergence of deep learning as a necessary approach in light of the limitations inherent in current practices. It further investigates the essential aspects of language model applications, ranging from fundamental principles to intricate transformer-based architectures such as BERT (Devlin et al., 2019) and its adaptations for dual-modality in both Natural and Programming Languages. The discussion progresses to trace the evolution from conventional models like Recurrent Neural Networks (RNNs) and Long Short-Term Memory (LSTMs) (Hochreiter and Schmidhuber, 1997) to the advanced transformer-based architectures, which are employed for complex adaptations in software vulnerability detection. This is done while referencing pivotal studies in the field.

## 2.1 Definition of Weakness and Vulnerability

To navigate the complexities of software security, it is essential to understand the specific lexicon that defines its threats. The terms *vulnerability* and *weakness* are central to this discourse, each signifying distinct but interrelated concepts. A *vulnerability* denotes an explicit flaw within a software system—be it in the code, design, or implementation—that can be exploited by threat actors to compromise system security or integrity. Such vulnerabilities vary in origin, ranging from simple coding mistakes to complex configuration errors, and are each identified by a unique *Common Vulnerabilities and Exposures (CVE)* identifier for global reference and analysis. CVE (Common Vulnerabilities and Exposures) is a system for naming and listing cybersecurity vulnerabilities and exposures. It was created in 1999 to solve the problem of different security tools having their own systems for naming vulnerabilities, which made it hard to coordinate and protect against security threats effectively. CVE provides a standardized way to identify vulnerabilities, making it easier for different cybersecurity tools and services to communicate with each other, share information, and improve security coverage. How CVE Works The process of creating a CVE Identifier begins with discovering and reporting a potential security vulnerability. Managed by The MITRE Corporation, CVE is widely used globally in various cybersecurity products and services (*Introduction to CVE*, n.d.).

While *vulnerability* refers to the immediately exploitable defects, a *weakness* indicates a more generalized condition within the software that may give rise to one or more vulnerabilities. Weaknesses might originate from inadequate coding practices, architectural flaws, or failure to adhere to security protocols and are categorized

within the *Common Weakness Enumeration (CWE)* system. CWE (Common Weakness Enumeration) is a community-developed list of common software and hardware weakness types that have implications for security. Established to provide a standardized language for identifying and describing these weaknesses, CWE aims to educate software and hardware developers, architects, designers, and acquirers on how to eliminate common mistakes before product delivery, thereby preventing vulnerabilities at their source. The CWE List, first released in 2006 and expanded over time to include hardware weaknesses, serves as a foundation for understanding and addressing security flaws in both software and hardware domains. It supports developers and security practitioners in describing weaknesses in a shared language, checking for weaknesses in products, evaluating tool coverage, and leveraging a baseline for weakness identification, mitigation, and prevention. CWE, managed by The MITRE Corporation and endorsed by the international CWE community, is instrumental in shaping security standards and practices across the industry (MITRE, 2023).

The nuanced distinction between a CVE and a CWE is foundational for professionals in the field. A CVE ID is allocated to a specific vulnerability instance, a weakness that has materialized and is susceptible to exploitation. In contrast, a CWE ID classifies the broader type of weakness, potentially leading to such vulnerabilities. This systematic classification not only aids in the precise identification and remediation of software vulnerabilities but also in the proactive strengthening of software against potential security threats.

## 2.2 Vulnerability detection

Software and system vulnerabilities are found, examined, and mitigated as part of vulnerability detection, an essential component of cybersecurity. Asset identification, vulnerability scanning, weakness detection, risk assessment, patch management, and continuous monitoring are all included in this multi-stage procedure. The CVE database records thousands of new vulnerabilities annually, highlighting the volume and increasing complexity of cyber threats. These numbers show the shortcomings of conventional approaches and the need for more advanced solutions. As Comparitech (2023) represents, recent data indicates that there were over 8,000 vulnerabilities identified in just the first quarter of 2022, which is a 25% increase from the same period the previous year. This pattern emphasizes how difficult cybersecurity is becoming and how important proactive detection techniques are.
One key aspect of vulnerability detection is the distinction between detecting *Com-*

*mon Vulnerabilities and Exposures (CVE)* identifiers and *Common Weakness Enumeration (CWE)* identifiers. While a CVE ID represents a cataloged vulnerability known to be exploitable, a CWE ID signifies a broader category of potential weaknesses in software design or implementation that could lead to vulnerabilities. Detecting weakness/vulnerability within programming code is particularly vital, as it identifies systemic issues that could manifest as multiple, possibly undiscovered, vulnerabilities (CVEs). This early detection of weaknesses enables developers and security professionals to address fundamental flaws before they can be exploited.
Detecting CWE identifiers can be especially beneficial when integrated into automated vulnerability detection systems. Integrating such knowledge allows these models to detect known vulnerabilities and predict and locate potential weaknesses that have not yet been exploited or documented as CVEs.
Traditional approaches to vulnerability detection rely on manual analysis and expert knowledge. However, this approach is time-consuming, error-prone, and not scalable, particularly with the increasing complexity of software and the variety of security threats. Advanced deep learning methods, particularly transformer-based models, have emerged in response to these challenges. They have shown promising results in automating the process of vulnerability detection. These models excel in interpreting complex programming languages and integrating domain knowledge like the CWE hierarchy, enhancing vulnerability detection by predicting and identifying potential weaknesses. This represents a significant advancement in cybersecurity, moving from a reactive to a more proactive, efficient, and tailored approach to vulnerability detection.

## 2.3 Existing Approaches to Vulnerability Detection

The following seven typical techniques for detecting vulnerabilities, described in Aptori (2023), exist at present: Overall, each approach has its strengths and weaknesses, and a combination of approaches is often used to provide comprehensive coverage of the software system under test.

1. **Static Application Security Testing (SAST)**: SAST, also called white-box testing, inspects the application's source code for potential vulnerabilities early in development. Tools like SonarQube are used for SAST. Pros include early detection of issues and detailed insights into the codebase, while cons involve potential high false positives and negatives and difficulty in identifying runtime-specific vulnerabilities.

2. **Dynamic Application Security Testing (DAST)**: DAST, also called black-box testing, tests applications in their operational state, identifying vulnerabilities exploitable by attackers, such as cross-site scripting and SQL injection. Tools like OWASP ZAP are used for DAST. Its advantages include finding runtime-specific vulnerabilities potentially ignored by SAST and not requiring source code access. However, it can be slow and provides less detailed information about vulnerabilities than SAST.

3. **Interactive Application Security Testing (IAST)**: IAST integrates elements of both SAST and DAST, employing agents within the application to monitor data flow and accurately identify vulnerabilities during runtime. Tools like Veracode are utilized for IAST. It offers precise runtime information and works well with custom code, but some infeasible application instrumentation might impact application performance.

4. **Software Composition Analysis (SCA)**: The main objective of SCA is to find security holes in third-party libraries and open-source components used in an application. These elements are frequently used extensively in modern software development, and each can potentially generate vulnerabilities. Tools like OWASP Dependency Check are used for SCA. It's effective in identifying vulnerabilities in external components and managing risks but might not cover all components and relies on databases that may not be fully updated.

5. **Penetration Testing (Pen Testing)**: Penetration testing (in short, Pen testing) involves simulated cyberattacks to find potential exploits in systems, using tools like Metasploit. It reveals real-world vulnerabilities and complex exploits but can be expensive and time-consuming, requiring skilled testers.

6. **Fuzz Testing (Fuzzing)**: Fuzzing is an automated method that inputs random data into software to find vulnerabilities, using tools like Aptori. It's effective in discovering edge case vulnerabilities and big-size codes but can produce a high volume of non-exploitable results and requires careful management.

7. **Runtime Application Self-Protection (RASP)**: RASP integrates into an application to protect against real-time attacks by analyzing behavior and context during operation. It offers real-time protection and detailed attack data but can impact performance and might not suit all applications.

## 2.4 Deep-learning based Vulnerability Detection

This cutting-edge method uses deep learning algorithms to find vulnerabilities in software automatically. By examining enormous code datasets, these algorithms can learn to identify intricate patterns and anomalies that might point to security vulnerabilities. This technique can eventually adjust to new threats and is especially good at identifying vulnerabilities that more conventional approaches might overlook. However, training requires a lot of computer power and huge, labeled datasets. The results can also be difficult to interpret, requiring professional analysis to comprehend and apply the conclusions.

1. **Recurrent Neural Network (RNN)-based:** An example of an RNN-based approach for vulnerability detection is "VulDeePecker" (Li et al., 2018), a deep learning-based system that uses a BiLSTM (Bidirectional Long Short-Term Memory) model to detect vulnerabilities from source code.

2. **Convolutional Neural Network (CNN)-based:** An example of a CNN-based vulnerability detection system is the work done by Zhou et al. (2019), using 1D-convolution CNNs and a dense layer on a string of tokens to detect vulnerabilities.

3. **Graph Neural Network (GNN)-based:** An example of a GNN-based method is a method by Russell et al. (2018), where they use Graph Neural Networks to learn from the code's abstract syntax tree (AST) to detect vulnerabilities and use Random Forest as a classifier.

4. **Transformer-based:** Hanif and Maffeis (2022) uses a transformer-based model called RoBERTa, introduced in Liu et al. (2019), to create program embeddings that can effectively identify vulnerabilities in software, demonstrating the adaptability and effectiveness of transformer architectures in understanding complex software structures for security purposes. Further, they connect it to multi-layer perception (MLP) and convolutional neural networks (CNN) to fine-tune vulnerability detection models.

Overall, deep learning-based approaches to vulnerability detection have shown promising results and have the potential to improve the accuracy and efficiency of vulnerability detection compared to traditional approaches. However, there are still challenges to overcome, such as the limited availability of labeled data and the difficulty of interpreting and explaining the decisions made by deep learning models.

## 2.5 Pre-trained Models for Programming and Natural Languages

This section explores the significant advances in language model development, focusing on their application in programming contexts. Starting with an overview of the evolution of language models, we trace their progression from early statistical models to the sophisticated Transformer architecture. We then delve into BERT, a groundbreaking model in natural language understanding, and extend our focus to CodeBERT and GraphCodeBERT, which represent specialized adaptations of the Transformer model for the realm of programming languages.

### 2.5.1 Understanding Pretraining in Machine Learning

Pretraining is a cornerstone concept in modern machine learning, especially within the realm of deep learning. Essentially, it refers to the process of training a neural network model on a vast dataset prior to its application on specific tasks. This initial phase typically involves self-supervised learning, where the model learns to predict or reconstruct parts of the input data, thereby gaining a broad and fundamental understanding of the data patterns. For example, in language models, pretraining might include tasks like predicting the next word in a sentence. This approach is particularly powerful as it allows the model to learn a rich data representation, making it more versatile and effective when later fine-tuned for specific tasks. Pretrained models, particularly in later iterations, have demonstrated remarkable effectiveness across various domains, underlining the significance of this training methodology.

### 2.5.2 The Evolutionary Development of Language Models

The journey of language models began in the 1950s, marking the dawn of using computers for natural language processing (Turing, 1950). The 1980s saw the rise of statistical language models, revolutionizing the field.
One of the earliest notable statistical language models (SLMs) was the n-gram model, described in Jurafsky and Martin (2023), which was developed in the 1960s and uses a probabilistic approach to predict the likelihood of a word based on the previous n-1 words in a sequence. The limit of n-gram models is that as the value of n increases, the size of the n-gram model also increases exponentially. This leads to the "curse of dimensionality", where the model becomes increasingly sparse, making it difficult to estimate probabilities accurately. Additionally, n-gram models cannot capture long-term dependencies and context that may be necessary for some natural language processing tasks (Weizenbaum, 1966).

The 1990s introduced neural language models, starting with feedforward neural network language models and progressing to more advanced architectures like recurrent neural networks (RNNs) and long short-term memory (LSTM) networks. These models, capable of processing sequential information and retaining context over longer sequences, offered significant improvements over traditional n-gram models. However, they faced challenges like computational intensity, parallelization difficulty, and issues like the vanishing gradient problem (Jelinek, 1992).
Recurrent Neural Networks (RNNs) process input sequences one element at a time while maintaining an internal state that encodes information from previous elements. The formula of Recurrent Neural Network (RNN) is defined as,

$$h_t = \sigma(W_{hh}h_{t-1} + W_{xh}x_t + b_h)$$

$$y_t = \text{softmax}(W_{hy}h_t + b_y)$$

Where $h_t$ is the hidden state at time step $t$, $\sigma$ is the activation function (usually the hyperbolic tangent or the sigmoid), $W_{hh}$ is the weight matrix for the hidden state, $W_{xh}$ is the weight matrix for the input $x_t$, $b_h$ is the bias vector for the hidden state, $y_t$ is the output at time step $t$, $W_{hy}$ is the weight matrix for the output, and $b_y$ is the bias vector for the output.

### 2.5.3 Transformers

Introduced in Vaswani et al. (2023), the transformer architecture marked another significant NLP development. Among the drawbacks of RNN and LSTM models were bottleneck problems brought on by sequential processing, which was costly and time-consuming computationally, and the problem of vanishing gradients, making capturing long-term dependencies challenging. However, the transformer may use attention to tackle these RNN and LSTM problems. When a transformer generates an output sequence, attention serves as a method to let the model preferentially focus on the most relevant parts of the input sequence. The attention mechanism works by computing a weighted sum of the encoder outputs, where the weights are learned based on the similarity between the decoder's hidden state and each encoder output.
The attention mechanism in transformers can be mathematically represented as follows:
Given a sequence of input vectors $x_1, x_2, ..., x_n$ and a query vector $q$, the attention

mechanism computes a weighted sum of the input vectors as:

$$\text{Attention}(q, \{x_i\}_{i=1}^{n}) = \sum_{i=1}^{n} \alpha_i x_i,$$

where the weights $\alpha_i$ are computed as:

$$\alpha_i = \frac{\exp(e_i)}{\sum_{j=1}^{n} \exp(e_j)},$$

And $e_i$ is a score assigned to each input vector based on its similarity to the query vector:

$$e_i = \text{score}(q, x_i).$$

The score function can take different forms, such as dot product, scaled dot product, or multi-layer perceptron (MLP). The choice of score function depends on the specific application and the desired properties of the attention mechanism.
Various attention mechanisms have been developed based on the original attention mechanism, including self-attention, cross-attention, and multi-head attention. Self-attention and cross-attention are distinguished based on whether they define the key and query from the same input sequence or not. Multi-head attention operates in parallel multiple times. As a result, each type has a distinct formula.

1. **Self-attention:**

   Self-attention is a type of attention where the input sequence is mapped to queries, keys, and values, which are then used to compute a weighted sum of the values. The weights are determined by the similarity between the queries and keys. Self-attention is used in the encoder layers of the transformer model to capture dependencies within the input sequence.

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

   Where $Q$, $K$, and $V$ are the query, key, and value matrices. $d_k$ is the dimensionality of the key vectors.

2. **Cross-attention:**

   Cross-attention is a type of attention used in the decoder layers of the transformer model. It computes a weighted sum of the values based on the similarity between the queries and the keys obtained from a different input sequence. For

example, the queries are the decoder's hidden states in machine translation, and the keys and values are the encoder outputs.

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

Where $Q$ is the query matrix, $K$ and $V$ are the key and value matrices obtained from a different input sequence, and $d_k$ is the dimensionality of the key vectors.

3. **Multi-head attention:**

   Multi-head attention is a variation of self-attention that allows the model to jointly attend to information from different representation subspaces. The query, key, and value matrices are projected into $h$ subspaces using learnable linear transformations in multi-head attention. The self-attention operation is then performed on each of these projected subspaces in parallel, and the results are concatenated and projected again to produce the final output.

$$\text{Multi-Head}(Q, K, V) = \text{Concat}(head_1, head_2, ..., head_h)W^O$$

Where $head_i = \text{Attention}(QW_i^Q, KW_i^K, VW_i^V)$ is the output of the $i$-th attention head, $W_i^Q$, $W_i^K$, and $W_i^V$ are learnable linear transformations, and $W^O$ is a learnable linear transformation applied to the concatenated outputs of all the attention heads.

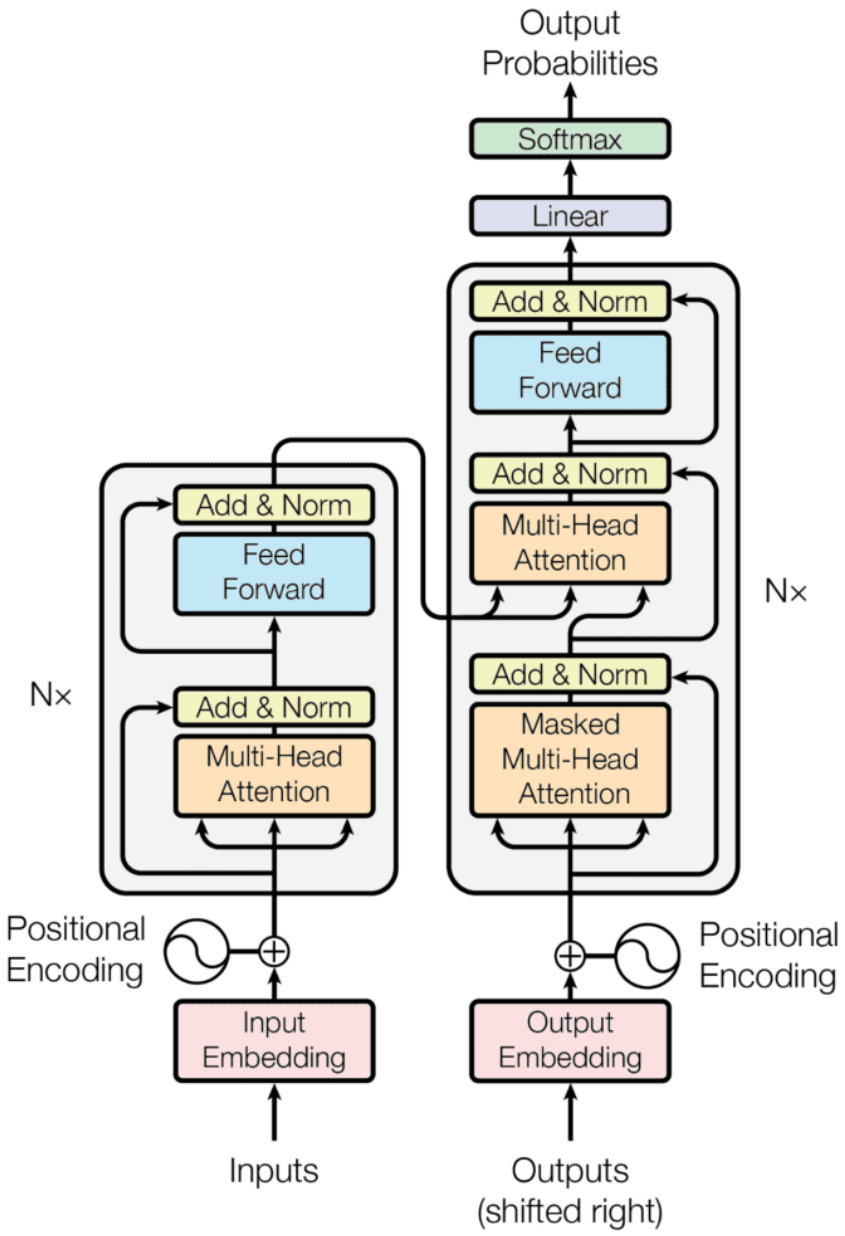



Figure 1: Architecture of Transformer (Vaswani et al., 2023)

The Transformer model is a complex neural network for processing sequences, with an encoder-decoder structure enabling parallelization and capturing long-range de-

pendencies. Figure 1 shows the basic architecture of the Transformer model, with the encoder on the left and the decoder on the right. The encoder maps input sequences (represented as a matrix of token embeddings) to a sequence of encoded representations via self-attention, adding positional encodings to maintain sequence order. Each encoder layer consists of multi-head attention and feed-forward networks with residual connections for gradient flow. Residual connections allow the gradients to flow more easily through the network during training. The decoder, mirroring the encoder's structure, also integrates masked multi-head attention for sequential prediction, using output from the encoder for context. The decoder takes a target sequence (also represented as a matrix of embeddings) and produces a sequence of output tokens. Finally, a softmax layer outputs probabilities for the next token in the sequence, with the entire model being trained end-to-end for tasks like machine translation.

To elaborate further and offer a more comprehensive explanation, the input to the transformer model is a sequence of tokens, which are typically represented as one-hot vectors or embeddings. Depending on the specific task and dataset, these tokens can represent words, subwords, or characters.

Before feeding the input sequence into the transformer encoder, positional encodings are added to each token embedding to provide information about the relative position of each token in the sequence. This is because the transformer model does not have a built-in notion of order or position and thus requires explicit positional information to be added. The positional encoding is a fixed-length vector added to the input embedding of each token, representing the token's position in the sequence.

The transformer encoder takes in the input sequence with positional embeddings and processes it through a series of self-attention and feedforward layers. In the self-attention layer, the encoder computes a weighted sum of the input sequence, with the weights for each token computed based on their similarity to all other tokens in the sequence. The output of the self-attention layer is then passed through a feedforward neural network to generate the final encoded representation of the input sequence.

The encoder output is then fed into the decoder, along with a target sequence for the task being performed. Like the encoder, the decoder has a series of self-attention and feedforward layers. In addition, the decoder has an additional cross-attention layer that attends to the output of the encoder to help generate the final output sequence.

During decoding, the model generates the output sequence one token at a time, with each token generated based on the previous tokens in the output sequence, the encoder output, and the previously generated decoder output. At each decoding

step, the model applies self-attention and cross-attention mechanisms to compute attention weights, which are then used to weight the input and encoder output representations. The resulting weighted sum is passed through a feedforward neural network to generate the next output token. This process is repeated until the entire output sequence is generated.
The transformer model is a powerful sequence-to-sequence model capable of generating high-quality outputs for various natural language processing tasks. Therefore, the latest powerful pre-trained language models are all built on the Transformer architecture and have evolved into different variants.

### 2.5.4 BERT

BERT (Bidirectional Encoder Representations from Transformers) is a transformative model in natural language processing introduced by Devlin et al. (2019). It represents a significant shift in how language models understand context, employing a deep learning technique based on the Transformer architecture. Unlike previous models that processed text in one direction (either left-to-right or right-to-left), BERT reads text bi-directionally, enabling a more comprehensive understanding of context. This bidirectional nature allows BERT to perform state-of-the-art language tasks, including question-answering, sentiment analysis, and language inference, making it a foundational model in modern NLP applications.

Regarding the dataset used for the pre-training BERT model, it was conducted on a large corpus of unlabeled text (like Wikipedia and BookCorpus). The Wikipedia dataset contains the entire English Wikipedia, which is a comprehensive and diverse source of general knowledge and provides a vast range of vocabulary and topics. BookCorpus is a collection of books. BookCorpus offers a rich source of narrative text that differs in style and structure from Wikipedia. These datasets, encompassing a wide array of topics and writing styles, are crucial for BERT's ability to understand and process complex language structures and contexts. This extensive pre-training enables BERT to develop a deep and nuanced understanding of language, contributing significantly to its performance on various downstream tasks.

The BERT input representation combines three types of embeddings:

1. **Token Embeddings**: These are the vector representations of the individual tokens (words or subwords) obtained from the input text. Each word is transformed into a vector that encapsulates its semantic meaning in a high-dimensional space.

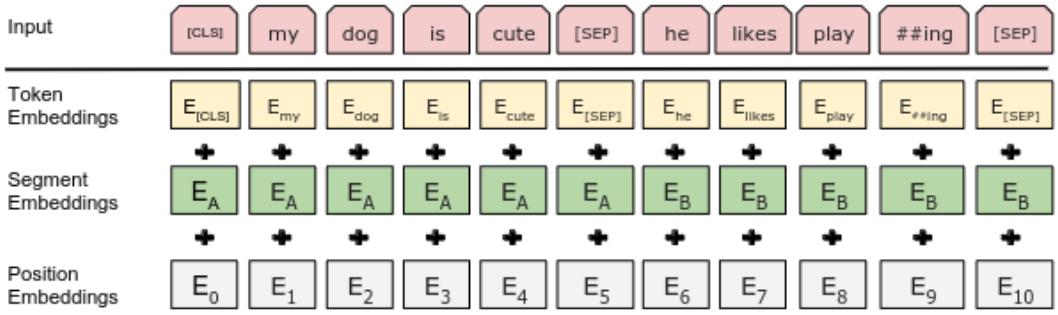



Figure 2: BERT Input Representation (Devlin et al., 2019)

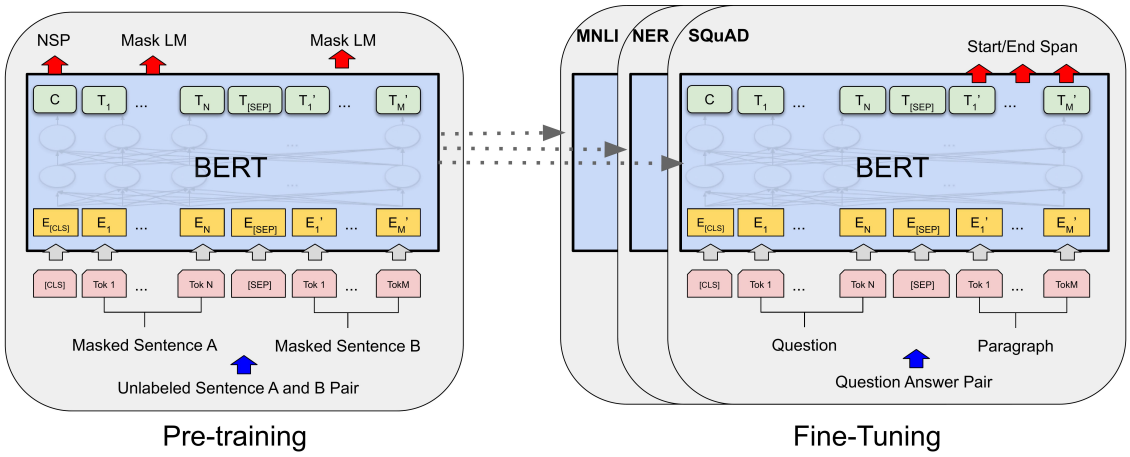



Figure 3: BERT Overview (Devlin et al., 2019)

2. **Segment Embeddings**: BERT can handle pairs of sequences for certain tasks, like question answering. Segment embeddings are used to differentiate between these two sequences. Each token of the first sequence receives a segment embedding $E_A$ and the second sequence $E_B$.

3. **Position Embeddings**: BERT employs position embeddings to encode the order of tokens in the sequence. Each token position is associated with a unique position embedding, indicating its sequential position. They are learned vectors that capture the position of tokens within the input sequence. These embeddings are crucial as they allow BERT, which processes tokens in parallel, to maintain word order information.

All three embedding types are combined element-wise to create the final input representation for each token, providing the model with comprehensive contextual information. The values of these embeddings are learned during pre-training.

Pre-training was conducted using two tasks: **Masked Language Modeling (MLM)** and **Next Sentence Prediction (NSP)**.
**Masked Language Modeling (MLM)** is a self-supervised modeling objective introduced to couple self-attention and deep bidirectionality without violating causality. In MLM, it generates the samples by taking among 15% of token samples, and it replaces 80% of them by `[MASK]`, 10% by random token, and leaves 10% unchanged. It predicts randomly masked tokens in a sentence. `[MASK]` token is only used in pre-training, not fine-tuning.
In **Next Sentence Prediction (NSP)** predicts if one sentence logically follows another. It generates samples by randomly sample * negative examples (cf. word2vec (Mikolov et al., 2013)). Half of the second sentence is the next sentence, and the rest is a randomly sampled sentence. It can be seen as self-supervised learning.
In Figure 3, BERT takes a sequence of tokens (words or subwords) as input, adding special tokens like `[CLS]` for classification tasks and `[SEP]` to separate segments. The `[CLS]` token is sequence representation for classification, and the `[SEP]` token is for separation of the two input sequences since the BERT model can take two sequences as input. Its output depends on the task, ranging from token-level predictions for tasks like named entity recognition to sequence-level outputs for classification. Depending on the downstream task, for a classification task, simply the `[CLS]` token can be used since it contains the representation of token embeddings by each class. Otherwise, the other tokens can be utilized.

### 2.5.5 CodeBERT

CodeBERT, introduced by Feng et al. (2020), is an extension of the transformer-based neural architecture used in BERT. It introduces a pioneering approach as the first large-scale pre-trained model for multiple programming languages intertwined with natural language. Its fundamental objective is to capture the intricate semantic connections existing between natural language and programming language. By doing so, CodeBERT acquires general-purpose representations that support various downstream NL-PL applications, including natural-language code search and code documentation generation.

CodeBERT demonstrates state-of-the-art performance in the aforementioned downstream tasks through fine-tuning, consistently surpassing the RoBERTa model. The

model leverages a multi-modal pre-training approach, incorporating both bimodal NL-PL data and unimodal PL/NL data. The dataset encompasses six programming languages: Python, Java, JavaScript, PHP, Ruby, and Go. The unimodal data is sourced from GitHub repositories, ensuring uniform training across the different languages. Bimodal data points consist of pairs comprising code snippets and function-level natural language documentation. The training process aligns with the approach employed in multilingual BERT.

CodeBERT adopts a hybrid-objective function for training, encompassing two distinct pre-training tasks. The first task resembles **Masked Language Modeling (MLM)** employed in BERT, where masked words are predicted based on contextual information. The second task, **Replaced Token Detection (RTD)**, introduces a novel learning objective that distinguishes CodeBERT from BERT. RTD involves detecting plausible alternatives sampled from generators. The training commences with MLM on natural and programming language data, followed by the RTD task. The connection between these pre-training tasks lies in the role of MLM in improving the generator by utilizing its output, which serves as the training result from the unimodal ML and PL data, as input for the RTD task that utilizes bimodal NL-PL pairs.

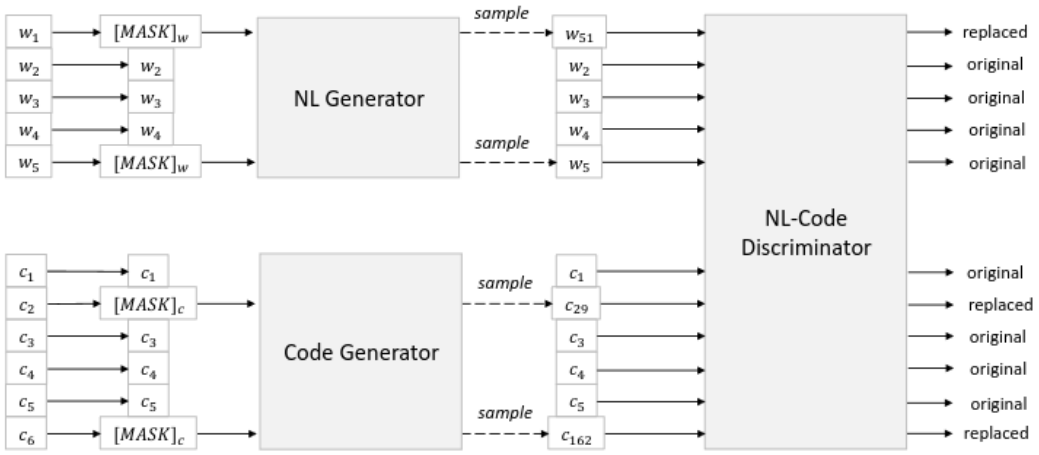



Figure 4: CodeBERT Pre-training (Feng et al., 2020)

The architecture of CodeBERT aligns with the design principles of BERT and RoBERTa (Liu et al., 2019), employing a multi-layer bidirectional Transformer. With a total parameter count of 125 million, CodeBERT shares the same archi-

tectural foundation as RoBERTa. The input format of the model consists of tokens structured as `[CLS]` w1, w2, `[SEP]` c1, c2, `[EOS]`. The `[SEP]` token acts as a special separator, concatenating the natural language and programming language text segments. The `[CLS]` token, positioned at the front, is another special token commonly used for classification or ranking tasks. Its final hidden representation constitutes an aggregated sequence representation, which is one of the model's outputs. The `[EOS]` token signifies the end of the segments. The model's output encompasses representations of the two segments along with the `[CLS]` token. The first represents the contextual vector representation of the natural language and programming language (code) tokens, while the second represents the aggregated sequence representation of the `[CLS]` token.

CodeBERT has demonstrated state-of-the-art performance in various downstream NL-PL tasks and has made significant contributions to the field of code understanding, enabling more advanced code analysis, search, and generation capabilities.

### 2.5.6 GraphCodeBERT

GraphCodeBERT is an advanced pre-trained model specifically designed for programming languages, taking into consideration the intrinsic structure of code. Unlike traditional approaches that rely on syntactic-level structures such as abstract syntax trees (ASTs), GraphCodeBERT exploits the semantic-level information known as data flow during its pre-training phase.

The data flow representation captures the relationships between variables, where nodes represent variables and edges indicate the connections or "origins" of values between variables. Compared to ASTs, data flow graphs are less complex and avoid unnecessary deep hierarchies, enhancing efficiency. Since the data flow remains consistent regardless of the abstract grammar employed in different programming languages when considering the same source code, this code structure holds vital code semantic information that is indispensable for comprehending and interpreting the code accurately.

Regarding the model architecture, the model backbone is based on BERT, utilizing the multi-layer bidirectional Transformer Devlin et al. (2019). In contrast to solely employing source code, it incorporates paired comments in the pre-training process to enhance the model's ability to handle code-related tasks involving natural languages, such as natural language code search. Additionally, the included data flow, represented as a graph, is part of the input to the model.

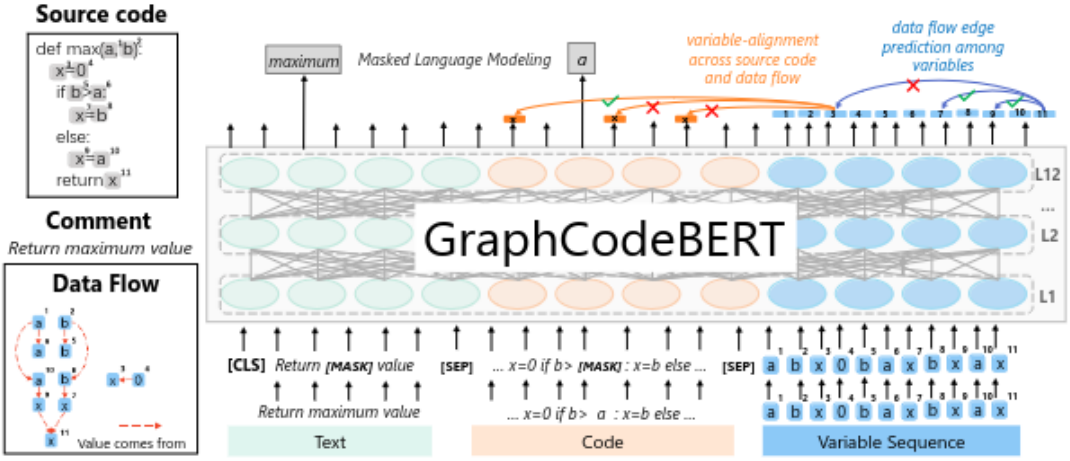



Figure 5: GraphCodeBERT Pre-training (Guo et al., 2021)

Regarding the input of the model, it is given a source code $C = \{c_1, c_2, ..., c_n\}$ and its corresponding comment $W = \{w_1, w_2, ..., w_m\}$, the corresponding data flow $G(C) = (V, E)$. Here, $V = \{v_1, v_2, ..., v_k\}$ represents the set of variables, and $E = \{\varepsilon_1, \varepsilon_2, ..., \varepsilon_l\}$ represents the set of directed edges that indicate the source of each variable's value. The sequence input $X = \{[\texttt{CLS}], W, [\texttt{SEP}], C, [\texttt{SEP}], V\}$ is formed by concatenating the comment, source code, and variables, where [CLS] is a special token denoting the beginning, and [SEP] is a special symbol used to differentiate different data types.

GraphCodeBERT takes this sequence input $X$ and converts it into input vectors $H_0$. Each token's input vector is constructed by adding the respective token and position embeddings. A specific position embedding is assigned to all variables to indicate that they represent nodes in the data flow. The model applies $N$ transformer layers over the input vectors to generate contextual representations $H_n = \text{transformer}_n(H_{n-1})$, where $n \in [1, N]$. Each transformer layer consists of an identical transformer architecture that performs a multi-headed self-attention operation (Vaswani et al., 2017) followed by a feedforward layer applied to the input $H_{n-1}$ in the $n$-th layer.

The output $\hat{G}_n$ of the multi-headed self-attention operation for the $n$-th transformer layer is calculated as follows:

$$Q_i = H_{n-1}W_{Q_i}, \quad K_i = H_{n-1}W_{K_i}, \quad V_i = H_{n-1}W_{V_i} \quad ,$$

$$\text{head}_i = \text{softmax}\left(\frac{Q_i K_i^T}{\sqrt{d_k}} + M\right) V_i \quad ,$$

$$\hat{G}_n = [\text{head}_1; ...; \text{head}_u] W_{O_n} \quad ,$$

where $H_{n-1} \in \mathbb{R}^{|X| \times d_{\text{hid}}}$ represents the linear projection of the previous layer's output to a set of queries, keys, and values using model parameters $W_{Q_i}$, $W_{K_i}$, and $W_{V_i}$, respectively.

A graph-guided masked attention mechanism integrates the graph structure into the Transformer model, enabling selective attention to relevant information. This attention mechanism prevents a query $q_j$ from attending to a key $k_i$ by assigning an infinitely negative value to the attention score $q_j^T k_i$, effectively eliminating the attention weight through a subsequent softmax operation.

To capture the dependency relationships between variables, it allows a node query $q_{v_i}$ to attend to a node key $k_{v_j}$ only if there exists a direct edge from node $v_j$ to node $v_i$ (i.e., $v_j, v_i \in E$) or if they refer to the same node (i.e., $i = j$). Otherwise, the attention is masked by assigning an infinitely negative value to the attention score.

To represent the relationship between source code tokens and nodes in the data flow, a set $E$ is defined, where $v_i, c_j/c_j, v_i \in E$ indicates that variable $v_i$ is identified from the source code token $c_j$. Based on this definition, it allows the node query $q_{v_i}$ and code key $k_{c_j}$ to attend to each other only if $v_i, c_j/c_j, v_i \in E$.

Formally, a graph-guided masked attention matrix as the mask matrix $M$ in Equation (4), where $M_{ij} = 0$ if either $q_i$ is one of `[CLS]` or `[SEP]`, or if $q_i, k_j$ belong to sets $W \cup C$ or $E \cup E$. Otherwise, the value is set to $-\infty$ to mask the attention.

The model employs two key components: *edge prediction*, which learns representations from the code structure, and variable alignment, which aligns representations between the source code and data flow. A guided mask attention function is also utilized to consolidate the code structure.

In summary, GraphCodeBERT's pre-training encompasses three principal tasks to augment its code understanding capabilities:

1. **Masked Language Modeling (MLM) task:** Adapted from BERT, leverages context from source code and comments to predict masked code tokens, promoting an integrated understanding of natural and programming languages.

2. **Edge Prediction task:** Designed to comprehend data flow within code by predicting data flow graph edges, which enriches the model's code representation learning.

3. **Node Alignment task:** Aims to align the representation between source code tokens and data flow, enabling the model to accurately identify where variables in the data flow are defined or used in the source code.

These tasks synergistically enhance GraphCodeBERT's grasp of code's structural and logical intricacies.

To train GraphCodeBERT, the CodeSearchNet dataset, comprising 2.3 million code snippets from six programming languages paired with natural language documents, is employed. The model's performance is evaluated on four downstream tasks: natural language code search, duplicate detection, code translation, and code improvement. Experimental results demonstrate that GraphCodeBERT achieves state-of-the-art performance across all four tasks. Furthermore, its integration of code structure and the introduced pre-training tasks contribute to its consistent prioritization of attending to data flows.

### 2.5.7 Model Comparisons

In this study, two models were utilized, both of which are based on BERT: CodeBERT and GraphCodeBERT. Given that the vulnerability detection task in this research is framed as a multiclass classification problem, we opted for encoder-based models within the realm of transformer-based models. BERT-based models, CodeBERT and GraphCodeBERT were chosen for their proven performance in many tasks, achieving state-of-the-art (SOTA) results and demonstrating high performance in various downstream tasks, particularly those based on classification.
Additionally, considering that the data in this study is text-based on programming languages (PL), CodeBERT, trained in an NL-PL multi-modality manner, was deemed more appropriate. This is due to its Natural Language and Programming Language training, offering potentially better alignment with the study's data characteristics.
Lastly, programming code is a type of textual data characterized by sequences and patterns. Unlike linear natural language text, programming code often exhibits intricate relationships between variables and tokens, including those spanning across different code segments. This prompted the inclusion of GraphCodeBERT, a model trained on CodeBERT base with additional graph information, to evaluate the impact of such graph relationships on learning and performance. This comparison

aimed to discern how graph information influences learning outcomes and performance metrics. Thus, these BERT-based models were selected to suit the specific requirements of this study's dataset and the classification task, ensuring a more appropriate and fair comparison.

Table 1: Comparison of NLP Models for Code Understanding

| **Aspect** | **BERT (General)** | **CodeBERT** | **GraphCodeBERT** |
|---|---|---|---|
| Data Size | 13GB | 180 GB | 180GB |
| Data Source | BookCorpus, Eng. Wikipedia | BookCorpus, Eng. Wikipedia, CC-News, OpenWebText, Stories, CodeSearchNet | BookCorpus, Eng. Wikipedia, CC-News, OpenWebText, Stories, CodeSearchNet |
| Base Model | BERT | RoBERTa | RoBERTa |
| Pre-training Objectives | Lask Language Modeling (MLM), Next Sentence Prediction (NSP) | MLM, Replaced Token Detection (RTD) | MLM, Edge Prediction, Node Alignment |
| Tokenization | WordPiece | Byte-Pair Encoding (BPE) | Byte-Pair Encoding (BPE) |
| Parameters | 110M | 125M | 125M |
| Attention Heads | 12 | 12 | 12 |
| Layers | 12 | 12 | 12 |
| Hidden Dimension | 768 | 768 | 768 |
| Vocab Size | 30522 | 50265 | 50265 |

## 2.6 Related Work

The domain of Software Vulnerability Classification (SVC) has evolved, with deep learning techniques increasingly being applied to predict and classify vulnerabilities more accurately. Recent advancements have seen a pivot from recurrent and convolutional neural networks to Transformer-based models, which capitalize on the hierarchical nature of vulnerabilities and incorporate domain-specific knowledge for enhanced detection.

### Multiclass Classification for Vulnerability Detection

Recent studies have broadened the scope from binary to multiclass classification to provide a more nuanced categorization of software vulnerabilities. This evolution reflects the need to classify beyond a simple vulnerable/non-vulnerable dichotomy, addressing the complex spectrum of vulnerabilities identified by CWE IDs. In the work by Contreras et al. (2023), a multiclass classification system was developed, integrating with the VSCode IDE to enhance developer experience. This system extends upon the binary classification framework introduced by Fu and Tantithamthavorn (2022), which distinguishes between vulnerable and non-vulnerable code segments. Further developments in this area include the ALBugHunter program (Contreras et al., 2023), which not only classifies code as non-vulnerable but also assigns CWE IDs and CWE types and calculates the Common Vulnerability Scoring System (CVSS) scores for a more detailed analysis. Concurrently, the VulRepair tool, as cited in Fu et al. (2022), takes non-vulnerable code inputs and suggests potential repair paths, illustrating the growing trend towards automated remediation in software security. A key resource for these studies has been the 'Big-vul' dataset, consisting of a significant corpus of C/C++ language vulnerabilities sourced from GitHub repositories. This dataset has facilitated the exploration of various machine learning models, including CodeBERT, which utilizes Byte-Pair Encoding (BPE) for effective tokenization and classification of CWE IDs and types. ALBugHunter focuses on a Multi-Objective Optimization (MOO) approach, which encompasses a combined Cross Entropy Loss Function and reflects the evolving complexity of vulnerability classification tasks. In another work Ziems and Wu (2021), a noteworthy study utilized the Software Assurance Reference Dataset (SARD), equally splited between vulnerable and non-vulnerable data. In a departure from previous research, this approach included non-vulnerable data as a distinct class, resulting in a classification scheme encompassing 124 unique labels, including 123 different CWE IDs and one additional class for non-vulnerable data. A distinct feature of the preprocessing stage in this study was the simplification of function names within the dataset, trans-

forming labels such as *cve vulnerability function_"good"/"bad"_variables* into a unified identifier like *func1*. This research did not employ pre-trained models but instead investigated five different model configurations: LSTM, BERT, and combinations thereof, including LSTM, BiLSTM, BERT, BERT+LSTM, and BERT+BiLSTM. These models were designed to address the limitation of BERT's maximum token limit of 512, which could lead to the potential omission of contextual information when dealing with long sequences. By integrating an LSTM-based model after BERT, the study aimed to mitigate this issue, with the BERT+BiLSTM configuration, in particular, demonstrating significant improvements over the vanilla BERT model. This work underscores the potential of hybrid models to compensate for the limitations of individual models while synergistically enhancing performance.

### Hierarchical Classification and High-Class Imbalance

The application of Transformer-based models has particularly excelled in leveraging the inherent hierarchy within CWE IDs, presenting an opportunity to enrich SVC with a more structured and informative approach to vulnerability classification. A novel approach, as explored in Fu et al. (2023), used a model distillation approach and addressed the prevalent class imbalance issue in real-world vulnerability data, specifically concerning CWE IDs. This study identified the severe class imbalance, often manifesting as a long-tailed label distribution, which can impede the learning process in deep learning models. The research offers a novel perspective on CWE ID classification by utilizing transformer-based hierarchical distillation. Such imbalances could lead transformer-based models to overfit certain prevalent CWE IDs while significantly underperforming on less common ones. Recognizing that techniques like Focal Loss and Logit Adjustment, typically employed in computer vision, could be adapted for textual data, this study used textCNN as a teacher model and transformer-based architectures such as CodeBERT, GraphCodeBERT, and CodeGPT as student models. This approach, termed hierarchical distillation, sought to mitigate class imbalances by grouping similar CWE IDs based on CWE abstract types into sub-distributions. They trained TextCNN teachers on individual simplified distributions, where they performed well only within their specific groups. Using a hierarchical knowledge distillation framework, they created a transformer student model to broaden their effectiveness in generalizing the TextCNN teachers' performance. A key innovation of this paper was maintaining compatibility with existing architectures by adding a special distillation token to the input, allowing such a hierarchical approach to be implemented without major architectural changes. While differing from the direct CWE ID classification in the current study,

this approach shares the objective of addressing class imbalances and learning the inherent hierarchy within the data. Furthermore, knowledge distillation provides an alternate perspective on integrating hierarchical structures instead of employing deep classifiers (Brust and Denzler, 2020), as in the current research.

# 3 Material & Methodology

## 3.1 Data Collection and Preprocessing

This sector outlines the process of data collection and the subsequent preprocessing steps undertaken to ensure the suitability and reliability of the chosen datasets. In line with our goal, we focus on gathering datasets that include various programming languages and a broad range of Common Weakness Enumeration Identifiers (CWE IDs). This strategy seeks to harness the layered structure of the CWE, pivoting towards a nuanced vulnerability detection. This section provides insights into the strategies employed to identify and process suitable datasets, contributing to the overall rigor and credibility of the research findings. For the data preprocessing phase, two datasets were employed for training purposes. To efficiently utilize CWE-IDs, the 'CWE-' prefix was stripped, retaining only the integer ID. Instances labeled as 'non-vulnerable' were reclassified as 0. The CWE hierarchy was transformed into a Directed Acyclic Graph (DAG) to integrate domain expertise. The label `0` represents a synthetic CWE ID and is absent from the genuine CWE hierarchy, so alterations were necessary. The foundational node of the CWE hierarchy is labeled `1000`. To streamline the application of hierarchical deep classifiers for hierarchical classification, it became imperative that the 'non-vulnerable' classification, denoted as `0`, also form a part of the CWE Hierarchy. As a solution, we synthetically introduced a new class node, `10000`, designating it as the novel root of the CWE hierarchy.
Subsequent preprocessing measures included the excision of duplicate entries and NaN values, rectification of labels, and omission of datasets that exhibited inconsistencies. We identified certain anomalies, such as non-vulnerable instances associated with CWE-IDs and vulnerable instances devoid of CWE-ID information. All data points maintaining these anomalies were discarded in the datasets. Additionally, datasets deviating from the CWE-ID hierarchical structure were excluded to maintain consistency. The specifics of preprocessing slightly varied depending on the datasets. Following preprocessing, it was transcribed into a CSV format and relabeled using only the integer ID of CWE-ID. The Big-Vul dataset, used extensively in Mining Software Repositories research and comprising 3,693 real-world code vulnerability instances from GitHub (including 886 vulnerable and 2,807 non-vulnerable instances across 91 CWE types and 348 projects), was originally a text file. It underwent preprocessing steps such as re-labeling and conversion into a CSV format for easier analysis. Similarly, the CVEfixes Dataset (v1.0.7), a substantial cybersecurity research resource updated until August 27, 2022, contains 7,798 vulnerability-fixing commits from 2,487 open-source projects. This dataset, detailing 7,637 CVEs and

209 CWE types and extensive documentation of source code changes in 29,309 files and 98,250 functions, was initially stored in an SQLite3 database. It was processed to extract relevant data through specific queries, and entries not meeting the designated CWE-ID criteria were excluded. Referring to the described database Tables 9, different queries, in Table 2, were used for the non-vulnerable and vulnerable datasets since the vulnerable datasets were related to the code before changing the method (changing the code to resolve vulnerable codes).

```
non_vul_query = """
SELECT cc.cwe_id, mc.code, cc.cve_id
FROM file_change f, fixes fx, cve cv, cwe_classification cc,
     method_change mc
WHERE f.hash = fx.hash
AND fx.cve_id = cv.cve_id
AND cv.cve_id = cc.cve_id
AND f.file_change_id = mc.file_change_id
AND mc.before_change = 'False'
"""

vul_query = """
SELECT cc.cwe_id, mc.code, cc.cve_id
FROM file_change f, fixes fx, cve cv, cwe_classification cc,
     method_change mc
WHERE f.hash = fx.hash
AND fx.cve_id = cv.cve_id
AND cv.cve_id = cc.cve_id
AND f.file_change_id = mc.file_change_id
AND mc.before_change = 'True'
AND cc.cwe_id IS NOT NULL
"""
```

Table 2: SQL Queries for Non-Vulnerable and Vulnerable Cases

A column named **vul**, signifying the binary classification, was appended. After these steps, the dataset was archived in CSV format. Conclusively, the columns across the two preprocessed datasets were merged, leading to a unified dataset featuring columns such as `code` (depicting the program code in a specific language), `vul` (indicating the binary classification), and `cwe_id` (representing the CWE-ID, stripped down to its integer part, and serving as the label for multiclass classification). This

meticulous preprocessing regimen ensured the datasets were primed for ensuing analyses and the development of vulnerability detection algorithms.

### 3.1.1 Balancing and Reassigning CWE IDs

Before splitting the dataset for vulnerability detection, a critical preprocessing step involved reassigning Common Weakness Enumeration (CWE) IDs. The dataset originally contained 171 unique CWE IDs, which served as target classes, excluding the broader nodes in the CWE hierarchy that could extend beyond 200 nodes, represented in Figure 17. However, the dataset exhibited significant class imbalance, a prevalent challenge in data-driven analyses. This imbalance was primarily evidenced by the disproportionate frequency of certain classes. Notably, the most frequent class was labeled 'non-vulnerable', indexed as `0`, constituting around 59,109 (32.3%) of the total 183,113 rows (Big-Vul dataset with 123,155 rows and CVEfixes dataset with 60,967 rows). This observation highlights a substantial imbalance in the data, revealing a "long tail" of classes with few instances, which starkly contrasts with the high frequency of this specific class.
Further compounding this imbalance, a subset of CWE ID categories was characterized by very few samples, with counts as minimal as 1 or 2. Such instances include CWE ID `23`, `27`, `80`, and `98`. The presence of these sparsely populated categories amplified the observed class imbalance, suggesting substantial underrepresentation. On the other hand, categories like CWE ID `20` (23,049 instances), `119` (27,331 instances), and `89` (7,111 instances) exhibited moderate representation. While these figures were more substantial than the sparse categories, compared to the most dominant class.
A strategic approach was employed to address this pronounced class imbalance and reduce data confusion. CWE IDs with less than 1,000 occurrences were reassigned to a CWE-ID higher in the hierarchy to ensure a minimum 5% class ratio for each CWE ID. The reassignment algorithm involved initially setting the root as the reassigned CWE ID for all entries and iteratively descending through each layer of the hierarchy if there are more than two child nodes with more occurrences than the cutoff (i.e., `1000`). Otherwise, descendants' nodes would be summed up to the parent nodes, and their CWE ID would be assigned to the parent's CWE ID. Reassigning samples with classes with less than 1000 samples had the effect of removing the corresponding CWE nodes. When the iteration is done, minor nodes less than the cutoff are removed because these nodes are unchangeable and remain less than 100. To prevent these very few nodes from violating the model performance similar to outliers could effect. This process allowed for consolidating minor CWE ID nodes into higher hierarchy levels without altering the hierarchy information. As a result, the total number of

target CWE IDs was reduced from 171 to 21 while preserving 96.9% of the data, with only about 3% being removed shown in its hierarchy structure in Figure 8.

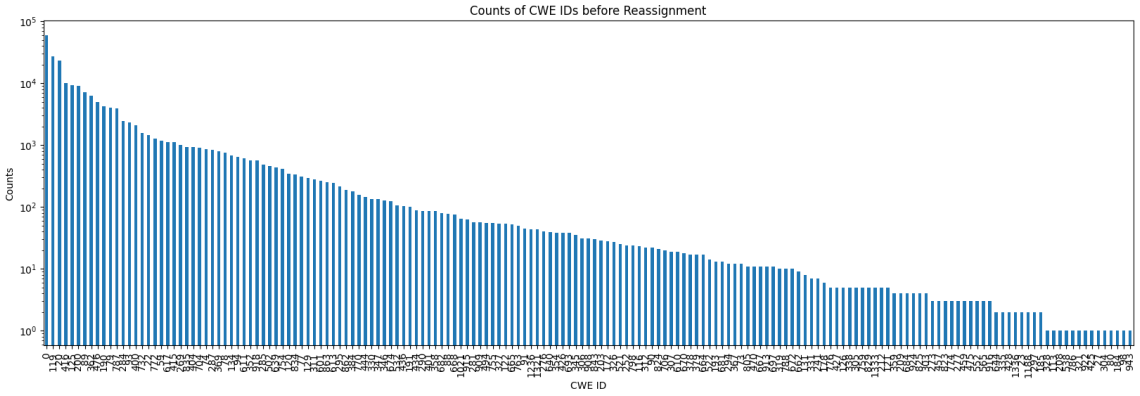



Figure 6: CWE ID Distribution in Before CWE Reassignment in Combined Dataset

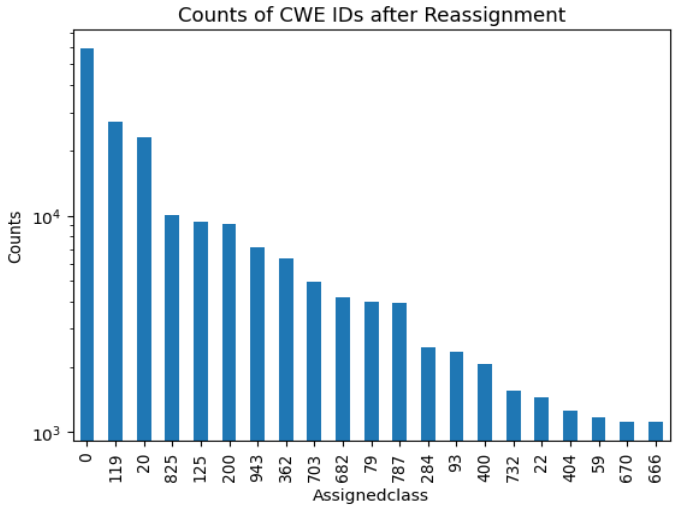



Figure 7: CWE ID Distribution in before CWE Reassignment in Combined Dataset

This CWE reassigning methodology ensures a more equitable representation of all classes within the dataset. Consequently, the robustness and generalizability of the analytical outcomes were significantly enhanced. The preprocessed CWE IDs were added to the dataset under the column `assignedcwe`, preceding the grouped stratified splitting method used for dividing the data into training, validation, and test sets.

### 3.1.2 Grouped Stratified Splitting

Effective data preprocessing is key for machine learning models in vulnerability detection in cybersecurity. This involves understanding Common Vulnerabilities and Exposures (CVE) and Common Weakness Enumeration (CWE) identifiers. A CVE ID is a unique identifier for a specific software or hardware vulnerability, detailing an individual security flaw. In contrast, a CWE ID represents a broader category of software weaknesses, describing types of vulnerabilities rather than specific instances. The distinction between these two identifiers is crucial: while a CVE ID points to a unique vulnerability, a CWE ID categorizes the general nature of the vulnerability. Our dataset preparation begins with grouping by CVE IDs, ensuring each unique vulnerability is contained within only one dataset subset (training, validation, or test), preventing data leakage, and guaranteeing the model is tested on completely unseen data. This is followed by stratification by CWE IDs, which ensures a balanced representation of various types of vulnerabilities across the datasets. This methodology prevents model bias towards certain vulnerability types and enhances its capability to generalize and detect a broad spectrum of cybersecurity threats.
In summary, the preprocessing phase was earmarked by:

- Integration and relabeling of CWE-IDs: By removing the 'CWE-' prefix and using the integer ID.
- Duplicate and NaN value elimination: Ensuring the quality and uniqueness of data entries.
- Conversion to pertinent labels: Ensuring that all data entries conform to a standardized labeling system.
- Omission of datasets displaying inconsistencies, Such as non-vulnerable instances having associated CWE-IDs or vulnerable instances lacking CWE-ID information.
- Adherence to the CWE-ID hierarchy: Excluding datasets that deviated from the stipulated hierarchical structure of CWE-IDs.
- Adding artificial nodes 0 and 10000 enables the integration of vulnerable and non-vulnerable categories within the same framework.
- Reassignment CWE-IDs to a high level of Hierarchy with certain criteria

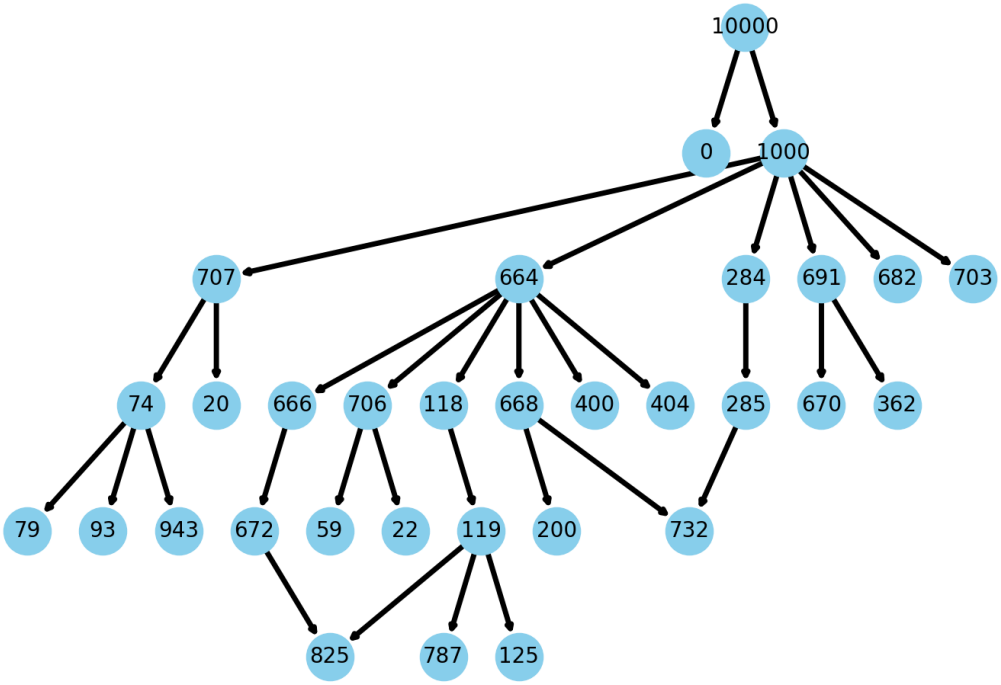



Figure 8: Directed Acyclic Graph of CWE Hierarchy after CWE Reassignment

After these preprocessing steps, each dataset underwent further tailored preprocessing. The MVD dataset, which was originally in a text format, was adapted to prevent any ambiguities that could hinder model performance. The MSR dataset and the CVEfixes dataset also underwent specific preprocessing activities. The result was a unified dataset with columns 'code', 'vul', and 'cwe_id', primed for subsequent analyses and model training. The culmination of preprocessing yielded a unified dataset optimized for ensuing analysis and training sessions.

## 3.2 Dataset Descriptions

In this study's preliminary phase of dataset selection, a key criterion was including a substantial number of Common Weakness Enumeration (CWE) identifiers. The threshold set for this parameter was a minimum of 30 CWE IDs per dataset. The primary objective of the research drove this requirement: to explore the influence of incorporating CWE hierarchical information on the performance of models in vul-

nerability detection tasks. Given the fundamental nature of this task as a multiclass classification, it is imperative to encompass a broad spectrum of hierarchical relationships within the CWE system.
Two datasets were chosen to fulfill this objective and maximize the potential for capturing the impact of these hierarchical relationships: the Big-Vul dataset and the CVEfixes dataset. The big-Vul dataset comprises 91 unique CWE IDs, whereas the CVEfixes dataset contains 209 CWE IDs. A thorough deduplication process was undertaken, eliminating overlapping CWE IDs across both datasets, culminating in a total of 171 unique CWE IDs for this research.
An additional advantage of both selected datasets is their accessibility. Sourced from open-source platforms, they are readily available for download from respective open-source repositories. This accessibility facilitates the replication of this study and promotes transparency and collaborative research within the field. By leveraging these comprehensive and publicly available datasets, the study aims to provide robust insights into the efficacy of using hierarchical information in CWE for enhancing vulnerability detection models.

### 3.2.1 Big-Vul Dataset (MSR)

The dataset often referred to in Mining Software Repositories (MSR) and known as the Big-Vul dataset Fan et al. (2020) represents an aggregation of real-world code vulnerabilities from open-source GitHub projects. It encompasses 3,754 code vulnerabilities, highlighting 91 distinct CWE types, all of which are sourced from 348 individual GitHub projects. After removing instances lacking CWE data, the dataset comprises 3,693 instances, including 886 vulnerable and 2,807 non-vulnerable instances. The Big-Vul dataset is crucial for in-depth analysis of code vulnerabilities, supporting various research directions. Its accompanying documentation offers detailed insights, and its GitHub repository provides open access for research purposes.

### 3.2.2 CVEfixes Dataset

The CVEfixes Dataset (version CVEfixes v1.0.7), as described in Bhandari et al. (2021), represents a significant advancement in the field of cybersecurity research, particularly in the analysis of vulnerabilities and their resolutions within open-source software ecosystems. This dataset, updated until August 27, 2022, is a modern and comprehensive resource for examining software vulnerabilities. It is intricately sourced from the National Vulnerability Database (NVD) and many public Git repositories, constituting a relational database of considerable breadth and depth.

The dataset encompasses an impressive array of 7,798 vulnerability-fixing commits drawn from a wide array of 2,487 distinct open-source projects. This extensive compilation covers 7,637 Common Vulnerabilities and Exposures (CVEs), spanning an array of 209 unique Common Weakness Enumeration (CWE) types. One of the standout features of the CVEfixes Dataset is its meticulous documentation of source code changes. It includes pre- and post-amendment versions of the source code, covering 29,309 files and 98,250 functions.

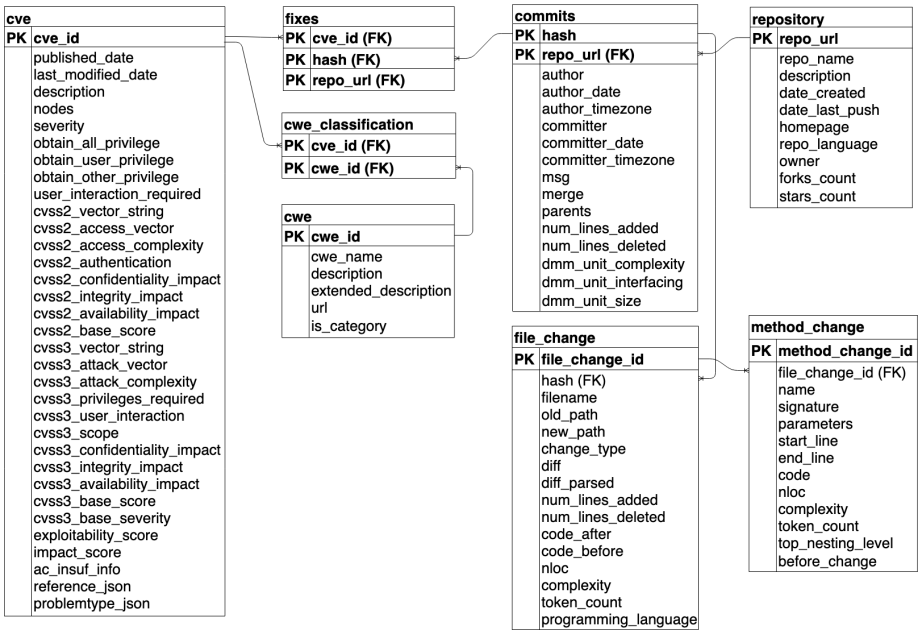



Figure 9: Entity Relation Diagram of CVEfixes Database (Bhandari et al., 2021)

In contrast to the Bigvul dataset, which is stored in a more straightforward format, as Figure 9 describes, this dataset is housed within a complex database comprising numerous tables and columns specifically designed to store code data. This structure necessitates meticulous querying to accurately extract the required dataset, particularly when considering the critical columns such as code_before and code_after. These columns represent code segments recorded before and after a method change. The distinction between these two states is crucial, as it directly impacts the dataset's relevance to the research question at hand.

When querying this dataset, it is imperative to exercise precision, as any inaccuracies in data extraction could result in the retrieval of an incorrect dataset. This, in turn, could have severe repercussions when the data is used as input for a machine learning model, particularly in the context of multiclass classification tasks. The model's

performance could be catastrophically affected if the input data does not accurately reflect the intended code changes or the nature of the programming language (PL) code-based texts.
The inherent characteristics of vulnerabilities in program codes further complicate this matter. Even minor modifications, which may seem negligible from a model's perspective, can significantly change program behavior and vulnerability type. As a result, datasets labeled as 'vulnerable' and 'non-vulnerable' must be carefully curated through distinct queries. These queries should account for the nuances in code differences precipitated by method changes. After this careful selection and extraction process, the datasets should be combined, following thorough cleaning and preprocessing steps. This rigorous approach is essential to ensure the integrity and applicability of the data for subsequent analysis and model training.

## 3.3 Domain Knowledge into Probabilistic Model

Detecting vulnerabilities in program source codes is paramount for ensuring security in both software and hardware systems. This study leverages domain knowledge, Common Weakness Enumeration (CWE) hierarchy, a structured framework that classifies prevalent software weaknesses. Common Weakness Enumeration (CWE) stands at the forefront of this effort. Developed as a categorization mechanism, CWE classifies software and hardware flaws. CWE aims to deepen the understanding of vulnerabilities through a community-driven project, paving the way for automated identification, rectification, and prevention tools. The inherent hierarchical structure of CWE, a significant domain knowledge in data security, offers potential advancements for vulnerability detection models. By weaving the CWE hierarchy into the research, the study draws upon the context and inter-relationships between CWE IDs. Therefore, this research aims to leverage the Common Weakness Enumeration (CWE) hierarchy to determine the effect of enhancing the accuracy and depth of vulnerability detection algorithms.
This section describes a methodology for how domain knowledge can be integrated and utilized for vulnerability detection. Through the contents below, you can learn how certain domain knowledge can be modeled, how to express this domain knowledge integration as a probability model, and more specifically, it describes in detail how domain knowledge is integrated and converted into a probabilistic model through the special label encoding and corresponding loss function that the probabilistic model utilizes.

### 3.3.1 CWE Hierarchy

The Common Weakness Enumeration (CWE) hierarchy is fundamental to this study. It is represented mathematically as a graph $W = (S, h)$, where $S$ represents a set of potential object categories (nodes), and $h$ signifies the hyponymy relations (edges) between these categories. The Relations such as hyponymy(is-a), antonymy(is-not), troponymy(is-a-way-of), and meronymy(is-part-of) are encoded in a graph structure, but this method uses hyponymy(is-a) relation, which is transitive in general. The set $S$ often includes a wider range of categories than those directly observed in the dataset, denoted as $C \subseteq S$. This wider range allows for the inclusion of broader, overarching categories. For example, as represented in reassigned CWE hierarchy DAG Figure 8, the category `CWE-707` serves as a precursor to both `CWE-74` and `CWE-20`, even if it is not explicitly present in the dataset.

The hyponymy relation $h \in S \times S$ is a key feature of this hierarchy, representing the directed connections in the graph. In simple terms, if $(s_1, s_2) \in h$, it indicates that $s_1$ is a broader category than $s_2$ or that $s_2$ is a specific instance within the category $s_1$. This hierarchy effectively captures the immediate class relationships and various levels of abstraction as graphical connections. It is important to note that this relationship is one-directional and non-repetitive, and the entire structure is assumed to be a directed acyclic graph (DAG), which is central to our analysis.

Within the CWE hierarchy, some nodes, such as the root node `10000` and its successor node `0`, are artificially added for structural completeness. These nodes, especially node `10000`, are designed to incorporate the non-vulnerable category (node 0) into the CWE framework. The actual CWE hierarchy begins with the root node `1000`. Adding nodes `0` and `10000` enables the integration of vulnerable and non-vulnerable categories within the same framework. This approach is adopted to align with the labeling methodology used in Brust and Denzler (2020), facilitating the application of specialized label encoding, loss calculation, and classification techniques in deep learning models.

Integrating the CWE hierarchy into our models involves transforming this structured knowledge into a probabilistic framework. This process includes special label encoding and the application of corresponding loss functions. Such adaptations are crucial for effectively applying the complex relationships within the CWE hierarchy to machine learning models. By utilizing deep learning techniques, this study aims to leverage the layered and interconnected nature of CWE IDs, translating them into a robust and sophisticated vulnerability detection system.

Incorporating the rich domain-specific expertise embedded within the CWE into vulnerability classification paradigms and observing the effect on model performance is the core part to watch in this research. To this end, publicly accessible datasets

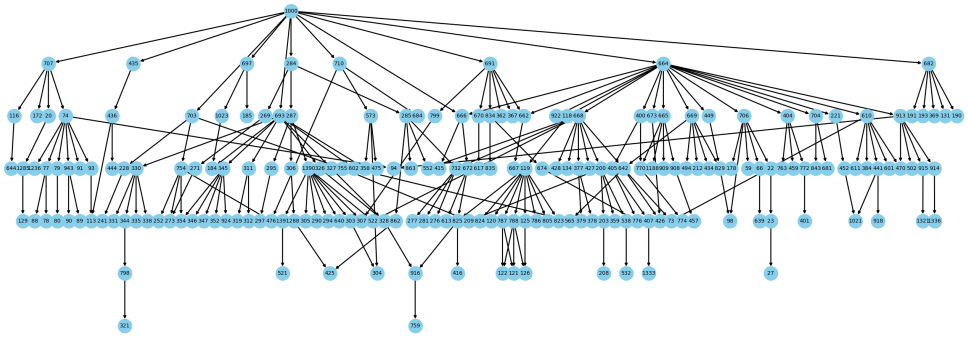


Figure 10: Directed Acyclic Graph of The Original CWE Hierarchy in the Datasets

were curated, encompassing a diverse array of programming languages and a staggering 21 distinct CWE IDs along with non-vulnerable synthetic CWE IDs. The structured paradigm provided by the CWE hierarchy facilitates an in-depth analysis of software vulnerabilities. Infusing this hierarchy into vulnerability detection mechanisms augments algorithmic efficiency by leveraging contextual relationships among CWE IDs. The aim is to bolster the accuracy and interpretability of vulnerability detection, thereby enhancing software vulnerability remediation measures. The meticulous data curation and preprocessing stages in this endeavor aim to harness the intricate hierarchical intricacies of CWE, with the ultimate goal of refining vulnerability detection accuracy.

### 3.3.2 Probabilistic Model

The method used to implement hierarchical classification using the CWE hierarchy structure, which is assumed to be supplied, was taken from Brust and Denzler (2020). This paper aims to integrate additional domain knowledge with classification using existing class hierarchies. It encodes properties such as class hierarchy into a probabilistic model. The derived novel label encoding and corresponding loss function can be used in a machine-learning environment.
Unlike the existing non-hierarchical methods, this probabilistic model utilizes the hierarchical structure, meaning that it uses all nodes in the hierarchy, not only the target label nodes. This method employs the hyponymy(is-a) relation to inform a class hierarchy. Given the hierarchical nature, a node cannot always be mutually exclusive. Therefore, instead of simply viewing the class label as a categorical random variable in this method, they consider each node as multiple independent Bernoulli

variables. This implies that a given example data $x$ can independently have multiple labels. Let $S = \{s_1, s_2, \ldots, s_c\}$ be a set of all classes in the hierarchy. Since the model considers the probability of any class $s$ occurring independently, it allows for even multi-label scenarios. For a given example $x$:

$$P(Y_s = 1|X = x), \tag{1}$$

in a more compact form,

$$P(Y_S^+|X). \tag{2}$$

In this approach, the model considers a hierarchical decomposition of classes, allowing for the possibility of co-occurring multiple independent parent classes. The assumptions are similar to those behind a one-hot encoding. Nevertheless, the model is restricted by additional properties based on the standard hierarchy definition.

**Hierarchical Decomposition** A class $s$ can have a number of separate parent classes, which they denote as $S' = s'_1, \ldots, s'_n$. We use $Y_{S'}^+$ to represent when we observe at least one of these parent classes and $Y_{S'}^-$ when none of the parent classes are observed:

$$Y_{S'}^+ \Leftrightarrow Y_{s'_1}^+ \vee \ldots \vee Y_{s'_n}^+ \Leftrightarrow Y_{s'_1} = 1 \vee \ldots \vee Y_{s'_n} = 1, \tag{3}$$

$$Y_{S'}^- \Leftrightarrow Y_{s'_1}^- \wedge \ldots \wedge Y_{s'_n}^- \Leftrightarrow Y_{s'_1} = 0 \wedge \ldots \wedge Y_{s'_n} = 0. \tag{4}$$

They make use of the hierarchical structure inherent to the data by assuming a marginalization of the conditional part of the model over the parents $Y'_s$:

$$P(Y_S^+|X) = P(Y_S^+|X, Y_S^+)P(Y_S^+|X) + P(Y_S^+|X, Y_S^-)P(Y_S^-|X). \tag{5}$$

**Simplification** To refine the model and incorporate assumptions that more accurately mirror the hierarchical challenge, they constrain the model and put on assumptions. If none of the parent classes $S' = s'_1, \ldots, s'_n$ of a given class $s$ are present, it is posited that the likelihood of observing $s$ in any example should be negligible:

$$P(Y_s^+|X, Y_{S'}^-) = P(Y_{s'}^+|Y_S^-) = 0. \tag{6}$$

Leading to a simpler model, discarding the latter part of Equation 5 by setting it to zero:

$$P(Y_s^+|X) = P(Y_s^+|X, Y_{S'}^+)P(Y_{S'}^+|X). \tag{7}$$

**Parental Independence** For recursive utility in the model, they envisage the random variables $Y_{s'_1}, \ldots, Y_{s'_n}$ to be mutually uninfluenced in a simplistic manner. Since each parent independently occurs to each other, we can use the product for simple calculations. By invoking the definition of $Y^+_{S'}$:

$$P(Y^+_{S'}|X) = 1 - \prod_{i=1}^{|S'|}[1 - P(Y^+_{s'_i}|X)]. \tag{8}$$

**Parentlessness** Within a DAG that is not void, they anticipate at least a single node without incoming edges, a class without parents. For instance, the '10000' node in our case. For a class $s$ with no observed parents, they postulate:

$$P(Y^+_s|X, S' = \emptyset) = 1. \tag{9}$$

It's critical to note that this is not universally applicable across all hierarchical classification scenarios. Specifically, even though there is only one root, if a hierarchy is constituted by numerous disjoint subsets, $P(Y_s|X, S' = \emptyset)$ should be explicitly defined.

### 3.3.3 Inference

In the previous section on the probabilistic model, we began by outlining how hierarchical information can be incorporated into a model. We explained the reasoning and methodically developed formulas to determine the conditional probabilities of all nodes within the hierarchy.

This section provides a concise overview of the model's inference methodology. In the inference process, the model filters out non-target labels, ensuring that the inferred labels are indeed the target labels. This is done by calculating the conditional probabilities for each node and using these probabilities to identify the most likely target label in the dataset. The node with the highest probability is then chosen for classification. This procedure uses a specific label encoding and a corresponding loss function within the probabilistic model.

Specific label encoding and a corresponding loss function are crucial to the model inference. Specific label encoding gives the correct labels regarding the corresponding example target in the model's hierarchy. The loss function is designed to calculate the loss based on all the nodes in the hierarchy but leverages a loss mask, which masks unrelated nodes in the hierarchy. In this way, the model can only focus on relevant nodes in the hierarchy and prevent confusion from all nodes, which might

overwhelm the model depending on the size of the hierarchy. The predictions made through this inference are then connected to subsequent learning processes, bridging the gap between theoretical understanding and practical application.

In certain scenarios, the model's output can be limited to the classes $C$ present in the data instead of considering every class $S$ modeled, including their parent classes. This approach is based on having a predefined set of classes during testing, rather than addressing a problem with an undefined class set. It refers to this scenario as *mandatory labeled node prediction* (MLNP), contrasting with the more flexible *arbitrary node prediction* (ANP). For predicting a specific class $s$ from a given example $x$, the process involves identifying the class with a high likelihood of two concurrent factors: (i) the occurrence of the class ($Y_s^+$) and (ii) the non-occurrence of any child class ($Y_{S''}^-$).

Inference is conducted by extending the method similar to what was defined in Equation 8. The selection of class $s(x)$ from a sample $x$ is done by maximizing the product of the probability of observing a parent class and the probability of not observing any child class, formalized as:

$$s(x) = \underset{s \in S \subseteq CS}{\operatorname{argmax}} \; P(Y_s^+|X) \prod_{i=1}^{|S''|} 1 - P(Y_{s_i''}^+|X, Y_s^+). \tag{10}$$

Adopting the assumption of pairwise independence among the children, inference in the manner outlined below:

$$s(x) = \underset{s \in CS}{\operatorname{argmax}} \; P(Y_s^+|X) \prod_{i=1}^{|S'|} 1 - P(Y_{s''}^+|X, Y_s^+). \tag{11}$$

Acknowledging the ability to decompose $P(Y_s^+|X)$ as performed in Equation 5 and displayed as a product in Equation 8:

$$s(x) = \underset{s \in CS}{\operatorname{argmax}} \; P(Y_s^+|X, Y_{S'}^+) \cdot (1 - \prod_{i=1}^{|S'|} 1 - P(Y_{s_i'}|X)) \cdot \prod_{i=1}^{|S'|} 1 - P(Y_{s''}^+|X, Y_s^+). \tag{12}$$

### 3.3.4 Label Encoding and Loss Function

Moving on to the training phase, applying our model within a machine learning context focuses on estimating the conditional probabilities rather than modeling $P(Y_s^+|X)$ for each class directly. This slight shift in approach refines the task of the learning algorithm, narrowing it down to discrimination among siblings in the hierarchy and facilitating the use of the hierarchical recursive inference as described previously.

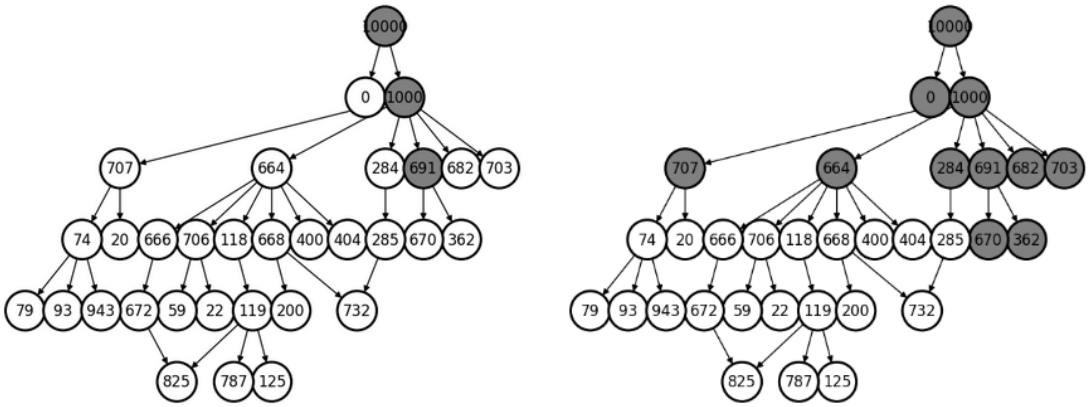



Figure 11: Encoding(left) and Loss Mask(right) for Reassigned CWE Hierarchy (criteria: `CWE-691`)

#### Label Encoding

Key elements of the approach include a method for label encoding $e : S \to \{0,1\}^{|S|}$ and a specific type of loss function. In Figure 11, this approach encodes a label $y \in S$ by utilizing the hyponymy relationship $h \in S \times S$, particularly its transitive closure $T(h)$, with the encoding defined as follows:

$$e(y)_s = \begin{cases} 1 & \text{if } y = s \text{ or } (y,s) \in T(h), \\ 0 & \text{otherwise.} \end{cases} \tag{11}$$

However, we need an appropriate loss function that maintains the conditional structure of each estimator. This means if we have a label $y$, we should only train a component $s$ if one of its related parents $s'$ is also related to $y$ through $T(h)$, or if $y$ is a parent itself. We fulfill this with a loss mask $m : S \to \{0,1\}^{|S|}$, described as:

### Loss Mask

A specific loss function with a loss mask is introduced to facilitate model training with the prescribed label encoding. In Figure 11, the **loss mask** $m(s)$ is a function that identifies the relevant classes for calculating the loss during the training of a hierarchical classification model. In this way, the loss mask $m(s)$ ensures that a component $s$ should only be trained if its corresponding label $y$ is present in the hierarchy relative to $s$. It is defined as:

$$m(s) = \begin{cases} 1 & \text{if } y = s \text{ or } \exists(s, s') \in h : y = s' \text{ or } (y, s') \in T(h), \\ 0 & \text{otherwise.} \end{cases} \tag{13}$$

where $s$ is a predicted class, $y$ is the true class label, and $T(h)$ represents the transitive closure of the hyponymy relation $h$ defining the class hierarchy. It filters the relevant labels according to the hierarchy by setting component $s$ as one if it is relevant to the current prediction, its descendants, and its transitive closure. The mask ensures that the loss is computed only for the correct class during training, facilitating the learning of hierarchical relationships. That is, the loss mask $m(s)$ directs the focus of the model's training process by highlighting relevant class labels.
The intuition behind the loss mask is to penalize the model for incorrect predictions in a way sensitive to the classes' hierarchical structure. For example, as represented in Figure 11, if the true label is `CWE-691`, which is a subclass of `CWE-1000`, the model should not be penalized as much for predicting `CWE-284` (another subclass of `CWE-1000`) as it would be for predicting a completely different class, e.g., `CWE-0` or `CWE-732`, etc. Using the loss mask ensures that the model's mistakes are evaluated in the context of the hierarchy, encouraging it to learn the structure of the classes rather than treating all mistakes equally.

### Loss Function

An objective function $f(x)_s$ is formulated to approximate the conditional probabilities $P(Y_s^+|X, Y_{S'}^+)$. The estimator function, $f : X \to [0, 1]^{|\mathcal{S}|}$, is employed together with the encoding and loss mask to calculate the loss for a particular data point $(x, y)$. The objective function $L$ is expressed as:

$$L(y, f) = m(y)^T(e(y) - f(x)^2). \tag{14}$$

The minimization of this objective function facilitates the learning algorithm's ability to estimate conditional probabilities while respecting the hierarchical structure of the data.

### 3.3.5 Comparing Four Loss Weight Methods in Hierarchical Classification

This subsection outlines our approach to comparing four distinct loss weight methods in hierarchical classification by varying the focus on different hierarchical characteristics. In our hierarchical classification approach, we implement four distinct loss weight methods: `default`, `equalize`, `descendants`, and `reachable_leaf_nodes` referring to Brust and Denzler (2020). Each of these methods assigns different weights to the nodes in the hierarchy, influencing how the model learns and interprets the semantic knowledge within the hierarchical structure.

**Inversed Class Frequency in Hierarchical Setup:** A critical step in our method involves calculating the inversed class frequency for all nodes in the hierarchy, including the target nodes. This frequency measures how often each node is 'affected' in the hierarchy, whether as a target node, an ancestor of a target node, or a successor of an ancestor. This calculation forms the basis for initializing the loss weights in `equalize`, `descendants`, and `reachable_leaf_nodes` methods.

**Differentiating the Four Loss Weight Methods**

1. **Default Loss Weight:** Assigns a uniform class weight of 1 to all classes, treating each node equally regardless of its position or frequency in the hierarchy.
2. **Equalize Loss Weight:** The loss weight for each node is initialized to its inversed class frequency. This method balances the influence of frequently and infrequently occurring nodes in the hierarchy.
3. **Descendants Loss Weight:** Each node's loss weight is adjusted by multiplying it by the number of its descendants and then adding one. This method emphasizes nodes with more descendants, potentially reflecting their broader impact in the hierarchy.
4. **Reachable Leaf Nodes Loss Weight:** Modifies each node's loss weight by multiplying it by the number of reachable target nodes (nodes present in the target labels and thus predictable). Considering their hierarchical position, this approach focuses on the nodes' direct relevance to the target labels.

### 3.3.6 Expected Outcomes and Rationale of the Method

This hierarchical approach with Deep Classifier (Brust and Denzler, 2020) is selected to effectively tackle challenges that need to be resolved, increasing the research task's difficulty and complexity.

Our research involves multiclass classification of CWE IDs using real-world datasets with detailed CWE hierarchy structures and long-tailed distribution, as explained in Sections 3.1 and 3.2. This complexity required a method capable of addressing significant class imbalances and effectively learning semantic knowledge representation. Since CWE IDs have a hierarchical structure, using this information with a deep classifier in hierarchical classification seemed promising. We expected that integrating this hierarchical label data, instead of just using it for straightforward classification, would improve model performance.

Unlike conventional classification methods that directly utilize a pre-trained language model's output for target label prediction, hierarchical classification processes this output through a deep classifier, applying it across the entire hierarchy. This technique enables a more nuanced focus on the nodes' interconnectedness and hierarchical relationships, facilitating a more in-depth semantic and contextual analysis.

As mentioned in Section 3.3.5, our experiment involves testing four different loss weight methods: equalize, descendants, and reachable leaf nodes. We aim to determine how these methods impact the model's performance, particularly assessing the benefits of hierarchical information in the default loss weight method against traditional methods like cross-entropy loss. Furthermore, we plan to evaluate the effectiveness of class-weight-based methods like Focal Loss compared to standard classification approaches.

The anticipated outcome of integrating hierarchical methods into our research is higher classification results performance than non-hierarchical methods. We hypothesize that the more complex the hierarchy information, the better the performance yielded by the model based on hierarchical methods. When combined with transformer-based models, particularly those pre-trained on programming codes like CodeBERT and GraphCodeBERT, the effectiveness in classifying CWE IDs in complex dataset settings is expected to be significantly enhanced. The ability to use different loss weights could be expected to prove beneficial in addressing class imbalances, thereby helping to mitigate the challenges encountered in our research.

In summary, we anticipate that the hierarchical classification method, especially when integrated with advanced transformer models and tailored loss weight strategies, will lead to superior performance in our complex multiclass classification task. This approach is expected to effectively harness the nuanced hierarchy of CWE IDs and help the unique challenges of our research.

## 3.4 Evaluation Metrics

### Binary Classification

In binary classification, outcomes are categorized into two classes: positive and negative, with True Positives (TP), False Positives (FP), True Negatives (TN), and False Negatives (FN).

- **Accuracy** – Proportion of correct predictions (TP and TN) among total cases. It is useful overall but can be misleading in imbalanced datasets.

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

- **Balanced Accuracy** – Average of recall for each class. Suitable for imbalanced datasets.

$$\text{Balanced Accuracy} = \frac{1}{2}\left(\frac{TP}{TP + FN} + \frac{TN}{TN + FP}\right)$$

- **Precision** – Proportion of correctly identified positives. Crucial when false positives are costly.

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

- **Recall (Sensitivity)** – Proportion of actual positives correctly identified. Key when false negatives are costly.

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

- **F1 Score** – Harmonic mean of precision and recall. Useful for balancing the two.

$$\text{F1 Score} = 2 \times \frac{\text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}}$$

### Multiclass Classification

Multiclass classification deals with more than two classes, evaluating each class in a "one versus all" manner.

- **Accuracy** – Proportion of true results among total cases. It can be misleading in imbalanced datasets.

$$\text{Accuracy} = \frac{\sum \text{Correct Predictions}}{\sum \text{Total Predictions}}$$

- **Balanced Accuracy** – Average recall for each class, $i$, where $N$ is the number of classes. Important in imbalanced scenarios.

$$\text{Balanced Accuracy} = \frac{1}{N} \sum_{i=1}^{N} \frac{TP_i}{TP_i + FN_i}$$

- **Precision and Recall** – Calculated and averaged for each class, $i$.

$$\text{Precision}_i = \frac{TP_i}{TP_i + FP_i}$$

$$\text{Recall}_i = \frac{TP_i}{TP_i + FN_i}$$

- **Weighted F1 Score** – This is the average of the F1 scores for each class, weighted by the number of true instances for each class. It's a more informative metric in the context of imbalanced datasets.

$$\text{Weighted F1 Score} = \sum_{i=1}^{n} w_i \times \text{F1 Score}_i$$

  where $w_i$ is the proportion of true instances for class $i$ in the dataset

- **Macro-average F1 Score** – Arithmetic mean of F1 Scores of all classes, $i$, where $N$ is the number of classes. Robust against class imbalance.

$$\text{Macro-average F1 Score} = \frac{1}{N} \sum_{i=1}^{N} F1_i$$

# 4 Experiments & Results

This chapter details the experimental setup and results aimed at assessing the effectiveness of deep learning models in vulnerability detection within code. Utilizing CodeBERT and GraphCodeBERT as base pre-trained models from Microsoft, this study focuses on fine-tuning these models for a vulnerability detection downstream task. The loss function for the hierarchical model is based on binary cross-entropy (BCE), which is applied to each node within the CWE hierarchy. Four options for loss weighting are explored: `default`, `equalize`, `descendants`, and `reachable_leaf_nodes`. Each option assigns varying weights to nodes within the hierarchy, with `default` providing no additional weighting. In contrast, the standard multiclass classification employs Cross-Entropy and Focal Loss, with Focal Loss being particularly relevant for addressing class imbalances analogous to the hierarchical model's weighting schemes. The core of the experimentation involves a multiclass classification task, predicting specific CWE IDs (Common Weakness Enumeration identifiers) representing various vulnerabilities in code. The subsequent sections will present the detailed methodology of each experiment, followed by the results and a comprehensive analysis comparing the performance of the hierarchical and non-hierarchical approaches in vulnerability classification.

## 4.1 Dataset and Models

For this experiment, two Programming Language(PL)-based Transformer-based models, CodeBERT and GraphCodeBERT, are used. Both models are BERT-based models and implemented by Microsoft. The models are fine-tuned using a combined dataset comprising MSR (Mining Software Repositories) and CVEfixes datasets. This diverse dataset provides a robust foundation for assessing the models' capabilities in identifying various vulnerabilities. The dataset partitioned into training, validation, and test subsets, underwent stratification to maintain a uniform distribution of Common Weakness Enumeration (CWE) IDs. Instances with the same CVE ID were assigned to the same set to prevent redundancy.

### 4.1.1 Transformer-based Model Architectures

Transformer-based pre-trained models form the backbone of our approach. This model is designed for binary classification. In multiclass classification, the models are designed to identify one of the 21 unique CWE tags, including an artificial class `0`, representing 'non-vulnerable.' While non-hierarchical models adapt pre-trained architectures with specific loss functions, hierarchical models necessitate a custom

approach with a deep hierarchical classifier to manage the complex structure of the CWE hierarchy.

## 4.2 Research Questions

1. **Impact of Domain Knowledge Integration:** How does integrating domain-specific knowledge, particularly the Common Weakness Enumeration (CWE) hierarchy, affect the performance of deep learning models in vulnerability detection tasks?

2. **Effectiveness of Loss Functions and Weighting Schemes:** How do different loss functions and their associated weights influence the outcomes of hierarchical versus non-hierarchical classification models?

3. **Comparative Analysis of Classification Approaches:** How does the hierarchical multiclass classification strategy, employing various loss weights (`default`, `equalize`, `descendants`, `reachable_leaf_nodes`), compare with the standard multiclass classification in terms of predicting CWE IDs in code? How do these methods address the challenge of class imbalance and model performance consistency?

## 4.3 Objectives of the Experiments

The primary objective of this study is multifaceted, aiming to delve into the nuances of vulnerability detection using Transformer-based language models and to assess the impact of integrating domain-specific knowledge into these models. Key areas of focus include:

1. **Evaluating Hierarchical versus Standard Classification Approaches:** This involves a comparative analysis of hierarchical and standard classification methodologies to determine their respective efficacies in vulnerability detection tasks. The intent is to discern each approach's potential advantages and drawbacks, particularly in dealing with the complexities inherent in software vulnerability detection.

2. **Impact of Domain Knowledge Integration:** This research investigates how the incorporation of domain knowledge, specifically the Common Weakness Enumeration (CWE) hierarchy, affects the performance of deep learning methods in classification tasks. This exploration aims to highlight the value of

domain knowledge in refining the predictive capabilities of transformer-based models.

3. **Addressing Class Imbalance:** Given that class imbalance is a prevalent issue in vulnerability detection datasets, the experiments are designed to evaluate how effectively this imbalance can be mitigated. By exploring various methodologies, the study seeks to contribute to developing more balanced and equitable deep learning models in cybersecurity.

Through these objectives, the research aims to advance the understanding of transformer-based models in vulnerability detection, emphasizing the integration of domain knowledge as a pivotal element for enhancing model accuracy and effectiveness.

## 4.4 Fine-tuning

The experimental framework involves fine-tuning two preeminent pre-trained models, CodeBERT and GraphCodeBERT, utilizing a comprehensive dataset from MSR and CVEfixes. The primary goal is to employ these models in a multiclass classification setting, focusing on accurately predicting specific CWE IDs indicative of vulnerabilities in software code.
The combination for the experiments was designed to be 12 in total. The number of experiments can be calculated by the number of models and number of loss methods, which is 6, including two loss functions for standard classification and four loss weight options for hierarchical classification.

### 4.4.1 Classification Tasks with Various Loss Methods

Our experiments compared the standard and hierarchical classifications with different model configurations.

- **Standard Multiclass Classification:**
  - Utilizing Cross Entropy Loss to establish a baseline performance.
  - Implementing Focal Loss, a class weight-based loss function, to address class imbalances.
- **Hierarchical Multiclass Classification (Deep Classifier Method):**
  - Utilizing a unique classifier layer with an embedding function and a loss function tailored to the hierarchical nature of the CWE IDs.

  - Applying a range of loss weighting schemes to evaluate their impact on model performance:
    * **Default:** No additional weighting, serving as a control setup.
    * **Equalize:** Aiming to equal weighting for all node classes by inversed class frequency from base class weight, inversed class frequency.
    * **Descendants:** Focusing on hierarchical descendant classes.
    * **Reachable Leaf Nodes:** Targeting the reachable leaf node in target label classes from base class weight, inversed class frequency.

Each classification approach is designed to assess the models' efficacy in vulnerability detection and evaluate how different methodologies impact the mitigation of class imbalance and the overall consistency and bias in model performance.
The experiments involve fine-tuning two base pre-trained models, CodeBERT and GraphCodeBERT, with a combined dataset (MSR+CVEfixes) for a multiclass classification task. The task aims to predict specific CWE IDs (vulnerabilities) in code.

### 4.4.2 Model Configurations

| **Model** | **Loss Function** | **Loss Weights** | **Classification Type** |
|---|---|---|---|
| CodeBERT | Cross Entropy, Focal Loss | Default, Class Weights | Non-Hierarchical |
| GraphCodeBERT | Cross Entropy, Focal Loss | Default, Class Weights | Non-Hierarchical |
| CodeBERT with Hierarchical Classifier | BCE (per node) | Default, Equalize, Descendants, Reachable Leaf Nodes | Hierarchical |
| GraphCodeBERT with Hierarchical Classifier | BCE (per node) | Default, Equalize, Descendants, Reachable Leaf Nodes | Hierarchical |

### 4.4.3 Loss Function Details

- **Binary Cross-Entropy (BCE) Loss**: They are applied to each node in the CWE hierarchy for hierarchical classification. Different weight options will be tested. Binary Cross-Entropy Loss is used in binary classification tasks. It calculates the Loss for each class separately and then averages them.

$$\text{Binary Cross-Entropy Loss} = -\frac{1}{N}\sum_{i=1}^{N}[y_i \log(p_i) + (1 - y_i)\log(1 - p_i)]$$

  Where $N$ is the number of observations, $y_i$ is the true label, and $p_i$ is the predicted probability for the $i^{th}$ observation.

- **Cross Entropy Loss**: Used for non-hierarchical classification to be compared with the `default` loss weight option, equivalent to no specific weighting in hierarchical classification. Cross Entropy Loss measures the performance of a classification model whose output is a probability between 0 and 1. It increases as the predicted probability diverges from the actual label.

$$\text{Cross Entropy Loss} = -\sum_{c=1}^{M} y_{o,c}\log(p_{o,c})$$

  Here, $y_{o,c}$ is a binary indicator for whether class label $c$ is the correct classification for observation $o$, and $p_{o,c}$ is the predicted probability of word $o$ being of class $c$.

- **Focal Loss**: Utilized for addressing class imbalance. Compatible with the `equalize`, `descendants`, and `reachable_leaf_nodes` weight options in hierarchical models. Focal Loss addresses class imbalance by focusing on hard-to-classify examples. It modifies the cross-entropy criterion to focus more on difficult cases.

$$\text{Focal Loss} = -\alpha_t \sum_{c=1}^{M}(1 - p_{t,c})^{\gamma} y_{t,c}\log(p_{t,c})$$

  Here, $\alpha_t$ is a weighting factor for class $t$, $p_{t,c}$ is the predicted probability of the true class $c$ for the $t^{th}$ observation, $y_{t,c}$ is the true label, and $\gamma$ is the focusing parameter, typically greater than 0.

The distinction between Hierarchical and Non-Hierarchical classification lies in utilizing all nodes in the CWE hierarchy versus targeting specific nodes. This design aims to assess the impact of domain knowledge integration on the effectiveness of deep learning classification methods.

### 4.4.4 Hyperparameter Optimization (HPO)

In this investigation, Hyperparameter Optimization (HPO) entails the meticulous adjustment of four pivotal hyperparameters integral to enhancing the efficacy of the utilized classification model. The HPO framework employed is Optuna , a sophisticated tool for hyperparameter tuning. The models' hyperparameter optimization was conducted with 4 GPU resources with 100GB of memory (Tesla V100-SXM2-32GB, NVIDIA A100-SXM4-40GB) and 24 CPUs per task. The hyperparameters subjected to optimization are:

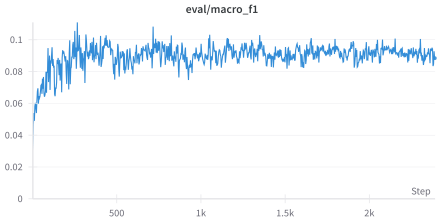
1. **Classifier Learning Rate:** This parameter governs the magnitude of updates to the classifier's parameters, influencing the progression of gradient descent. A range of $[10^{-5}, 10^{-1}]$ is explored with logarithmic scaling to ensure a comprehensive examination of its impact.

2. **Classifier Factor:** Distinct learning rates for the base model and the classifier are established through this parameter. Its utility maintains a lower learning rate for the base model relative to the classifier. The classifier factor defines the base learning rate as a function of the classifier learning rate.

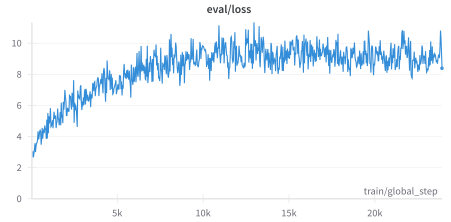
3. **Gradient Accumulation Steps:** Constrained by GPU memory, the maximum batch size was capped at 32. To circumvent this, gradient accumulation was employed, aggregating gradients over multiple sub-batches to simulate larger batch sizes. This parameter was sampled within the range of $[1e1, 1e3]$.

4. **Weight Decay:** A regularization parameter, weight decay, modulates the extent of penalty applied to the model's parameters during training. It was sampled similarly with logarithmic scaling within the range $[10^{-5}, 10^{-1}]$.

For efficient HPO, a Pruner and Parallelism method was implemented. The Pruner, particularly the HyperbandPruner, offers advantages in multiclass classification due to its ability to eliminate less promising hyperparameter configurations rapidly. Parallelism in Optuna leverages a database to store all experiment data, allowing models to share information and accelerating the HPO process. The Pruner also accesses this database to determine the feasibility of continuing the current study.

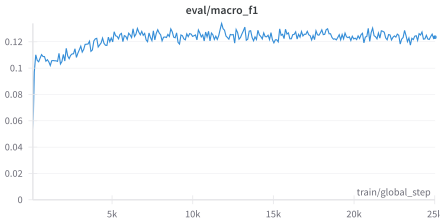



(a) Macro F1-Score for hCodeBERT-descendants

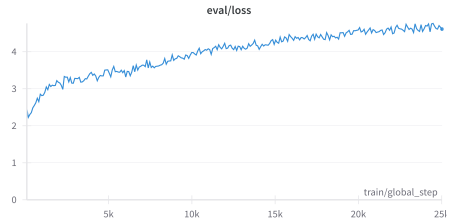



(b) Evaluation Loss for hCodeBERT-descendants

Figure 12: Evaluating hCodeBERT-Descendants: Macro F1-Score and Loss Metrics




(a) Macro F1-Score for GraphCodeBERT-FL




(b) Evaluation Loss for GraphCodeBERT-FL

Figure 13: Evaluating GraphCodeBERT-FL(Focal Loss): Macro F1-Score and Loss Metrics

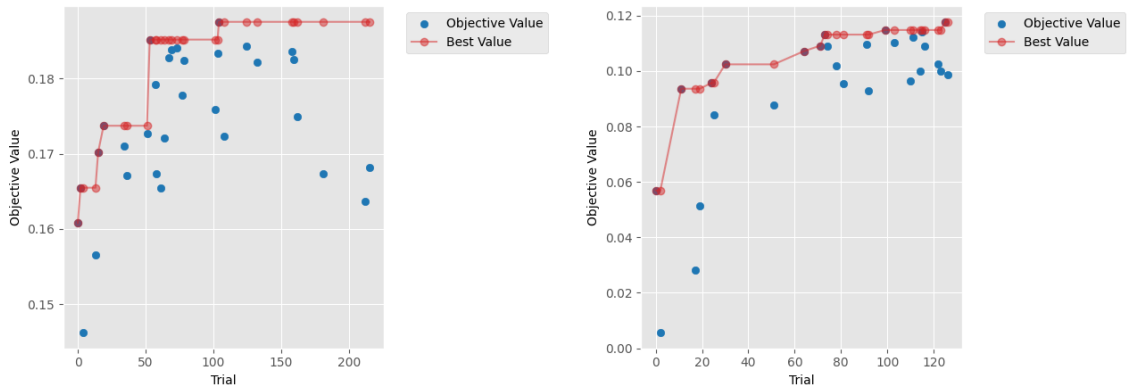



Figure 14: Hyperparameter Optimization History Plots of Two Best Models (left: GraphCodeBERT-FL, right: hCodeBERT-descendants)

In Figures 12 and 13 show the HPO results of evaluation loss and macro f1-score of the two best-performed models, *hCodeBERT-descendants* and *GraphCodeBERT-FL*. The macro f1-score is used as an evaluation metric for HPO. As shown in figure 12a and 13a, the Focal Loss-based models, *GraphCodeBERT-FL*, slowly converged, so it had relatively long training steps(25k) compared to another loss-based models in HPO runs, e.g. *hCodeBERT-descendants* took 2.5k steps. The evaluation loss plots in figure 12b and 13b for both models show they were overfitting, so they had to do the early stopping. Especially for hierarchical method-based models that link so fast that they had to stop early, the final models were selected by the best macro f1-score result. The best hyperparameters from HPO results are listed in Table 3.
In Figure 14, the hyperparameter optimization history plots of two best-performed models, *GraphCodeBERT-FL* and *hCodeBERT-descendants*, represent the read point is the best macro f1-score value and both best objective value points increased slowly over trials. This indicates that HPO went as it was supposed to. More plots for all CodeBERT and GraphCodeBERT models are shown in Figures 19 and 20. In Figures 21 and 22, the hyperparameter slice plots for CodeBERT and GraphCodeBERT models can help to set better hyperparameter range by reading their tendencies.

### 4.4.5 Advanced Methods in Model Fine-Tuning

In machine learning, addressing challenges such as memory limitations and ensuring efficient and practical training of complex models, such as transformer-based architectures, necessitates advanced Hyperparameter Optimization (HPO) techniques and fine-tuning strategies.

- **On-the-fly Tokenization** This method dynamically adapts to new data, enhancing flexibility and memory efficiency by eliminating the need for storing preprocessed tokens. Ideal for real-time applications like live translation, it allows tailored preprocessing and improves model generalization by exposing it to a diverse linguistic range, thereby enhancing performance on varied datasets.

- **Gradient Accumulation** A technique to train large-scale neural networks within memory constraints by accumulating gradients across smaller mini-batches before updating model weights. Facilitates training with larger batch sizes without a proportional increase in memory requirement, which is beneficial for large models.

- **Tree-structured Parzen Estimator (TPE) Sampler** A Bayesian optimization approach that predicts promising hyperparameter configurations based on

historical data. Efficient in high-dimensional spaces and faster in finding optimal hyperparameters than traditional search methods.

- **HyperbandPruner** Utilizes the multi-armed bandit principle to dynamically allocate resources to hyperparameter configurations, focusing on the most promising ones. Reduces computational expenses in hyperparameter tuning, particularly in scenarios with numerous hyperparameters.
- **Adaptive Learning Rate Strategy** It involves using distinct learning rates for pre-trained models and classifiers, with a mechanism to keep the pre-trained model's learning rate lower. It prevents catastrophic forgetting in pre-trained models while allowing classifiers to be flexible and adaptable.
- **Parallelism** Utilizes parallelism to speed up hyperparameter optimization by distributing the evaluation of different hyperparameter sets across multiple processors or machines. Instead of testing configurations sequentially, Optuna tests them concurrently, reducing the time needed to find the most effective hyperparameters for a machine-learning model. This approach parallelizes the hyperparameter search process, not the training of individual models.

These methods collectively enhance the Hyperparameter optimization and fine-tuning process, ensuring optimal model performance while navigating the constraints of hardware resources and the complexity of model architectures.

## 4.5 Evaluations

The foundational models employed are CodeBERT and GraphCodeBERT, with 'h' prefixed models indicating the incorporation of a hierarchical classifier. The study conducted experiments across twelve distinct model configurations. For binary classification, evaluation metrics included accuracy, precision, recall, and F1-score. Multiclass classification assessments utilized accuracy, balanced accuracy, macro F1-score, and weighted F1-score to gauge model performance.

### 4.5.1 Binary Classification

First, since the model is initially fine-tuned for multiclass classification tasks, the model focuses on improving model performance in predicting class as balanced as possible. Therefore, the validation set was balanced, and the evaluation metric was the **macro F1-score**, the average F1-score per class. The model did not focus on predicting the 0 type as accurately as possible but tried to predict other classes

more. This can significantly decrease the binary classification performance. Hence, the binary classification performance is not as good as related research, which shows a high vulnerability detection performance in binary classification tasks. This fact should be taken into account as an essential fact for analysis.

As per the aforementioned fine-tuning objective, the models are not fine-tuned for binary classification problems, so the binary prediction results are transformed into binary labels with multiclass prediction labels. All prediction results were changed in a way that converted all non-zero labels to 1 and left the `0` label as it was.

| **Model** | **Accuracy** | **Precision** | **Recall** | **F1-Score** |
|---|---|---|---|---|
| CodeBERT-CE | 26.08 | 0.00 | 0.00 | 0.00 |
| GraphCodeBERT-CE | 26.08 | 0.00 | 0.00 | 0.00 |
| CodeBERT-FL | **73.36** | 75.92 | **93.67** | **83.87** |
| GraphCodeBERT-FL | **73.81** | 76.54 | **93.10** | **84.01** |
| hCodeBERT-default | 69.62 | **80.00** | 78.52 | 79.26 |
| hGraphCodeBERT-default | 71.74 | 78.18 | 85.68 | 81.76 |
| hCodeBERT-equalize | 70.39 | 78.17 | 83.17 | 80.59 |
| hGraphCodeBERT-equalize | 69.01 | 78.11 | 80.70 | 79.38 |
| hCodeBERT-descendants | 70.26 | 79.93 | 79.81 | 79.87 |
| hGraphCodeBERT-descendants | 70.16 | 79.20 | 80.87 | 80.03 |
| hCodeBERT-reachable_leaf_nodes | **73.79** | **80.43** | 85.30 | 82.79 |
| hGraphCodeBERT-reachable_leaf_nodes | 69.14 | 79.22 | 78.96 | 79.09 |

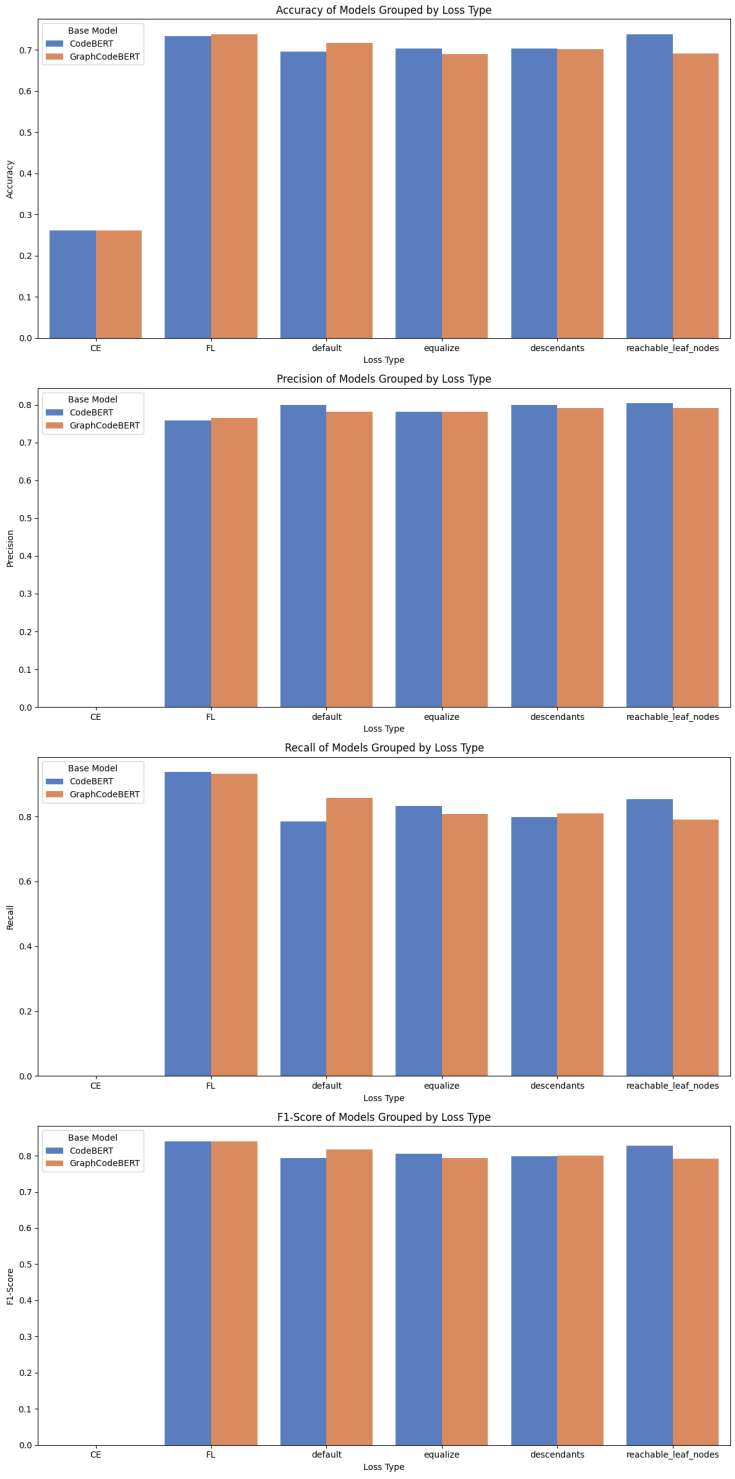



Figure 15: Model Performance Plot for Binaryclass Classifiation

### 4.5.2 Multiclass Classification

As previously mentioned in the binary classification results, this study has been fine-tuned to address a multiclass classification problem, with training focused on enabling the model to predict across classes as evenly as possible, considering the severe class imbalance in the data. In such scenarios, simple metrics like accuracy or F1-score may not be ideal for evaluation because the prevalence of majority classes can skew them. To counter this, the study employed a balanced validation set for Hyperparameter Optimization (HPO) and used the macro F1-score, a balanced measure, as the evaluation metric. This research includes models like CodeBERT and GraphCodeBERT and their hierarchical counterparts, hCodeBERT and hGraphCodeBERT, which are trained considering domain knowledge such as the CWE hierarchy. The analysis will use the given metric results to examine the impact of integrating domain knowledge on model performance.

| **Model** | **Accuracy** | **Balanced Accuracy** | **Macro F1-Score** | **Weighted F1-Score** |
|---|---|---|---|---|
| CodeBERT-CE | **26.08** | 4.76 | 1.97 | 10.79 |
| GraphCodeBERT-CE | **26.08** | 4.76 | 1.97 | 10.79 |
| CodeBERT-FL | 15.80 | 13.38 | 11.31 | 18.39 |
| GraphCodeBERT-FL | 18.33 | **16.36** | **13.36** | 20.53 |
| hCodeBERT-default | 20.85 | 11.10 | 11.44 | 20.86 |
| hGraphCodeBERT-default | 18.91 | 8.47 | 9.59 | 20.07 |
| hCodeBERT-equalize | 20.02 | 12.19 | 11.76 | 19.86 |
| hGraphCodeBERT-equalize | 18.37 | 8.50 | 7.51 | 18.05 |
| hCodeBERT-descendants | **25.34** | **15.05** | **13.84** | **23.54** |
| hGraphCodeBERT-descendants | 20.86 | 10.01 | 10.33 | 20.87 |
| hCodeBERT-reachable_leaf_nodes | 22.14 | 12.60 | 12.45 | **23.10** |
| hGraphCodeBERT-reachable_leaf_nodes | 21.22 | 12.14 | 11.64 | 20.84 |

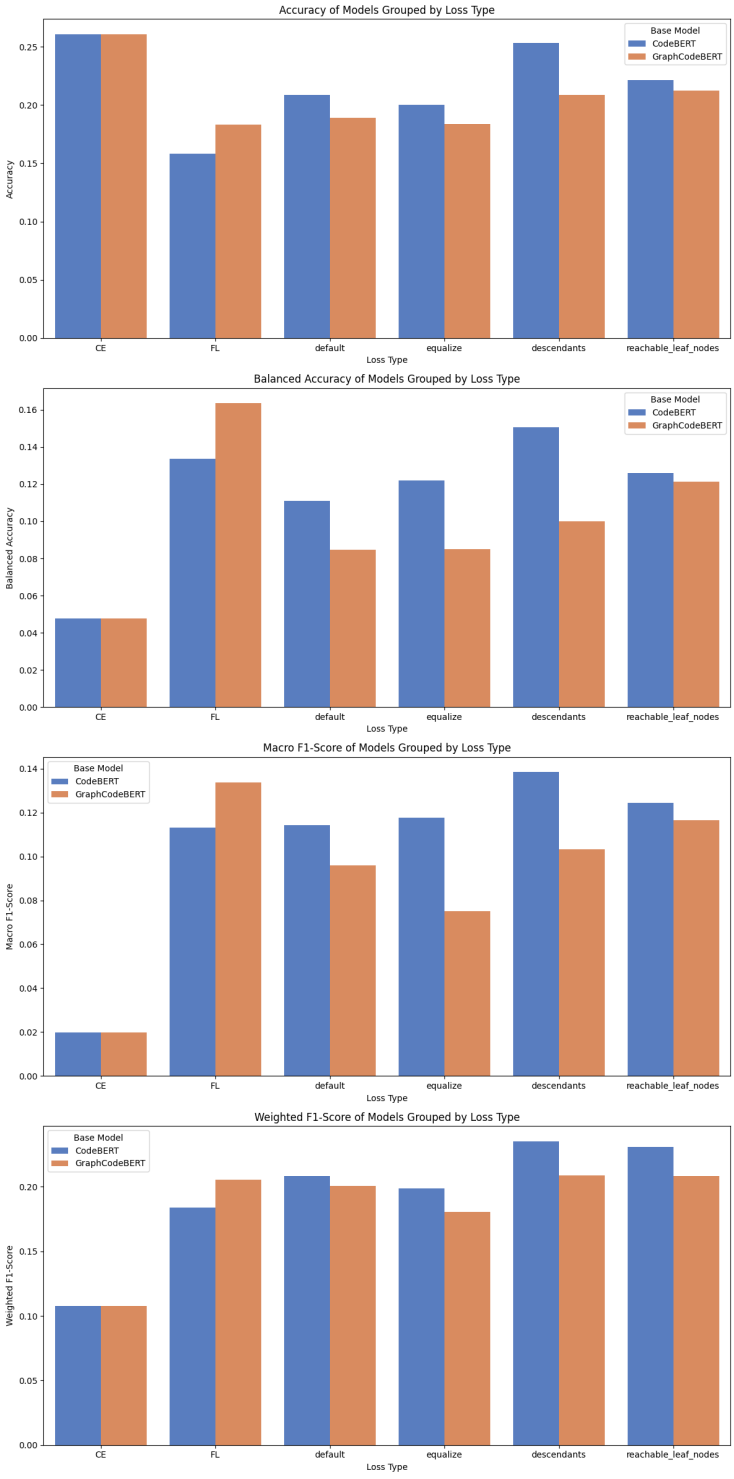



Figure 16: Model Performance Plot for Multiclass Classifiation

# 5 Discussion

In binary classification results, shown in Table 4.5.1 and Figure 15, CodeBERT and GraphCodeBERT models with Cross-entropy Loss show the Lowest Performance: Both models exhibit deficient performance in the CE configuration with zero precision, recall, and F1-score. This indicates that these models cannot effectively classify binary classes, especially positive ones, in this specific setting. Neither model can classify positive classes or negative classes for all data points. In this experiment, only a negative class was predicted.

Both *CodeBERT-FL* and *GraphCodeBERT-FL* perform significantly better than the CE configuration. We use cross-entropy Loss and focal Loss for standard classification. This suggests that the FL configuration is more suitable for binary classification tasks for these models since focal Loss uses weight for each class, which is the inverse class frequency and is effective in class imbalance problems.

The `reachable_leaf_nodes` configuration performs well for both model types, with CodeBERT slightly outperforming GraphCodeBERT. Since the other hierarchical loss weight methods, `default`, `equalize`, and `descendants`, did not show slightly good or even similar performance. Still, since this `reachable_leaf_nodes` loss weight is the closest to the focal loss setting, it might reveal a similar version to a model with focal loss performance. The `reachable_leaf_nodes` method and Focal Loss have in common that both focus on giving weight to actual prediction targets. Focal Loss is not a hierarchical method, so it only provides class weight to an existing class. `reachable_leaf_nodes` start with inverse class frequency for all nodes in the hierarchy, but it gives more weight to reachable target nodes. This indicates the effectiveness of giving class weight to target classes for binary classification.
In multiclass classification results, represented in Table 4.5.2 and Figure 16, the provided numbers of percentage points show that *CodeBERT-CE* and *GraphCodeBERT-CE* have identical performance across all metrics, with an accuracy of 26.08, balanced accuracy of 4.76, macro F1-score of 1.97, and weighted F1-score of 10.79. Compared to the high accuracy, which is the highest among all other models, other balanced accuracy, macro f1-score, and weighted f1-score show the poorest values. These indicate that cross-entropy-based models are completely overfitting, so they only predict a few major classes, which gives high accuracy but poor balanced accuracy; macro f1-score and weighted f1-score, on the other hand—especially in terms of balanced accuracy and macro F1-score, suggesting difficulties in classifying less frequent classes.

This study explores the performance differences in multilabel classification mod-

els trained with non-class-weighted, cross-entropy Loss to capture pure hierarchical classifier impact on classification tasks. The **Cross-Entropy loss** is selected for normal classification to compare the pure effect of leveraging the hierarchical information without weighing the class. As a counterpart, `default` loss weight for hierarchical classification could be compared. As the results show, cross-entropy-based models completely lost the ability to learn. However, the *hCodeBERT-default* and *hGraphCodeBERT-default* models show slightly better and lower than 20% accuracy, which still shows comparable accuracy.

The hierarchical models, *hCodeBERT-default* and *hGraphCodeBERT-default*, showed big improvements compared to *CodeBERT-CE* and *GraphCodeBERT-CE* in all evaluation metrics. The balanced accuracy shows 6.34 and 3.71 percentage points improvement for CodeBERT and GraphCodeBERT, respectively. Moreover, the performance increased for Macro-F1 scores of 9.47 and 7.62 percentage points and for the weighted-f1 scores of 10.07 and 9.28 percentage points. This indicates the model using hierarchical information could classify more class labels than counterpart models in multiclass classification tasks. This indicates integrating the hierarchical information would positively affect the model performance.

Furthermore, it leverages various hierarchical node-based loss weights to capture the influence of hierarchical information on classification tasks. **Focal Loss**, also class weight-based loss, known for its effectiveness in class imbalance problems, is a counterpart for comparison. The research examines the impact of three hierarchical loss weight methods—`equalize`, `descendants`, and `reachable_leaf_nodes`—on model performance, contrasting them with a model performance with focal Loss. The investigation aims to determine how applying different class weights influences model performance in standard and hierarchical multiclass classification tasks.

As shown in Tables 4.5.2 and Figures 16, overall, *hCodeBERT-descendants* performs best for all evaluation metrics, except for cross-entropy-based models, which do not have exact values that can be considered in practice. The highest accuracy is 25.34%, which is an overwhelmingly high value compared to other models, and this is not only accuracy, but all other metrics, macro f1-score and weighted f1-score, except that balanced accuracy is low by about 1.31%. Shows the highest value. When compared to the counterpart focal loss-based models, *CodeBERT-FL* and *GraphCodeBERT-FL*, *hCodeBERT-descendants* shows better overall values. Next, *hCodeBERT-reachable_leaf_nodes* performed well in Accuracy and Weighted-F1 score. Suppose we were to compare equalized loss weight and Focal Loss, which use the same inverse class frequency as weight per class because they have something

in common. In that case, they are surprisingly equalize-based models, which used all layer information and gave weights in the same way as focal Loss for all layers; macro f1-score of both *hCodeBERT-equalize* and *hGraphCodeBERT-equalize* were almost similar or dropped to 0.45 and -6.15 percentage points, respectively, based on the FL-model counterpart. Model performance may vary depending on the model, hyperparameters, learning environment, etc. Still, even if there is a common feature of using the inverse class frequency as a class weight if the class weight is set on all nodes, it is relatively complex to predict the target class that should be expected. Instead, the weight value may be distributed to other nodes, affecting or decreasing its weight. To calculate the class weight used in hierarchical classification, the occurrence of each class counts how many times each node in a graph is "affected" (either as a target node, an ancestor of a target node, or a successor of an ancestor of a target node). Therefore, unlike Focal Loss, which directly obtains class weights only from classes predicted from all target labels, this hierarchical method may perform poorly because the hierarchy structure greatly affects the class weight value.

All three loss weights of `equalize`, `descendants`, and `reachable_leaf_nodes` depend on which node the model focuses on based on the inversed class frequency calculated by Equalize. The class weight varies depending on the class. Equalize uses the inversed class frequency as is, Descendants adds weight to the class corresponding to the node's descendants, and `reachable_leaf_nodes` adds weight to predictable class nodes reachable from the node. Therefore, unlike `equalize`, `descendants`, and `reachable_leaf_nodes` are loss weights that focus more on predictable classes. Therefore, looking at the evaluation metric value, `equalize` has balanced accuracy and balanced measure more than `descendants`, and `reachable_leaf_nodes` because its purpose is to balance all classes equally. The Macro F1-score is higher, and on the contrary, overall, the more actual predictable targets are hit, the higher the accuracy and the lower the weighted f1-score. `descendants`, and `reachable_leaf_nodes` have something in common: more weight is given to predictable classes, so generally, similar results can be seen in all metrics. Depending on the complexity of the hierarchy, the deviation between the two may vary.

The results demonstrate that embedding domain knowledge like the CWE hierarchy significantly bolsters the models' capability to identify and classify vulnerabilities. Hierarchical models, especially those utilizing Focal Loss to address class imbalances, show marked improvements in binary classification tasks. This underscores the importance of selecting appropriate loss functions and weighting schemes for effective class imbalance management. In terms of the methodology, the study employed various loss weight strategies to compare the effectiveness of different classification

models.

In multiclass classification scenarios, hierarchical models, particularly those employing `descendants` and `reachable_leaf_nodes` loss weighting schemes, outperform non-hierarchical models, showcasing their efficiency in handling the complexities of the CWE hierarchy. The findings reveal that hierarchical models with specialized loss weights generally surpass the performance of the standard cross-entropy (CE) method.

A thorough analysis indicates that high-performing models share the strategy of assigning variable weights to classes, concentrating on predictable targets. This approach correlates higher weight allocation to actual labels with enhanced performance metrics. The `equalize` strategy, by distributing weights uniformly across hierarchy levels, tends to yield more balanced outcomes than other weighting schemes. Both `descendants` and `reachable_leaf_nodes` show similar results in metrics like hierarchy depth and class distribution, varying their effectiveness based on the specific hierarchy structure.

It is important to contextualize the performance results of this study. Despite having lower performance metrics than related studies, our research dealt with higher task complexity. It utilized real-world data from GitHub repositories, as opposed to synthetic data. The data's fine-grained hierarchical nature adds to this complexity, making direct comparisons with other studies, which often use less detailed data, inappropriate. For example, as described in Chakraborty et al. (2020), the Zou et al. (2019) study, using the less complex synthetic dataset, Multiclass Vulnerability Dataset (MVD) for multiclass classification, showed poor performance on real-world datasets. Although it was re-trained with a real-world dataset(ReVEAL(Chakraborty et al., 2020)) and tested on binary classification, but it showed 15.7% in weighted f1-score. Furthermore, our study's models, though not trained on binary classification, demonstrate significant performance (ranging from 79.09% to 84.1% in weighted f1-score) on even more complex real-world datasets. This indicates that despite appearing modest, our results are realistic and reflect the study's intricate nature and the challenging data it handles.
Lastly, this study has the limitation that learning quickly overfits. The negative class with the largest data size is 59,109, and the number of data point of the smallest positive class is 1. Overall, because of this class imbalance with long tail distribution, despite the reassignment, the cutoff was 1000, so the class imbalance was alleviated, but it still exists. To make the most of hierarchical information, we did not use a larger cutoff, but if we do use it, there is a trade-off in class imbalance. Considering

an environment where there is absolutely as much vulnerability-free code as possible in a program, it is difficult to avoid serious class imbalance. Even when going for multi-label classification instead of binary, there is still an absolutely small amount of data for vulnerabilities that are not easily discovered in the code. In addition, when learning with a synthetic dataset to secure minor vulnerability data, serious performance degradation occurs on the real word dataset. If a method to more effectively resolve class imbalance can be applied by considering these factors, further improved performance can be expected in the future. Additionally, if there is a way to assign loss weight to the target label more effectively by adding weight factors, etc., as variables, performance may also be affected.

# 6 Conclusion

This research focuses on improving vulnerability detection by integrating domain-specific knowledge into transformer-based models such as CodeBERT and GraphCodeBERT. It contributes to understanding how this domain knowledge integration can enhance the models' ability to detect vulnerabilities in code. A distinctive feature of this study was the incorporation of the Common Weakness Enumeration (CWE) hierarchy into these models, enhancing their ability to classify vulnerabilities accurately. The study initially faced a significant challenge due to the severe class imbalance in the combined Big-Vul and CVEfixes datasets, having fine-grained CWE hierarchy. To address this issue, we approached reassigning CWE IDs, adding artificial roots, and using Loss weight-based Loss and HPO with a balanced validation set and evaluation measures, such as macro f1-score and balanced accuracy. This approach differed from traditional evaluation metrics to better compare the performance of hierarchical methods. In the HPO process with the Optuna, on-the-fly tokenization, adaptive learning rate strategy, gradient accumulation, and parallelism proved essential in overcoming computational limitations and enhancing the training process. We compared standard classification with Focal and CE Loss to hierarchical classification using a deep classifier with varied loss weights in experiments. The results demonstrated that hierarchical models outperformed non-hierarchical models, especially those utilizing the `descendants` and `reachable_leaf_nodes` loss weighting schemes. More in detail about the results, the comprehensive experiments have highlighted the limitations of standard cross-entropy Loss, particularly under class imbalance conditions, and underscored the efficacy of hierarchical models with specialized loss weights, such as Focal Loss and hierarchical loss weight methods. Hierarchical models employing the, `default`, `equalize`, did not perform as well as `default`, `equalize` or Focal Loss models; however, `descendants` and `reachable_leaf_nodes` loss weighting schemes have shown better performance, aligning with the study's objective to harness the full potential of hierarchical data structures. The findings suggest that the weight on predictable target nodes in the loss function plays a pivotal role in model performance. This underlines the effectiveness of our approach in managing the complexities of the CWE hierarchy and the challenges posed by class imbalance. In conclusion, this study addresses the technical challenges of class imbalance and data complexity. The detailed results are in Section 4. It contributes significantly to understanding how domain-specific hierarchical information can enhance the performance of machine learning models in the context of code vulnerability detection.

# A Appendix

## A.1 CWE Hierarchy

### A.1.1 CWE Hierarchy before CWE Reassignment

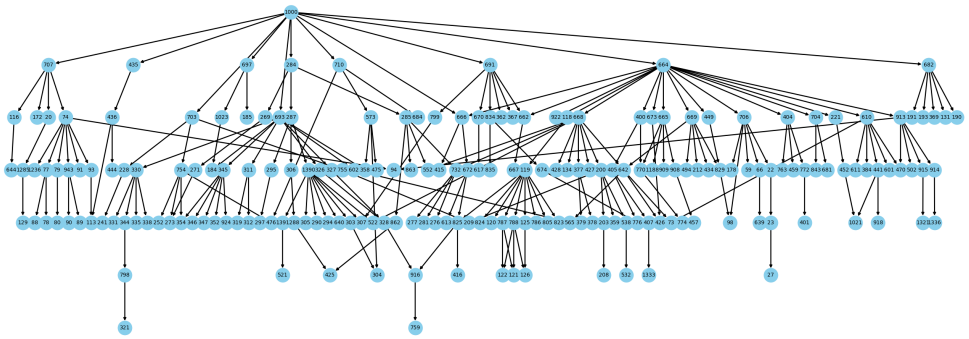


Figure 17: CWE Hierarchy before CWE Reassignment

### A.1.2 CWE ID Distribution in After CWE Reassignment in Dataset

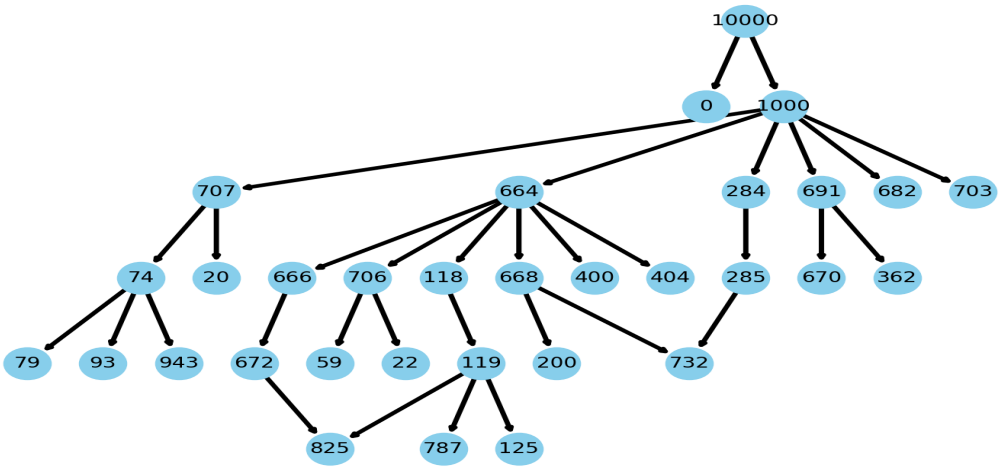



Figure 18: CWE Hierarchy after CWE Reassignment

## A.2 Hyperparameter Optimization (HPO)

### A.2.1 Hyperparameter Optimization History Plot for CodeBERT

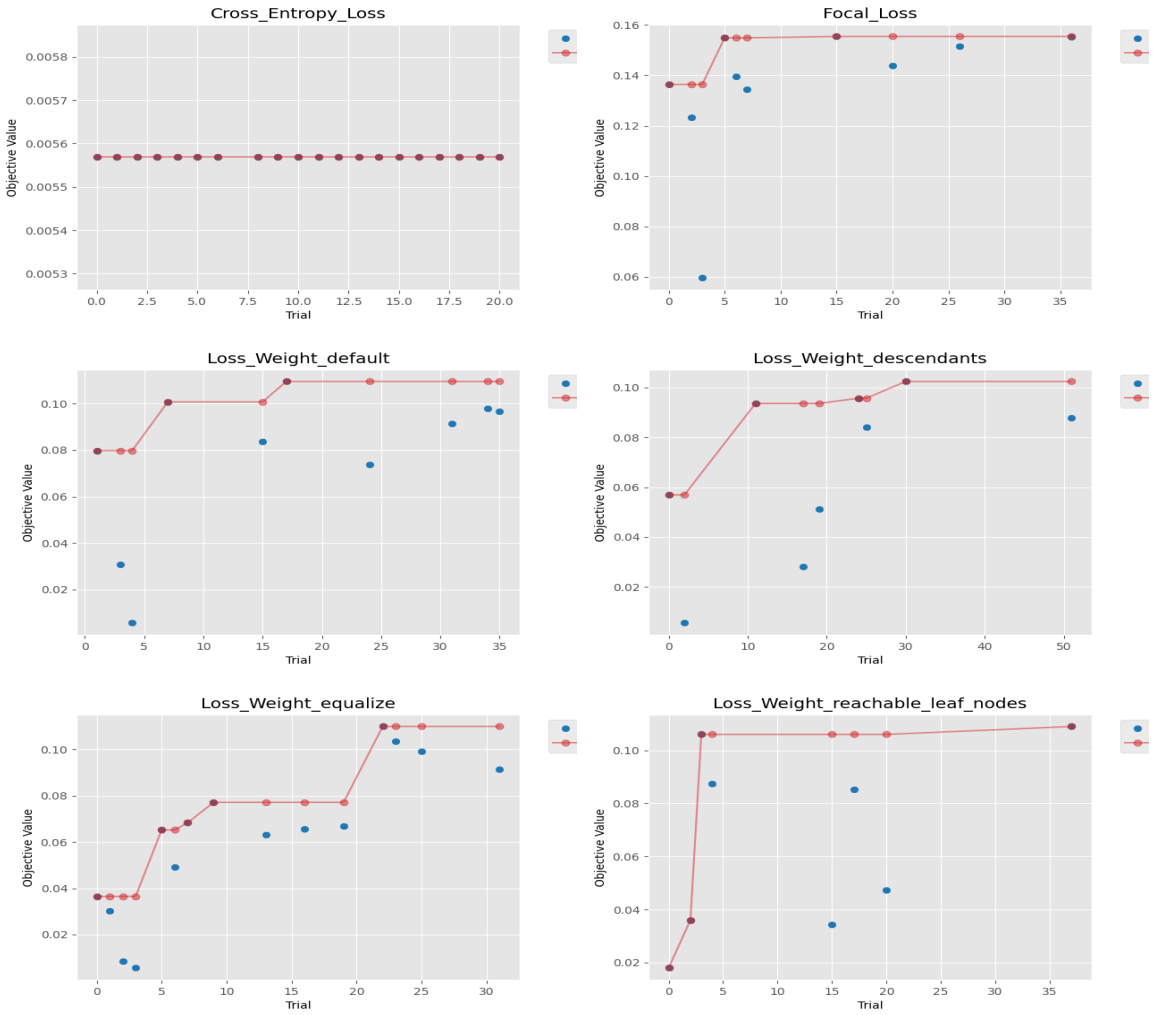



Figure 19: Hyperparameter Optimization History Plot for CodeBERT (Objective Value: Macro F1-Score)

### A.2.2 Hyperparameter Optimization History Plot for GraphCodeBERT

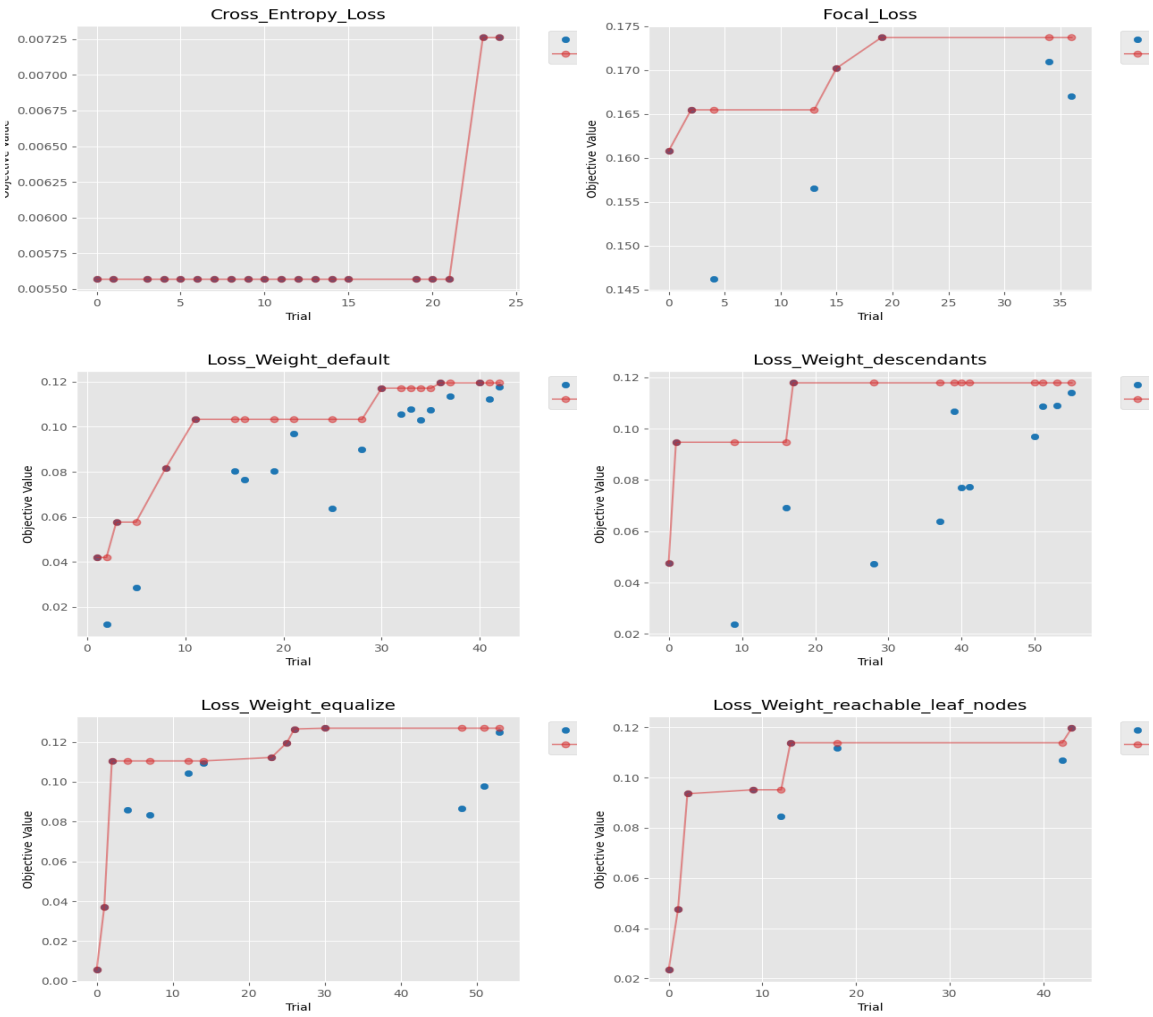



Figure 20: Hyperparameter Optimization History Plot for GraphCodeBERT (Objective Value: Macro F1-Score)

### A.2.3 HPO Slice Plot for CodeBERT

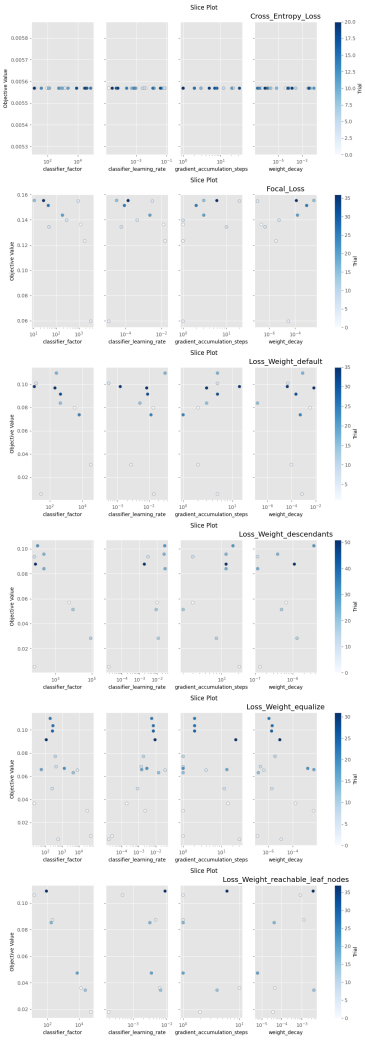



Figure 21: HPO Slice Plot for CodeBERT (Objective Value: Macro F1-Score)

### A.2.4 HPO Slice Plot for GraphCodeBERT

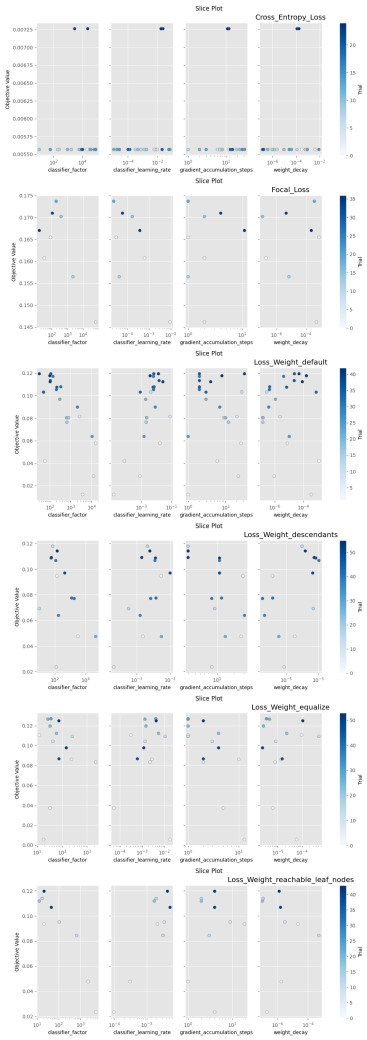



Figure 22: HPO Slice Plot for GraphCodeBERT (Objective Value: Macro F1-Score)

### A.2.5 Fine-Tuning Results of CodeBERT-CE

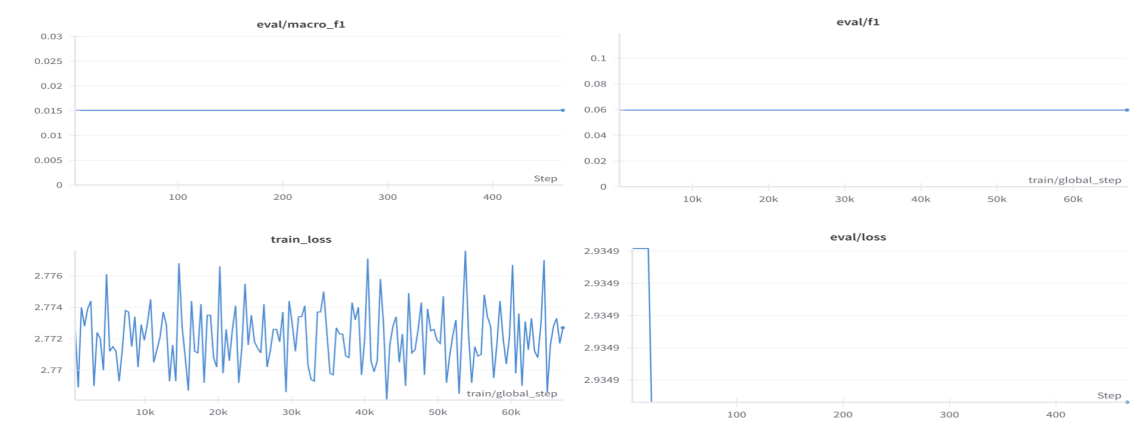



Figure 23: HPO Slice Plot for CodeBERT-CE (Objective Value: Macro F1-Score)

### A.2.6 Fine-Tuning Results of GraphCodeBERT-CE

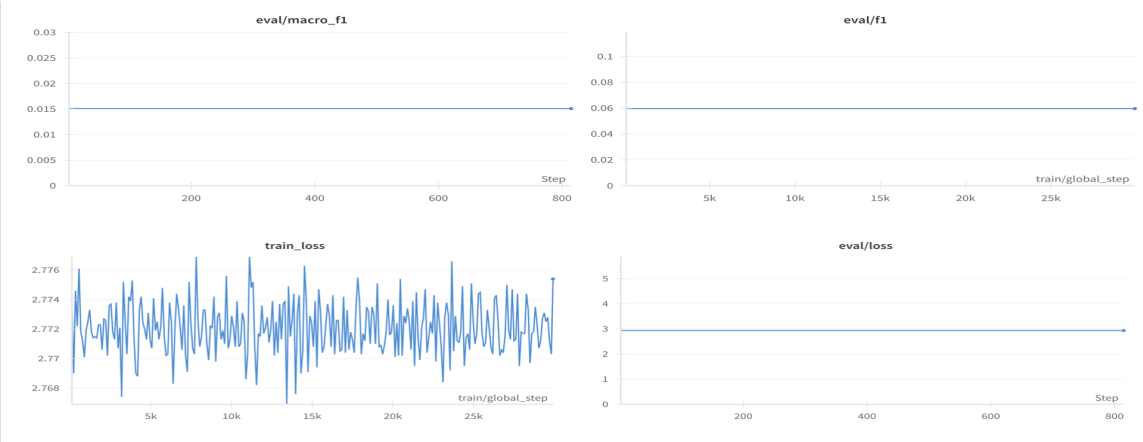



Figure 24: HPO Slice Plot for GraphCodeBERT-CE (Objective Value: Macro F1-Score)

### A.2.7 Fine-Tuning Results of GraphCodeBERT-FL

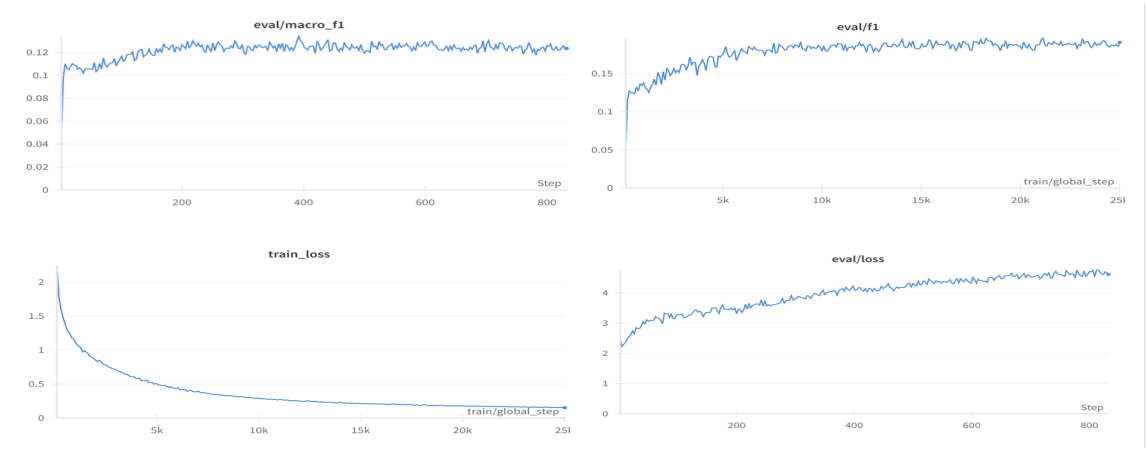



Figure 25: HPO Slice Plot for GraphCodeBERT-FL (Objective Value: Macro F1-Score)

### A.2.8 Fine-Tuning Results of hCodeBERT-`descendants`

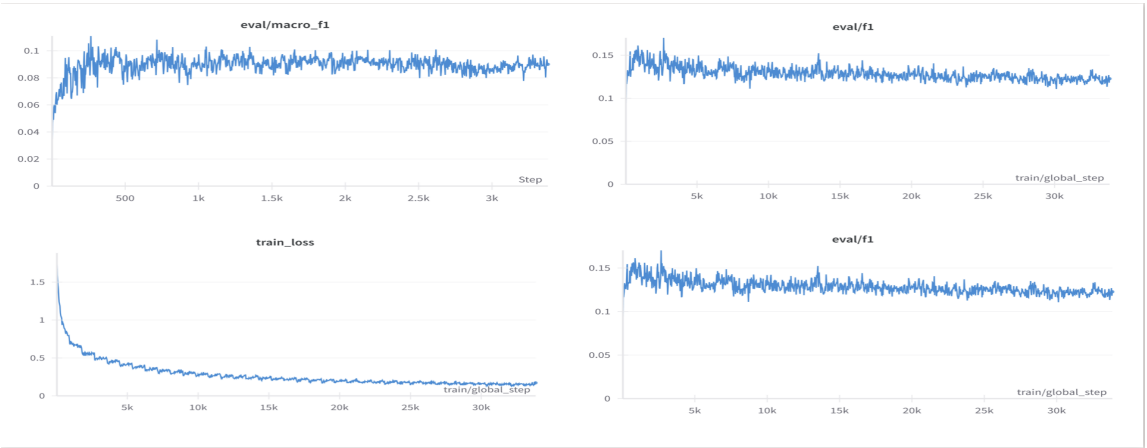



Figure 26: HPO Slice Plot for hCodeBert (Objective Value: Macro F1-Score)

### A.2.9 Best Parameters in HPO

| **Model** | **Classifier Learning Rate** | **Classifier Factor** | **Gradient Accumulation Steps** | **Weight Decay** |
|---|---|---|---|---|
| CodeBERT-CE | 0.00179 | 34.94 | 3 | 6.45e-06 |
| GraphCodeBERT-CE | 0.02353 | 2882.76 | 12 | 0.00012 |
| CodeBERT-FL | 0.00155 | 2280.82 | 11 | 2.22e-05 |
| **GraphCodeBERT-FL** | **0.00021** | **336.09** | **14** | **0.00125** |
| hCodeBERT-default | 0.02200 | 266.17 | 15 | 0.00363 |
| hGraphCodeBERT-default | 0.04375 | 178.04 | 9 | 0.00401 |
| **hCodeBERT-descendants** | **0.00438** | **54.76** | **5** | **8.28e-07** |
| hGraphCodeBERT-descendants | 0.00241 | 18.73 | 8 | 9.98e-06 |
| hCodeBERT-equalize | 0.00555 | 97.60 | 6 | 2.52e-06 |
| hGraphCodeBERT-equalize | 0.01469 | 98.33 | 5 | 0.00307 |
| hCodeBERT-reachable_leaf_nodes | 0.00072 | 10.81 | 6 | 0.00436 |
| hGraphCodeBERT-reachable_leaf_nodes | 0.00192 | 10.52 | 6 | 5.23e-07 |

Table 3: Best Hyperparameters in HPO (Two Best Models: GraphCodeBERT-FL and hCodeBERT-descendants)

# References

Aptori (2023). A comprehensive guide to application security testing methods, `https://aptori.dev/blog/a-comprehensive-guide-to-application-security-testing-methods`. ALast Updated June 18, 2023.

Bhandari, G., Naseer, A. and Moonen, L. (2021). CVEfixes: Automated Collection of Vulnerabilities and Their Fixes from Open-Source Software, *Proceedings of the 17th International Conference on Predictive Models and Data Analytics in Software Engineering (PROMISE '21)*, ACM, p. 10.

Brust, C.-A. and Denzler, J. (2020). Integrating domain knowledge: Using hierarchies to improve deep classifiers, *in* S. Palaiahnakote, G. Sanniti di Baja, L. Wang and W. Q. Yan (eds), *Pattern Recognition*, Springer International Publishing, Cham, pp. 3–16.

Chakraborty, S., Krishna, R., Ding, Y. and Ray, B. (2020). Deep learning based vulnerability detection: Are we there yet?

Comparitech (2023). 25+ cyber security vulnerability statistics and facts of 2023, `https://www.comparitech.com/blog/information-security/cybersecurity-vulnerability-statistics/`. Last Updated June 14, 2023.

Contreras, C., Dokic, H., Huang, Z., Stan Raicu, D., Furst, J. and Tchoua, R. (2023). Multiclass classification of software vulnerabilities with deep learning, *Proceedings of the 2023 15th International Conference on Machine Learning and Computing*, ICMLC '23, Association for Computing Machinery, New York, NY, USA, p. 134–140.
**URL:** *https://doi.org/10.1145/3587716.3587738*

Devlin, J., Chang, M.-W., Lee, K. and Toutanova, K. (2019). Bert: Pre-training of deep bidirectional transformers for language understanding.

Fan, J., Li, Y., Wang, S. and Nguyen, T. N. (2020). A c/c++ code vulnerability dataset with code changes and cve summaries, *Proceedings of the 17th International Conference on Mining Software Repositories*, MSR '20, Association for Computing Machinery, New York, NY, USA, p. 508–512.
**URL:** *https://doi.org/10.1145/3379597.3387501*

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

Fu, M., Nguyen, V., Tantithamthavorn, C. K., Le, T. and Phung, D. (2023). Vulexplainer: A transformer-based hierarchical distillation for explaining vulnerability types, *IEEE Transactions on Software Engineering* **49**(10): 4550–4565.

Fu, M. and Tantithamthavorn, C. (2022). Linevul: A transformer-based line-level vulnerability prediction, *Proceedings of the 19th International Conference on Mining Software Repositories*, MSR '22, Association for Computing Machinery, New York, NY, USA, p. 608–620.
**URL:** *https://doi.org/10.1145/3524842.3528452*

Fu, M., Tantithamthavorn, C., Le, T., Nguyen, V. and Phung, D. (2022). Vulrepair: A t5-based automated software vulnerability repair, *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2022, Association for Computing Machinery, New York, NY, USA, p. 935–947.
**URL:** *https://doi.org/10.1145/3540250.3549098*

Guo, D., Ren, S., Lu, S., Feng, Z., Tang, D., Liu, S., Zhou, L., Duan, N., Svyatkovskiy, A., Fu, S., Tufano, M., Deng, S. K., Clement, C., Drain, D., Sundaresan, N., Yin, J., Jiang, D. and Zhou, M. (2021). Graphcodebert: Pre-training code representations with data flow.

Hanif, H. and Maffeis, S. (2022). Vulberta: Simplified source code pre-training for vulnerability detection, pp. 1–8.

Hochreiter, S. and Schmidhuber, J. (1997). Long Short-Term Memory, *Neural Computation* **9**(8): 1735–1780.
**URL:** *https://doi.org/10.1162/neco.1997.9.8.1735*

*Introduction to CVE* (n.d.). Online PDF accessed via Chrome Extension. [Online; accessed 29-November-2023].
**URL:** *chrome-extension://efaidnbmnnnibpcajpcglclefindmkaj/https://cve.mitre.org/docs/cve-intro-handout.pdf*

Jelinek, F. (1992). Statistical methods for speech recognition.

Jurafsky, D. and Martin, J. H. (2023). *Speech and Language Processing*, 3rd edition draft edn. Chapter3 : N-gram Language Models, Drafted of January 7, 2023. **URL:** *PDF link (if available online)*

Li, Z., Zou, D., Xu, S., Ou, X., Jin, H., Wang, S., Deng, Z. and Zhong, Y. (2018). Vuldeepecker: A deep learning-based system for vulnerability detection, *Proceedings 2018 Network and Distributed System Security Symposium*, Internet Society. **URL:** *http://dx.doi.org/10.14722/ndss.2018.23158*

Liu, Y., Ott, M., Goyal, N., Du, J., Joshi, M., Chen, D., Levy, O., Lewis, M., Zettlemoyer, L. and Stoyanov, V. (2019). Roberta: A robustly optimized bert pretraining approach.

Mikolov, T., Chen, K., Corrado, G. and Dean, J. (2013). Efficient estimation of word representations in vector space.

MITRE (2023). About common weakness enumeration. Accessed: 2023-11-29. **URL:** *https://cwe.mitre.org/about/index.html*

Russell, R. L., Kim, L., Hamilton, L. H., Lazovich, T., Harer, J. A., Ozdemir, O., Ellingwood, P. M. and McConley, M. W. (2018). Automated vulnerability detection in source code using deep representation learning.

Schmitt, J. (2023). Sast vs dast: What they are and when to use them, `https://circleci.com/blog/sast-vs-dast-when-to-use-them/`. Last Updated May 10, 2023.

Turing, A. (1950). Computing machinery and intelligence.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L. and Polosukhin, I. (2023). Attention is all you need.

Weizenbaum, J. (1966). Eliza - a computer program for the study of natural language communication between man and machine.

Wu, F., Wang, J., Liu, J. and Wang, W. (2017). Vulnerability detection with deep learning, *2017 3rd IEEE International Conference on Computer and Communications (ICCC)*, pp. 1298–1302.

Zhou, Y., Liu, S., Siow, J., Du, X. and Liu, Y. (2019). Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks.

Ziems, N. and Wu, S. (2021). Security vulnerability detection using deep learning natural language processing.

Zou, D., Wang, S., Xu, S., Li, Z. and Jin, H. (2019). vuldeepecker: A deep learning-based system for multiclass vulnerability detection, *IEEE Transactions on Dependable and Secure Computing* p. 1–1.
**URL:** *http://dx.doi.org/10.1109/TDSC.2019.2942930*

# Declaration of authorship

I hereby declare that the report submitted is my own unaided work. All direct or indirect sources used are acknowledged as references. I am aware that the Thesis in digital form can be examined for the use of unauthorized aid and in order to determine whether the report as a whole or parts incorporated in it may be deemed as plagiarism. For the comparison of my work with existing sources I agree that it shall be entered in a database where it shall also remain after examination, to enable comparison with future Theses submitted. Further rights of reproduction and usage, however, are not granted here. This paper was not previously presented to another examination board and has not been published.

Location, date

Name