FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

A Transformer-based GitHub Action for Vulnerability Detection

André Filipe Meireles do Nascimento

U. PORTO

FEUP FACULDADE DE ENGENHARIA UNIVERSIDADE DO PORTO

Master in Informatics and Computing Engineering

Supervisor: Professor Rui Maranhão

Co-Supervisor: Eduard Pinconschi, MSc

July 24, 2023

A Transformer-based GitHub Action for Vulnerability Detection

André Filipe Meireles do Nascimento

Master in Informatics and Computing Engineering

July 24, 2023

Abstract

Everything that humanity creates has its vulnerabilities, and software is no exception. As technology progresses, software usage and complexity increase, and so do software vulnerabilities (and the damage they cause). The software industry is aware of this problem, and concerns regarding detecting software vulnerabilities play a fundamental role since the start of projects' development to reduce the risks of attacks.

Nonetheless, due to the nature of this issue, it has yet to be solved. Companies implement several security measures in the Software Development Life Cycle (SDLC), such as security audits and static/dynamic code analysis. However, these are usually performed by security experts and only later in the SDLC when the software has already been developed. This approach is not practical since it is not possible to detect and fix vulnerabilities as soon as they are created, and only people with security expertise can perform those tasks.

Deep Learning (DL) has interesting applications for vulnerability detection, and exciting progress regarding this topic has been made. Transformers, a DL model type, can be used to detect vulnerabilities in high-level programming languages, such as Java. Code vulnerability scanners based on DL remove the need for expertise in the security area and allow developers to integrate these kinds of tools into the early stages of the SDLC.

Most software is hosted in online repositories, as developers and companies need better ways to store and manage their source code. GitHub, the world's largest open-source community, provides a platform for developers to share their code, which inherently leads to being home to various vulnerabilities.

In this thesis, we implemented a novel tool that leverages Deep Learning models to detect vulnerabilities in Java source code. The tool, J-TAS, implemented as a GitHub Action, is designed to be used early in the SDLC, marking a significant shift-left in vulnerability detection. This approach allows developers to identify and rectify vulnerabilities as soon as they emerge, and reduces the reliance on specialized security expertise.

However, our study found that the synthetic nature of the datasets used limited the tool's efficacy, leading to suboptimal results. Despite this, the work developed is marked as a proof-of-concept, standing as a testament to the innovation of Deep Learning in the DevOps scenario.

Keywords: Vulnerability Detection, Software Security, Transformer, Java

ACM Classification:
• Security and privacy → Software and application security → Software security engineering
• Computing methodologies → Artificial intelligence → Natural language processing

Resumo

Tudo o que a humanidade cria tem as suas vulnerabilidades, e o software não é excepção. À medida que a tecnologia avança, a utilização e a complexidade do software aumentam, assim como as vulnerabilidades do software (e os danos que causam). A indústria de software está ciente deste problema, e as preocupações relativas à deteção de vulnerabilidades de software desempenham um papel fundamental desde o início do desenvolvimento de projectos para reduzir os riscos de ataques.

No entanto, devido à natureza deste problema, ainda há muito a fazer para o resolver. As empresas implementam várias medidas de segurança no Ciclo de Vida do Desenvolvimento de Software (SDLC), tais como auditorias de segurança e análise de código estática/dinâmica. Contudo, estas são geralmente realizadas por peritos em segurança e só mais tarde no SDLC quando o software já tiver sido desenvolvido. Esta abordagem não é prática, uma vez que impossibilita detetar e corrigir vulnerabilidades assim que estas são criadas, e apenas pessoas com conhecimentos de segurança podem executar essas tarefas.

O *Deep Learning* (DL) tem aplicações interessantes para a deteção de vulnerabilidades, e foram feitos progressos entusiasmantes em relação a este tópico. *Transformer*, um tipo de modelo DL, pode ser utilizados para detetar vulnerabilidades em linguagens de programação de alto nível, tais como Java. Os scanners de vulnerabilidade de código baseados em DL eliminam a necessidade de perícia na área da segurança e permitem aos programadores integrar este tipo de ferramentas nas fases iniciais do SDLC.

A maioria do software está alojada em repositórios online, uma vez que os programadores e as empresas precisam de melhores formas de armazenar e gerir o seu código fonte. *GitHub*, a maior comunidade de código aberto do mundo, fornece uma plataforma para os programadores partilharem o seu código, o que conduz intrinsecamente a um aglomerado de várias vulnerabilidades.

Nesta tese, implementámos uma nova ferramenta que utiliza modelos de aprendizagem profunda para detetar vulnerabilidades no código-fonte Java. A ferramenta, J-TAS, implementada como uma GitHub Action, foi concebida para ser utilizada no início do SDLC, marcando uma mudança significativa na deteção de vulnerabilidades. Esta abordagem permite que os programadores identifiquem e rectifiquem as vulnerabilidades assim que estas surgem, e reduz a dependência de conhecimentos especializados em segurança.

No entanto, o nosso estudo constatou que a natureza sintética dos dados utilizados limitou a eficácia da ferramenta, levando a resultados abaixo do ideal. Apesar disso, o trabalho desenvolvido é marcado como uma prova de conceito, sendo um testemunho da inovação do Deep Learning no cenário de DevOps.

Palavras-chave: Deteção de vulnerabilidades, Segurança de software, Transformer

Classificação ACM:
• **Security and privacy → Software and application security → Software security engineering**
• Computing methodologies → Artificial intelligence → Natural language processing

Acknowledgements

First, I want to thank my supervisors, Professor Rui and Eduard, whose guidance and continuous support have been invaluable throughout this challenging journey. I am very grateful for all the time and effort that they have put into this project.

To my family, especially my parents, I owe an immeasurable debt of gratitude for all the love, support and encouragement they have given me throughout my life. All the goals that I have achieved, as well as those on the horizon, are thanks to them.

My heartfelt appreciation extends to my circle of friends for all the knowledge and experiences we shared. Thank you for all of the laughs and good times. A special thanks to Teté, who has been with me ever since I can remember; growing up together and sharing all the good and bad moments has been a pleasure.

Last but certainly not least, I want to thank my beloved Mariana for all the care and, especially, for all her patience with me. She has been my inspiration and motivation to keep going and be the best version of myself. I feel incredibly fortunate to have her by my side.

André

"Conseguimos!
Conseguimos, Portugal, Lisboa!
Esperávamos, desejávamos, conseguimos!
Vitória!"

Marcelo Rebelo de Sousa

Contents

List of Figures

List of Tables

Acronyms

AST	Abstract Syntax Tree
BCE	Binary Cross-Entropy
BERT	Bidirectional Encoder Representations from Transformers
BFP	Bug-Fixing Pair
BLSTM	Bidirectional Long Short Term Memory
CDG	Control Dependence Graph
CE	Cross-Entropy
CFG	Control Flow Graph
CPG	Code Property Graph
CV	Computer Vision
CVE	Common Vulnerabilities and Exposures
CVSS	Common Vulnerability Scoring System
CWE	Common Weakness Enumeration
DDG	Data Dependence Graph
DL	Deep Learning
FN	False Negative
FNR	False Negative Rate
FP	False Positive
FPR	False Positive Rate
FR	Functional Requirement
HF	Hugging Face
J-TAS	Java Transformer-based Automated Scanner
JSON	JavaScript Object Notation
LR	Learning Rate
LSR	Long Sequence Removal
LSTM	Long Short Term Memory
MLM	Masked Language Modelling
NFR	Non-Functional Requirement
NIST	National Institute of Standards and Technology
NLLLoss	Negative Log Likelihood Loss
NLP	Natural Language Processing
NSDR	Non Significant Data Removal
NSDR	Non-Significant Data Removal
NVD	National Vulnerability Database
PDG	Program Dependence Graph
RNN	Recurrent Neural Networks
RQ	Research Question
SARD	Software Assurance Reference Dataset
SARIF	Static Analysis Results Interchange Format
SDLC	Software Development Life Cycle
TN	True Negative
TP	True Positive
VD	Vulnerability Detection

Chapter 1

Introduction

1.1 Context

As technology advances, the use of software has become increasingly prevalent in every aspect of our lives. However, as software usage increases, so do the number of vulnerabilities. Software vulnerabilities are weaknesses in a piece of software that attackers can exploit to compromise the system [24]. These vulnerabilities can come in the form of security bugs or design flaws and can lead to a wide range of consequences, such as data breaches, identity theft, financial loss, and even the loss of life in the case of critical systems.

In order to mitigate the risks associated with software vulnerabilities, companies implement various security measures throughout the Software Development Life Cycle (SDLC). Nonetheless, most of these measures, such as security audits and static code analysis, are performed by security experts and only later in the SDLC when the software has already been developed. This way, developers, who are responsible for introducing vulnerabilities while creating software, are not heavily involved in the security process [72].

The shift-left principle refers to the idea of incorporating security into the early stages of the SDLC rather than waiting until the end of the process to perform testing [21]. Integrating security into the development process is essential since the earlier security is addressed, the less effort and cost will be required [67]. Adopting the shift-left principle can be challenging, as it requires a change in the way software is developed. Organizations may struggle to incorporate security into the development process, as it requires a shift in culture and the integration of new processes and tools.

Deep Learning (DL) has interesting applications for vulnerability detection, and research in this area has seen exciting progress. DL-based tools smooth security integration into the software development process, as they do not require security expertise to configure conventional techniques. Additionally, they do not suffer from high false-positive/false-negative rates [6]. Current state-of-the-art (SOTA) DL-based models have achieved a 90% F1-score, with some reporting as high as 99% accuracy in detecting vulnerabilities in source code, outperforming static analysers [69]. Despite these outstanding results, some problems still need to be addressed, such as a lack of real-world scenarios datasets, data imbalance, the semantic dependency between code, and lack of support for multiple programming languages [6, 60].

1.2 Motivation

Regardless of the progress made towards vulnerability detection, the number of software vulnerabilities reported is still rising at an alarming rate [5]. Thus it is essential to continue the research on this topic and to explore new approaches to improve the current state-of-the-art. The transformer model is a novel DL architecture, introduced in 2017 [75], that has been used in various Natural Language Processing (NLP) tasks, such as machine translation, text summarisation, and question answering. The architecture has rapidly become the dominant architecture for NLP, outperforming alternative neural models in tasks related to natural language understanding and natural language generation [77]. As for vulnerability detection, since common vulnerabilities often have defined structures and patterns [88], as do natural language texts, it is trivial to think of the use of transformers for this task.

GitHub's popularity as a software repository is undeniable. It is home to millions of projects and the most prominent open-source community in the world. Nonetheless, its popularity leads to many vulnerabilities created by open-source developers being hosted and shared within the community. This calls for the development of tools that can detect vulnerabilities in GitHub's repositories, which can be provided on GitHub's marketplace as GitHub Actions. GitHub Actions are small pieces of code that can be integrated into the development environment of the millions of GitHub users, thus enabling the integration of security into the development process.

Currently, GitHub has some vulnerability detection analysis already integrated into open-source repositories by default, and some Actions are available to further improve the security and quality of the code. However, there are no Actions based on the Transformer architecture. Taking into account the work carried out by Mamede *et al.*[46] for a proof-of-concept for a Transformer-based IDE extension for vulnerability detection, we have the opportunity to develop the first Transformer-based GitHub Action for vulnerability detection.

1.3 Objectives

Considering the promising advances in DL vulnerability detection, alongside the work carried out previously, and the popularity of GitHub as a software repository, we propose:

The first Transformer-based GitHub Action for vulnerability detection.

The tool developed must be able to identify potential vulnerabilities in Java files and provide meaningful feedback to developers, promoting a safe code development environment, with the strength of the Transformer model.

The creation of a curated vulnerability detection dataset, which can be used to train and evaluate the developed model.

Data quantity and quality are essential for developing a robust model, and the dataset must be representative of real-world scenarios to ensure the model's generalization.

To deepen current research on the use of Transformers for vulnerability detection.

Since it is a novel architecture and few studies have been conducted on this topic, we intend to contribute to the research on this problem by exploring different approaches and architectures, comparing them with state-of-the-art results.

1.4 Document Structure

The rest of this document is structured as follows:

- Chapter 2 provides the background of vulnerability detection, including key concepts, techniques and models used for this task.
- Chapter 3 conducts a thorough examination of the current state-of-the-art in Deep Learning-based vulnerability detection. This chapter analyses the most relevant works in the field, exploring their methodologies, techniques, and presenting the results achieved.
- Chapter 4 formalises the problem statement and outlines this dissertation's proposed solution. In this chapter, the specific objectives and work plan of the research are defined, providing a clear and concise overview of the proposed approach.
- Chapter 5 outlines the detailed methodology employed to implement the proposed solution. This chapter elaborates on the dataset utilized, the models developed, and the integration of the solution into a GitHub Action.
- Chapter 6 presents the results obtained from the conducted experiments. This chapter meticulously showcases and examines the outcomes, providing critical insights and analysis of the experimental findings.
- Chapter 7 summarizes the work conducted in this dissertation, highlighting our main contributions. Additionally, this chapter proposes future directions for further research and improvements to the proposed solution.

Chapter 2

Background

This chapter provides a comprehensive overview of the background information related to vulnerability detection. Section 2.1 discusses the current state of software vulnerabilities, their underlying causes, and methodologies for classifying and prioritizing. Section 2.2 focuses on SDLC models and security practices that organizations adopt, emphasizing the importance of addressing security issues early in the SDLC. Section 2.3 covers the traditional code analysis techniques and the application of Deep Learning for vulnerability detection. Section 2.4 provides an overview of the most commonly used Deep Learning models in vulnerability detection, including their architectural design and key features. Section 2.8 focuses on the evaluation metrics used to measure the performance of Deep Learning models.

2.1 Software vulnerabilities reports and causes

Society has undergone a rapid digitisation process in which technology has become an integral part of our daily life. During the COVID-19 epidemic, people were compelled to work from home and use internet services for education, entertainment, and communication, which raised technology usage even further [74]. With this digitisation comes the need to ensure the security and stability of the digital systems on which we rely.

As more software is developed and deployed, the risk of vulnerabilities has become a critical issue for both organizations and individuals. The steadily increasing number of software vulnerabilities reported in recent years is concerning, as it shows that software security is not improving. The National Vulnerability Database (NVD) reported a record of 25106 vulnerabilities in 2022, a 24.5% increase compared to 2021 [51].

Open-source software is publicly available software that anyone can modify and redistribute. It is a popular choice among developers as it allows them to build upon existing code and share their work with the community. However, the reuse of code, collaborative development and use of third-party libraries are the main causes of vulnerabilities [48]. Additionally, the use of AI code assistants, such as GitHub Copilot, has been shown to produce significantly less secure code [57].

2.1.1 Vulnerabilities classification

Several classification methods have been established to define and describe the various sorts of vulnerabilities in order to help understand and manage software vulnerabilities. These taxonomies play a crucial role in not only understanding the vulnerabilities' nature but also in preparing adequate countermeasures. They allow organizations to prioritize their efforts and resources in a more organized and efficient manner.

One of the widely recognized classification methods is the Common Vulnerabilities and Exposure (CVE) [1]. It is a dictionary of publicly known information about specific vulnerability instances within a product or system. Each record contains an identification number, a description, and at least one public reference for publicly known vulnerabilities [49].

Another important system is the Common Weakness Enumeration (CWE) [2]. It is a community-developed list of software weaknesses types, providing a common vocabulary for the community to develop tools for identifying these vulnerabilities and improving system security [11]. The NVD uses the CWE to score CVEs.

The Common Vulnerability Scoring System (CVSS) [3] is a standard for quantifying the severity of software vulnerabilities. It provides a numerical score based on the impact of the vulnerability and the likelihood of it being exploited. The CVSS score ranges from 0 to 10, with 10 being the most critical [52].

2.2 Shift-left principle and security in the SDLC

In the current digital landscape, software development plays a crucial role in the success of businesses. The SDLC is a methodology used to manage the development and maintenance of software products. Several models have been proposed to describe the SDLC, including the waterfall, spiral, agile, and DevOps models. Security has traditionally been regarded as a non-functional requirement, only considered at the end of the process, when the software is already developed [59]. However, the increasing concerns regarding software security lead to the creation of models that bring more awareness to this requirement and integrate it into the early stages of the SDLC, such as the DevSecOps model [4].

Shifting-left security is a practice that aims to incorporate security considerations into the SDLC as early as possible rather than as an afterthought. By addressing security issues early in the SDLC, organisations can reduce the effort and cost required to achieve the same level of security. Additionally, shifting-left security minimises technical debt by addressing security flaws earlier during development, rather than after the software is in production [67].

[1] `https://www.cve.org/`
[2] `https://cwe.mitre.org/`
[3] `https://www.first.org/cvss/`
[4] `https://www.devsecops.org/`

Manual code reviews, static code analysis, and penetration testing are commonly employed security practices within the SDLC. However, these practices can be labour-intensive and time-consuming [59], and require specialised tools typically operated by security experts, separately from software development teams. This results in additional overhead and can lead to a lack of ownership of security among the development team [72]. Furthermore, integrating these security practices may negatively impact the development process if not implemented effectively, posing a challenge for organisations prioritising speed and efficiency in the development process. Additionally, development teams may be resistant to incorporating these practices early on as they may view them as an added burden or a delay to the software delivery. Thus current security practices integrated into the SDLC are unsuitable for the shift-left principle.

2.3 Code analysis techniques

As a crucial part of software development, code analysis helps identify potential security vulnerabilities, allowing organisations to take appropriate measures to mitigate the risks that arise from them. However, code complexity and the sheer volume of code that needs to be analysed make this a daunting task. Code analysis tools provide a solution to this problem, but they are not without their own issues. It is critical to consider false positives and false negatives when evaluating the effectiveness of a code analysis tool. When a tool reports a vulnerability that does not exist, this is referred to as a false positive. In contrast, false negatives arise when a tool fails to identify a vulnerability that actually exists. False positives are a significant issue, as they can lead to unnecessary work and cause developers to lose trust in the tool. False negatives, on the other hand, can lead to security vulnerabilities going unnoticed, creating a false sense of security, which can have serious consequences.

2.3.1 Manual code review

Manual code review involves a human analyst carefully reviewing the code by hand to identify potential security weaknesses [19]. It can include looking for common vulnerabilities, such as SQL injection, cross-site scripting, and buffer overflows, as well as reviewing the code for compliance with security standards.

Manual code review is an effective method for identifying vulnerabilities, as it allows a thorough examination of the code in its full context. However, it is the most time-consuming and labour-intensive method of code analysis and does not scalable to large codebases. Furthermore, it is susceptible to human error, as it requires the analyst to deeply understand the code and the vulnerabilities it is sensitive to [7].

2.3.2 Static code analysis

Static analysis involves examining the source code without executing it. It is a rule-based method, meaning analysis is performed against manually predefined vulnerability rules.

Static code analysis tools have 100% code coverage, can detect multiple vulnerabilities, and do not require high-security knowledge as manual code review. Additionally, due to the absence of the need for code execution, these techniques offer quick analysis times and are well-suited for integration into the SDLC at an early stage. Nonetheless, these tools report every vulnerability, even with the slightest probability, and are susceptible to many false positives since they are unable to extract the full context of the code [36]. This can lead to a loss of trust in the tool and unnecessary work to confirm every potential vulnerability [7]. Furthermore, it cannot detect all vulnerabilities, such as those dependent on runtime conditions.

2.3.3 Dynamic code analysis

Dynamic analysis involves analysing the behaviour of code in runtime. As such, these techniques require the code to be valid to execute, which is not always the case, especially earlier in development.

Dynamic analysis tools are known for their high accuracy in detecting vulnerabilities, as they test the program with a range of extreme inputs. Nevertheless, this approach also has several limitations, such as limited code coverage as they only test the executed code, the difficulty and time-consuming process of creating appropriate test sets, and the potential for long runtime times [36]. Additionally, they are susceptible to false negatives since they cannot detect vulnerabilities that are not triggered by the defined test inputs.

2.3.4 Hybrid code analysis

Hybrid code analysis combines the advantages of static and dynamic analysis, providing a more comprehensive view of the code. This approach is capable of identifying vulnerabilities that are not triggered by the defined test inputs, and detect vulnerabilities that are dependent on runtime conditions. Thus can identify a broader range of vulnerabilities and provide more accurate results than the previously mentioned techniques alone. However, this approach naturally inherits the drawbacks of both static and dynamic analysis, such as the need for manually defined rules, a large test set, and the susceptibility to false positives and false negatives [79].

2.3.5 Deep Learning-based analysis

In order to overcome the limitations of the previously mentioned techniques, Deep Learning-based code analysis has been proposed. Deep Learning is a subfield of machine learning that utilizes artificial neural networks to learn from complex data. It has been applied in a variety of domains, including computer vision, natural language processing, and cybersecurity. In the context of code analysis, deep learning has been used for malware identification, prediction of methods names and types, semantic code search, and classification of vulnerable code [66].

Unlike traditional code analysis techniques, deep learning-based code analysis does not require manually defined rules and can learn features from the vulnerable code itself [40]. When trained on an adequate amount of data, DL models can learn general patterns and features of the code, allowing them to perform well on new and unseen code. The use of DL for vulnerability detection, which aims to learn vulnerable code patterns in the training data, has achieved promising results. However, the effectiveness of DL-based code analysis significantly relies on the quality of the training data. Furthermore, the collection of a sufficient amount of data necessary to train the models properly requires much effort. These issues lead to data being the main barrier to adopting these techniques in practice [10].

2.4 Deep Learning models overview

Several deep learning models have been proposed to detect vulnerabilities in software. The appropriate model selection for a given task depends on the type of information one wishes to extract from the source code. Token-based and graph-based models are the popular choices for vulnerability detection [6]. This section provides an overview of some of the most employed DL models for vulnerability detection, including their architecture and key features.

2.4.1 Recurrent Neural Networks

Recurrent Neural Networks (RNN), first proposed by Hopfield [35], are a type of artificial neural networks designed to handle sequential data. They are particularly well-suited for tasks such as natural language processing, speech recognition, and time series analysis, where the goal is to predict the next value in a sequence based on the previous values [18].

RNNs are composed of interconnected layers: the input layer, the hidden layer, and the output layer. The input layer receives the input data, the hidden layer is responsible to processes that data, and the output layer outputs the result. The hidden layer is composed of a series of neurons, each of which is connected to the neurons in the previous layer, as well as to itself through a feedback loop. This loop allows the network to remember the previous states and use them to predict the next output [64], effectively creating a working memory. An illustration of RNN architecture is presented in Figure 2.1.

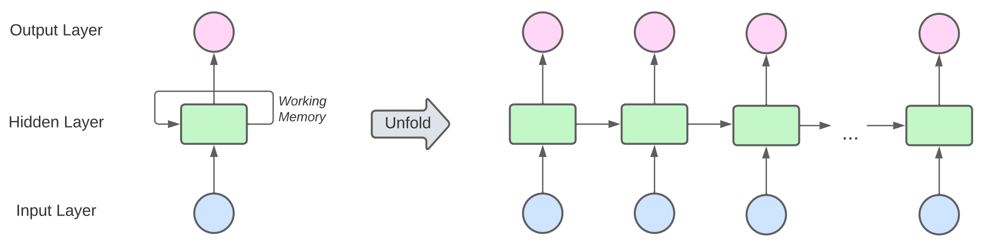


Figure 2.1: RNN architecture

The training of RNNs is typically done using Backpropagation through Time (BPTT), a variant of Backpropagation that allows the network to learn from data sequences. The gradients are computed by propagating the error back through the network, considering the dependencies between the different time steps. However, as the dependencies increase, the gradient calculation becomes increasingly unstable, resulting in either an explosion or decay of the gradients [70]. This is known as the vanishing/exploding gradient problem, which poses a significant challenge in training RNNs on long sequences. The exploding gradient problem can be addressed by implementing techniques such as gradient clipping, which limits the gradient values to a fixed range, or by specifying some maximum number of time steps along which the error is propagated. On the other hand, the vanishing gradients problem is more complex and requires modifications to the basic RNN architecture. To address this issue, Long Short-Term Memory (LSTM) and Gated Recurrent Unit (GRU) network architectures have been proposed [83].

2.4.2 Long Short-Term Memory Neural Networks

The Long Short-Term Memory (LSTM) neural network is a type of RNN that is able to learn long-term dependencies in sequences, overcoming the exploding and vanishing gradients problem by using gates. Hochreiter and Schmidhuber [34] first proposed it, and it has since become one of the most popular RNN architectures.

The LSTM network is composed of memory cells that store information over time. Each cell has an input gate, an output gate, and a forget gate, which control what is stored, read and written on the cell [64]. The input and output gates control the flow of information, preventing irrelevant information from entering or leaving the memory block. The forget gate weights the information inside the cells, allowing the network to forget irrelevant information, which can be helpful to prevent biases [68]. This way, besides integrating the standard working memory of vanilla RNNs, LSTM also have long-term memory mechanisms, excelling on tasks that require the network to remember a limited amount of data over long periods of time [23].

Conventional RNNs analyse only the sequence's direction, making predictions on past data. Bidirectional Long Short-Term Memory (BLSTM) networks are a variant of LSTM in which the output of a particular time step is computed based on both the past and future data [83]. They are created by stacking forward and backward LSTMs, as shown in Figure 2.2. This allows the network to learn from both directions of the sequence, making it more robust to noise and errors, therefore improving the accuracy of the predictions [65]. However, their increased complexity makes BLSTM networks more computationally expensive than their standard counterparts.

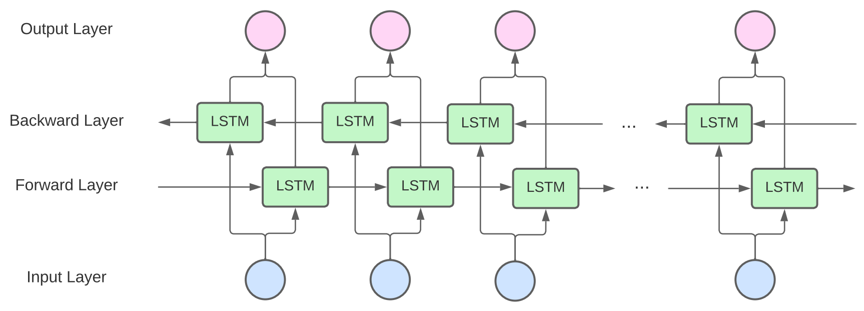


Figure 2.2: Unfolded BLSTM architecture

2.4.3 Gated Recurrent Unit Neural Networks

The Gated Recurrent Unit (GRU) is a simpler variant of LSTM proposed by Cho *et al.* [8]. It comprises two gates: the reset gate and the update gate. The update gate is responsible for the flow of information from the previous state to the current state. The reset gate controls the flow of information from the current input to the current state, deciding how much of the past information to forget.

Since GRU networks are simpler than LSTM, they are more computationally efficient. They are faster to train and can even outperform LSTMs in the long text and small dataset scenario, achieving a higher performance-cost ratio [82]. GRUs can also be employed in a bidirectional manner, like LSTMs, resulting in Bidirectional Gated Recurrent Unit (BGRU) networks.

2.4.4 Gated Graph Neural Networks

Gated Graph Neural Networks (GGNNs) are a type of neural network designed to handle graph-structured data. They were first proposed by Li *et al.* [39], as a variation of the more general Graph Neural Networks (GNNs) with gating mechanisms similar to GRUs, and use BPTT to compute the gradients.

Vectors and the edges by adjacency matrices represent the nodes. In order to update the nodes, a series of operations are performed on the vectors of the current node, the vector of the previous node, and the adjacency matrix. The operations are carried out by a series of gates, which control the flow of information between nodes. This allows the network to focus on essential information while handling noisy or irrelevant data more effectively.

GGNNs are well-suited for various applications, including natural language processing and code analysis, due to their ability to handle graphs of varying sizes and shapes. However, one downside of this type of network is that it represents edge information as label-wise parameters, which may pose issues even for small-sized label vocabularies [2].

2.4.5 Transformer

The Transformer is a sequence-to-sequence model first proposed in 2017 [75] and has since become the *de facto* standard for many NLP tasks, surpassing RNN models in results while being faster [26]. Transformers discard the recurrent nature of RNNs and instead use attention mechanisms to model the dependencies between elements in a sequence.

The Transformer's architecture, represented in Figure 2.3, consists of an encoder and a decoder, which are stacks of identical layers composed of a multi-head self-attention, position-wise feed-forward network and normalization mechanisms. The encoder is responsible for mapping the input sequence to a vector representation, whereas the decoder generates the output sequence of symbols, one at a time. An attention mechanism connects the encoder and decoder, allowing the decoder to focus on specific parts of the input sequence. Since this model does not use any recurrence or convolution, positional encoding must be added to the input sequence to provide the model with information about the relative or absolute position of the tokens in the sequence.

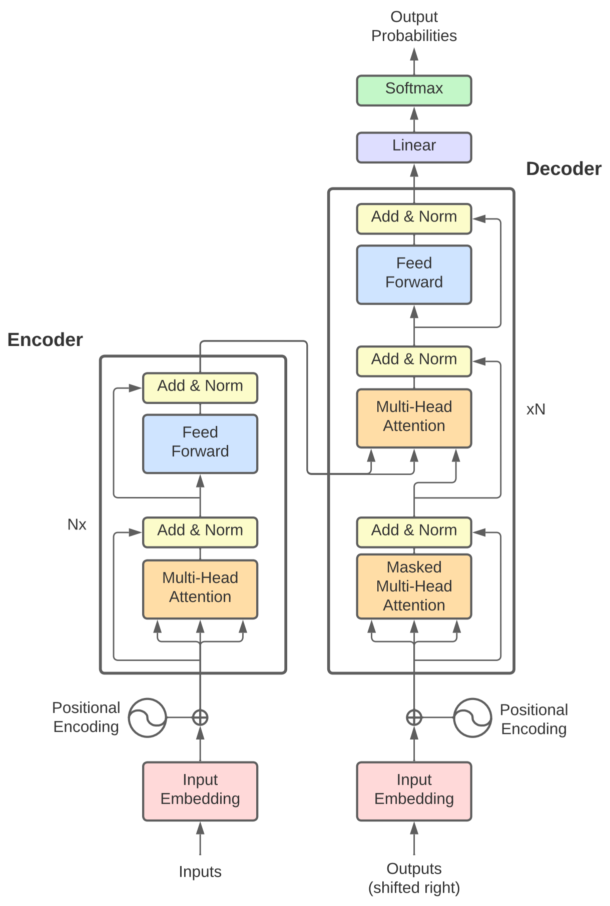


Figure 2.3: Transformer model architecture

This architecture can be used in three different ways [43]:

- Encoder-decoder: Uses the complete architecture, as described above, for sequence-to-sequence tasks, such as machine translation.
- Encoder-only: Uses only the encoder, usually applied to natural language understanding tasks, such as text classification and sequence labelling.
- Decoder-only: Uses only the decoder, also removing the encoder-decoder attention component. This is employed for language modelling tasks, such as sequence generation.

Self-attention mechanism

The attention mechanism is the crucial component of the Transformer model. It allows the model to focus on specific elements of the input sequence during processing and generate output dependent on the input. Multihead attention (comprising multiple attention heads) is used to implement this mechanism, which allows the model to simultaneously attend to information from different representation subspaces at different positions [75].

The Transformer uses three matrices to calculate attention: Query, Key, and Values. These are previously calculated by multiplying the input embeddings with three weight matrices, W_Q, W_K, and W_V. Each self-attention head calculates the scaled dot-product attention, which is the dot product of the query and key vectors, divided by the square root of the dimensionality of the key vectors. This dot product is then passed through a softmax function to obtain the attention weights, which are then multiplied by the values matrix, finally obtaining the output. Then, the outputs of all the different heads are concatenated and passed through a linear transformation, proceeding to the normalization and feed-forward layers.

As benefits of the self-attention layers, Vaswani *et al.* [75] state the reduction of the computational complexity when compared to recurrent layers, the ability to model long-range dependencies, and the parallelization of the computation. Additionally, the authors emphasize that self-attention can result in more interpretable models, as it is easier to understand what parts of the input sequence the model is focusing on.

2.4.5.1 BERT

BERT (Bidirectional Encoder Representations from Transformers) [13] is a Transformer-based model that uses only the encoder blocks, developed By Google Research and released in 2018. Due to its bidirectional nature, BERT can learn the context of a word based on the words that come before and after it in a text.

The model is pre-trained on a large corpus of English text, using two unsupervised tasks: Masked Language Modeling (MLM) and Next Sentence Prediction (NSP). MLM is a task where the model is trained to predict the masked words in a sentence, obtaining a bidirectional pre-trained model. NSP is used to train the model to understand the relationship between sentences. These pre-training techniques allow the model to learn general language understanding, which can then be adapted to downstream NLP tasks.

BERT can be fine-tuned for various NLP tasks without requiring significant architecture changes: the input and output layers for the specific task are added, and the parameters are fine-tuned on the new dataset. Compared to pre-training, fine-tuning is relatively inexpensive [13] and enables BERT to achieve state-of-the-art results on many NLP tasks, such as question answering, sentiment analysis, and text classification.

There are several sizes of BERT models, which differ in the number of encoder layers, the hidden size, and the number of attention heads. Devlin *et al.* [13] present two BERT models: BERT-base and BERT-large. BERT-base has 12 encoder layers, 768 hidden units, 12 attention heads, and 110 million parameters, while BERT-large has 24 encoder layers, 1024 hidden units, 16 attention heads, and a total of 340 million parameters. The model choice will depend on the downstream task and the available computational resources.

For the input representation, BERT uses WordPiece embeddings [80] with a 30000 token vocabulary. The WordPiece tokenization uses a vocabulary of subword units to tokenize words, allowing the representation of rare words or out-of-vocabulary words as a sequence of subwords.

BERT vocabulary also contains special tokens, such as the '[CLS]' token, which represents the beginning of a sentence, and the '[SEP]' token, which is used to separate two sentences. Once the input text has been tokenized, each token is then represented as an integer, between 0 and the vocabulary size (in this case, 30522). Afterwards, the input is truncated or padded to a maximum length of 512 tokens.

2.4.5.2 BERT variants

BERT has undoubtedly been one of the most remarkable models in the NLP field, and many variants have been proposed to improve its performance or tailor it to specific tasks. DistilBERT [62] is a smaller version of BERT that has 40% fewer parameters than BERT, while being 60% faster. These improvements were achieved by distilling the knowledge from a larger pre-trained model into a smaller one. RoBERTa (Robustly optimized BERT approach) [44] improved BERT by training the model longer, with more data and bigger batches. Additionally, the NSP task was removed, and a dynamic masking strategy was adopted.

There are also BERT variants designed to handle code and programming-related text better, which are more relevant for this thesis. CodeBERT [20] is the first bimodal model, pre-trained on both natural language and six programming languages (Python, Java, JavaScript, PHP, Ruby, and Go). It outperformed other state-of-the-art models on several code-related downstream tasks, such as natural language code search and code-to-documentation generation. JavaBERT [12] is a BERT variant pre-trained on Java code for MLM task. It outperformed CodeBERT in the said task, indicating that CodeBERT cannot predict Java tokens as accurately as JavaBERT.

2.5 Intermediate code representations

Before exploring the state-of-the-art DL-based vulnerability systems, it is essential to understand the most common intermediate code representations used and their importance. Instead of feeding raw source code to the models, current systems use intermediate representations to extract features and incorporate syntactic and semantic information.

Abstract Syntax Tree (AST) is commonly used as the initial intermediate representation generated by compilers to examine syntactic errors in the code [78]. It is an ordered tree-like representation of the source code, where each node represents a statement, expression or operand. This representation is considered abstract because it captures the structure of the programming language rather than its concrete syntax [71]. Despite ASTs being easy to parse and providing a good represenation of the code structure, proving useful for similar code detection, they are not suitable for more complex code, as they fail to incorporate control flow and data dependencies [81].

Control Flow Graph (CFG) is a representation of the flow of control in a piece of code, represented as a directed graph. Each node in the graph corresponds to a basic block of code (statement and predicates), and the edges represent the flow of control in the code [71]. CFGs can be constructed from ASTs by first considering sstructured control statements, such as if, while, and for statements, to form a preliminary CFG. This preliminary CFG is then augmented with control flow information from unstructured control statements, such as goto, break, and continue statements, to create a complete CFG representation of the code. From a security perspective, CFGs have become a standard code representation for understanding programs through reverse engineering, as they reveal the control flow of an application. They also prove valuable in detecting variations of known malicious applications and guiding fuzz testing tools. However, CFGs do not capture data flow within the code, thus failing to detect statements that process information that an attacker has manipulated. [81].

Program Dependence Graph (PDG) explicitly represents data dependencies within code. It is also a directed graph where the nodes represent a predicate or statement, like CFG, but the edges connecting the nodes represent control dependencies and data dependencies [71]. PDG is a joint data structure combining the Data Dependence Graph (DDG) and the Control Dependence Graph (CDG). The DDG component is composed of data dependency edges that illustrate the dependence or effect of one variable on another. The CDG component, in contrast, consists of control dependency edges that demonstrate the impact of a predicate on the values of variables. [55].

Each of these representations alone is insufficient to characterize a vulnerability type. In order to capture both syntactic and semantic features of the code, Yamagushi *et al.* [81] proposed **Code Property Graph (CPG)**, a joint data structure that combines the advantages of AST, CFG and PDG. To construct the CPG, the AST, CFG and PDG representations of the code are modelled as property graphs and merged afterwards into a single graph representation. It enables modelling patterns for common vulnerabilities in graph traversals that can be refined to control FPR and FNR. With this novel representation, they could cover considerably more vulnerability types.

2.6 Transfer learning and Domain adaptation

Transfer learning and domain adaption are two interrelated strategies in machine learning that deal with the idea of applying knowledge learned in one situation to another. The key difference lies in what is being transferred. In domain adaptation, the focus is on adapting to new data distribution, while in transfer learning, the focus is on leveraging knowledge from a task to a related one.

Transfer Learning is the process of using a pre-trained model as the starting point for a new related task. The idea is to leverage the knowledge gained from the initial task (source task) to improve learning in a new, related task (target task). This way, we can leverage the knowledge gained from a large dataset to solve a different problem. This approach is particularly useful when the target task has limited labelled data, as the pre-trained model provides a strong foundation and learned representations that can benefit the new task [54].

One popular transfer learning method is **fine-tuning**, where the pre-trained model's weights are used as an initialization for the new model and then further trained using the target task data. This allows the model to adapt its learned representations to the specifics of the target task, leading to improved performance.

Domain Adaptation is a form of transfer learning used when the training data (source domain) differs from the data we want to make predictions on (target domain) while the task remains the same. The goal is to bridge the gap between the source and target domains, enabling the model to generalize well to the target domain [3].

In the case of deep learning models, we can reduce this domain shift by fine-tuning the model that was trained on the source domain on the target domain [1].

Both of these techniques are fundamental to the success of deep learning models, as they can enhance performance and promote generalization. They allow models to leverage the knowledge gained from pre-training on large-scale datasets, reducing the need for extensive labelled data and accelerating training time.

2.7 Classification tasks in ML

Classification problems are categorised based on how labels are associated with input samples, resulting in two main types: **single-label** and **multi-label** classification [14]. Single-label classification refers to the process of learning from a collection of samples, each linked to a single label l from a set of disjoint labels L. This problem can be further divided into two categories: **binary** and **multiclass** classification. On the other hand, in multi-label classification, each sample in the dataset is associated with a set of labels Y, where Y is a subset of the label set L ($Y \subseteq L$) [85].

The most basic type of classification is binary classification, in which instances fall into two mutually exclusive classes. It is employed when a true/false decision is required. In vulnerability detection, for example, binary classification can be used to identify whether a method is vulnerable. Because binary classification models just involve discriminating between two classes, they are often easy to develop and interpret.

Multiclass classification extends binary classification by allowing instances to be assigned to one of several classes. In this approach, each instance is assigned a single label from a predefined set of classes. For example, multiclass classification can be used in vulnerability detection to categorise methods into multiple vulnerabilities, such as injection, authentication, or authorisation. Multiclass classification models are capable of dealing with more complex problems involving multiple classes.

Multi-label classification takes the classification process a step further by allowing instances to be assigned multiple class labels simultaneously. This approach acknowledges that methods might belong to more than one class simultaneously. For example, a single method can be vulnerable to various attacks in vulnerability detection. These complicated relationships can be captured by multi-label classification models, which yield more nuanced predictions. Multi-label classification, on the other hand, can be more challenging to implement than binary or multiclass classification.

When deciding which type of classification to use, it is essential to consider the specific requirements and characteristics of the problem at hand. Moreover, the choice of classification type can also depend on the available dataset and the desired specificity of the predictions. Each classification type has advantages and limitations, and selecting the right strategy is crucial for constructing a successful classification model.

2.8 Evaluation metrics

Evaluating the performance of a vulnerability detection model is a crucial step to ensure the reliability and accuracy of the results. In the literature review [40, 86, 60, 69], when comparing the performance of different models, the authors use several standard metrics such as accuracy, precision, recall, f1-score, false positive rate, and false negative rate.

A confusion matrix is a tabular representation of the performance of a classification model. It compares the predicted values of the model with the actual values, and presents the results in a format that allows easy identification of instances where the model may be misclassifying (confusing one class for another). The matrix is typically composed of four elements: number of **True Positives (TP)**, **True Negatives (TN)**, **False Positives (FP)**, and **False Negatives (FN)**.

TP: positive sample correctly predicted as positive.

TN: negative sample correctly predicted as negative.

FP: negative sample incorrectly predicted as positive.

FN: positive sample incorrectly predicted as negative.

We can use these values to calculate the **False Positive Rate (FPR)** and **False Negative Rate (FNR)**. The measures the proportion of negative samples that are misclassified as positive. In the context of vulnerability detection, that means the proportion of non-vulnerable samples incorrectly classified as vulnerable. It can be calculated according to the formula:

$$FPR = \frac{FP}{FP + TN} \tag{2.1}$$

The FNR measures the proportion of positive samples incorrectly classified as negative. In the context of vulnerability detection, that means the proportion of vulnerable samples incorrectly classified as non-vulnerable. The formula is as follows:

$$FNR = \frac{FN}{FN + TP} \tag{2.2}$$

Confusion matrices are also helpful to derive other complex metrics that provide a more detailed insight into the model's performance:

Accuracy is the proportion of samples that are correctly classified. It is a typical metric used to judge model performance. However, it is only suitable when the classes are balanced. For example, if we have a dataset where 90% of the samples are non-vulnerable, and the rest 10% are vulnerable, a model that always predicts non-vulnerable will have an accuracy of 90%. In this case, the model is biased, but it still has a high accuracy of 90%.

$$Accuracy = \frac{TP+TN}{TP+FN+TN+FP} \tag{2.3}$$

Precision is the proportion of positive samples that are correctly classified. It is focused on controlling the number of False Positives.

$$Precision = \frac{TP}{TP+FP} \tag{2.4}$$

Recall is the proportion of positive samples that are correctly classified. An emphasis is placed on controlling the number of False Negatives.

$$Recall = \frac{TP}{TP+FN} \tag{2.5}$$

F1-score is a metric that measures the balance between precision and recall, and is calculated as the harmonic mean of these two values. The harmonic mean is preferred over the arithmetic mean due to its ability to penalize extreme values. For example, if the precision is 1 and the recall is 0, the f1-score will be 0. However, if the precision is 0.5 and the recall is 0.5, the f1-score will be 0.5. Therefore, f1-score is particularly helpful when dealing with imbalanced datasets, where precision and recall are often in conflict. Maximizing the this metric limits both false positives and false negatives as much as possible.

$$F1 = \frac{2 * Precision * Recall}{Precision + Recall} \tag{2.6}$$

To evaluate the performance of language models, we can use **perplexity**. It measures how well a probability model predicts a sample. A lower perplexity score indicates that the model is better at predicting the sample. We can define perplexity as the exponentiation of the entropy of the model with the following formula:

$$Perplexity = e^{L_{CE}} \tag{2.7}$$

where L_{CE} is the **cross-entropy loss** of the model. The "loss" in machine learning, also referred to as the cost function, is a measure of how well a machine learning model can predict the correct output. It quantifies the disparity between the predicted and actual values for an instance in the dataset. Cross-entropy is a popular loss function used in ML classification tasks. It is defined, for N classes, as follows:

$$L_{CE} = -\sum_{i=1}^{N} t_i \log(p_i) \tag{2.8}$$

where t_i and p_i are the true label and the predicted probability of the sample belonging to the i-th class, respectively [38].

2.9 Learning curves in Machine Learning

A learning curve is a graphical representation of the generalisation error of a model as a function of the number of training samples [56]. In the context of classification problems, these curves typically include both training and validation errors. By analysing the shape of the learning curve, we can determine whether the model suffers from high bias (underfitting) or high variance (overfitting) or whether the model is well suited for the data (good fit) [76].

In the case of underfitting, the model is too simple to learn the underlying structure of the data or the data has no inherent pattern for the task at hand. This scenario is characterised by a high error on both training and validation sets, and both values hit a plateau, with a small gap between them, as the number of training samples increases.

On the other hand, overfitting occurs when the model is too complex and starts to learn the noise in the training data, or the dataset is too small, and the training data does not represent the underlying distribution of the data. In this case, the training error decreases over time, while the validation error rises after a certain point as the model memorises the training data and fails to generalise to unseen data. A large gap between the training and validation errors characterises this scenario.

Diagnosing the shape of the learning curve is a crucial step in building a machine learning model. It helps determine the model's suitability for the data or whether we need to collect more data or use a more complex model. Additionally, it allows us to determine the best epoch to stop the training process to avoid overfitting.

2.10 GitHub Actions fundamentals

GitHub Actions is a continuous integration and continuous delivery (CI/CD) platform provided by GitHub, designed to automate software development workflows directly within a repository. It is designed to help developers streamline their workflow by automating tasks such as testing code, deploying software, and much more. Detailed information about creating, managing and publishing Actions can be found in the official documentation[5].

One can configure a GitHub Action **workflow** to be automatically triggered when an **event** occurs in the repository. This **workflow** can have one or more **jobs**, each executed inside a virtual

[5] `https://docs.github.com/en/actions`

machine **runner**. **Jobs** contain one or more **steps** that either run a script or an **action**. These GitHub Actions components are described in more detail below [29]:

Events are specific activities in a GitHub repository that can trigger a workflow run. These can be pushing code to a branch, opening a new issue, creating a pull request, etc.[6]. Workflows can also be triggered on a schedule.

A workflow is a configurable automated process made up of one or more jobs. Workflows are defined using YAML files and stored inside the *.gitgub/workflows* directory in a repository. A repository can have several workflows, each for a different purpose, and workflows can be referenced within other workflows to reuse common steps.

Jobs are a set of steps in a workflow. Each job runs in a separate virtual machine and can be configured to run in parallel or sequentially. By default, a workflow with multiple jobs will run those jobs in parallel and only finish when all jobs have been completed successfully. However, jobs can be configured to run sequentially by defining dependencies between them.

Actions are custom applications that perform a specific task in a workflow. They are the smallest portable building block of a workflow and can be used to encapsulate common tasks in reusable ways. Developers can build their own actions or reuse actions created by the GitHub community available in the GitHub Marketplace[7].

Runners are servers responsible for running the jobs inside a workflow. GitHub provides hosted runners for Linux, Windows and MacOS, but users can also host their own runners on their own infrastructure. GitHub runners are free for public repositories, and free-tier GitHub accounts have a limit of 2000 minutes per month for private repositories. This limit can be increased by upgrading to a paid plan.

Listing 2.1 shows a simple workflow example that is triggered when a change is pushed to the *main* branch. This workflow, named *example-workflow*, has a single job named *example-job* that runs on a Linux virtual machine. This job has two steps: the first one uses an action to checkout the repository code to the machine, and the second one echoes a messaage.

[6]`https://docs.github.com/en/actions/using-workflows/events-that-trigger-workflows`
[7]`https://github.com/marketplace`

```yaml
# Name of the workflow (optional).
name: example-workflow

# Specify the trigger for this workflow.
on:
  push:
    branches:
      - main

# Define the list of jobs that make the workflow.
jobs:
  # Name of the first and only job.
  example-job:

    # Specify the runner to use.
    runs-on: ubuntu-latest

    # Define the list of steps to run.
    steps:
      # The uses keyword indicates that this step runs an action.
      - name: Checkout code
        uses: actions/checkout@v3

      # The run keyword tells the job to execute a command on the runner.
      - name: Echo a success message
        run: echo "Repository code was successfully checked out"
```

Listing 2.1: GitHub workflow example.

Chapter 3

State of the Art

This chapter reviews the current state of Deep Learning-based vulnerability detection. Section 3.1 analyses the contributions of the most relevant systems in the field, including their architecture, datasets, and results. Section 3.2 discusses the different types of datasets available and highlights the lack of a publicly available Java dataset suitable for vulnerability detection. Section 3.3 provides an overview of the security tools available on the GitHub Marketplace.

3.1 Deep Learning-based vulnerability detection systems

DL-based vulnerability detection systems have emerged as a powerful tools in the field of software security. These systems leverage the power of DL models, such as the ones presented in Section 2.4, to automatically identify vulnerabilities in software systems, allowing for more efficient and effective detection and mitigation of security threats. This section provides a high-level overview of current DL-based vulnerability detection systems, emphasising key contributions, data sources, and techniques employed.

3.1.1 VulDeePecker

In 2018, Li *et al.* [42] introduced the first deep learning-based vulnerability detection system, VulDeePecker, which demonstrated the potential of deep learning models to detect vulnerabilities. This system operates as a binary classifier, classifying source code related to library/API function calls as either vulnerable or non-vulnerable. The authors also created the first dataset specifically designed for deep learning approaches, focusing on buffer error and resource management vulnerabilities in the C/C++ programming languages, and is derived from both the National Vulnerability Database (NVD)[1] and the Software Assurance Reference Dataset (SARD)[2].

The VulDeePecker system comprises two phases: the learning phase and the detection phase. For the intermediate source code representation, the authors introduced the concept of code gadgets. Code gadgets are lines of code that are semantically related to one another regarding data

[1]https://nvd.nist.gov/
[2]https://samate.nist.gov/SARD/

dependency. During the learning phase, code gadgets are generated, transformed into a vector representation using word2vec [3], and then fed along with their labels to a BLSTM neural network for training. In the detection phase, code gadgets are generated from the source code of the targeted programs, transformed into a vector representation, and then fed to the trained BLSTM neural network for classification as vulnerable or non-vulnerable. In both phases, when converting the code gadgets to vectors, they are submitted to a normalization step to remove irrelevant information. Non-ASCII characters and comments are removed, and then user-defined variables and function names are replaced with generic names (e.g. "VAR1" and "FUNC1").

VulDeePecker outperformed the state-of-the-art vulnerability detection systems, achieving an F1 score of 90.5%, FPR of 5.7% and FNR of 7%.

3.1.2 μVulDeePecker

Zou *et al.* [89] presented μVulDeePecker, an extension of VulDeePecker, the first deep learning-based system for multiclass vulnerability detection. This system can identify the type of vulnerability present in source code by refining code gadgets through the integration of control dependency and introducing the concept of code attention. This mechanism captures more information about a statement and is the main contributor to the system's multiclass capability. In both the learning and detection phases, code attentions are generated from normalized code gadgets.

A new dataset was created, from SARD and NVD, as the original VulDeePecker dataset was unsuitable. It lacked information about vulnerability types, did not contain data control dependency, and lost some statements in the code gadgets that contribute to identifying the vulnerability type. The experiments on the testing set of the new dataset yielded impressive results, with a multiclass F1 score of 94.22%, a multiclass False Positive Rate of 0.02%, and a multiclass False Negative Rate of 5.73%.

3.1.3 SySeVR

SySeVR [41] is another successor to the VulDeePecker system, developed to address its limitations. This system is designed to classify source code as vulnerable or not at a slice level. In contrast to the previous system, SySeVR discards the use of code gadgets and code attention, instead utilizing the concepts of Syntax-based Vulnerability Candidates (SyVCs) and Semantics-based Vulnerability Candidates (SeVCs). SyVCs reflect the syntax characteristics of vulnerabilities, while SeVCs extend this to include the semantic information induced by data and control dependencies. A new dataset was created to support this system, comprising 126 types of vulnerabilities sourced from SARD and NVD. Several machine learning and deep learning models were tested, with the best-performing model being the BGRU, achieving an F1 score of 92.6%, a False Positive Rate of 1.4%, and a False Negative Rate of 5.6%.

[3] https://radimrehurek.com/gensim/models/word2vec.html

3.1.4 Devign

Devign [87] uses a novel graph neural network model to classify a function as vulnerable or not. It comprises three components: a graph embedding layer, a gated graph recurrent layer, and a Conv layer. The graph embedding layer is used to generate a graph representation of the source code, which is composed of various subgraphs: Abstract Syntax Tree (AST), Control Flow Graph (CFG), Data Flow Graph (DFG), and Natural Code Sequence (NCS). This joint representation enables the model to capture the source code's syntactic and semantic information. The gated graph recurrent layer employs GGNN to learn node features by aggregating and passing information about neighbouring nodes in graphs. Finally, the Conv module is used to extract the features of the graph representation and classify the source code as vulnerable or not. The Devign dataset was created using code from 4 large open-source C projects (Linux Kernel, QEMU, Wireshark, and FFmpeg) and manually labelled by four security experts.

3.1.5 ReVeal

In their study, Chakraborty *et al.* [6] explored the generalizability of state-of-the-art vulnerability detection models, including VulDeePecker, SySeVr, and Devign. The results indicated that none of these models performed well in real-world settings, with an average drop of 73% in performance. The leading causes for this decline were the use of simple token-based models, data duplication between training and testing datasets, and data imbalance. To address these issues, the authors proposed the ReVeal system. This system employs CPG to extract syntactic and semantic information from the source code and a Gated Graph Neural Network (GGNN) model to learn node features. The authors created a robust and comprehensive real-world dataset from the Linux Debian Kernel and Chromium open-source projects. They filtered out non-security commits and then, for each patch, labelled the previous versions of all changed functions as vulnerable. The Synthetic Minority Over-sampling Technique (SMOTE) was employed during the model's training phase to address the imbalance in the number of vulnerable and non-vulnerable functions.

3.1.6 First Transformer-based vulnerability detection system

Ziems and Wu [88] were the first to propose a Transformer-based vulnerability detection system. They use the base BERT model, fine-tuning it to create a model that identifies the type of vulnerability present in source code, with file-level granularity. A custom dataset focused on C/C++ was built from the SARD, containing over 100,000 files and 100 CWEs. Some preprocessing was applied, including removing all comments and replacing function names with generic names, similar to the approach taken by VulDeePecker [42].

The authors tested traditional LSTM, BLSTM, and BERT models. The inputs to these models were the entire code files that were tokenized into words and sub-words. However, the BERT model proposed had a limit of 256 tokens, so the token sequence was split into several chunks. Three different experiments were performed to address this limitation. In the first experiment, each chunk was fed separately into the model, and a vanilla RNN took the output of each BERT segment. A softmax classifier was applied to the final output, calculating the probability of each CWE. The second and third experiments were similar, but a LSTM and a BLSTM network replaced the RNN, respectively. In the second experiment, contextual information was kept between each sequence of tokens. In the third experiment, due to the bidirectional nature of the BLSTM, the model could learn the context from both directions.

The LSTM and BLSTM models achieved only 72% and 79% accuracy, respectively, while the BERT model achieved 85% accuracy. Employing BERT with the LSTM (second experiment) improved the results substantially, achieving 93.19% accuracy. BERT+BLSTM (third experiment) achieved the best accuracy, 93.49%. These results demonstrate that BERT performs better than RNN-based models and that it is crucial to keep contextual information across the sequence of tokens for optimal performance.

3.1.7 ISVSF

The intelligent sentence-level vulnerability self-detection framework (ISVSF) [84] was designed to address the limitations of the representation scheme and pattern mining of previous approaches. Unlike token-level approaches, the sentence-level approach divides the input sequence into designated sub-blocks.. Control flow Abstract Syntax Tree (CFAST) is used as an intermediate representation.

The authors obtained samples from NVD, SARD and GitHub to create a representative dataset focused on Java programming language. These samples were then labelled as vulnerable if they contained CWE or CVE tags, while the remaining samples were labelled as non-vulnerable. The dataset includes a total of 118 types of CWEs. According to the authors, synthetic and semi-synthetic data from SARD and NVD can improve the model's ability to extract vulnerability patterns, whereas real data from GitHub is more representative of real-world scenarios. This data composition enables the model to generalize more effectively.

BLSTM was the chosen classifier for ISVSF, which predicts whether a Java method is vulnerable. Additional models were created to compare the performance of ISVSF with other state-of-the-art models. The baseline model was inspired in VulDeePecker, using AST to express source code (instead of code gadgets), word2vec to embed the AST and a BLSTM as the classifier. Another model, ITVSF (Intelligent Token-level Vulnerability Self-detect Framework), is similar to ISVSF but uses BERT instead of word2vec to embed the words. The results showed that ISVSF outperformed both models, achieving an F1 score of 96.58%, with a 1.35% FNR and 4.73% FPR.

3.1.8 LineVul

LineVul [22] is a Transformer-based approach for line-level vulnerability prediction. This system has a two-step approach: function-level vulnerability and live-level vulnerability prediction. The first step performs Byte Pair Encoding (BPE) tokenization on the source code. Then a customized Transformer model, fine-tuned on the CodeBERT model, performs binary classification on the function level. In the second step, for predicted vulnerable functions, the system utilises the

Transformer's self-attention mechanisms to identify the vulnerable lines. The authors posit that tokens that contribute the most to the classification of the vulnerable function are likely to cause the vulnerability.

The dataset created by Fan *et al.* [17] was used, since it was the only one that provided the ground truth for the line-level vulnerability, unlike Devign [87] and ReVeal [6] datasets which only provide the function-level vulnerability. This dataset comprises C/C++ code from 348 open-source Github projects and contains 91 different CWEs. LineVul achieved an F1 score of 91% for function-level vulnerability detection. When utilising the same dataset in other state-of-the-art models, their results dropped significantly. VulDeePecker achieved an F1 score of 19%, Devign achieved 26%, and ReVeal achieved 30%. Regarding live-level vulnerability prediction, LineVul achieved a top-10 accuracy of 65%.

3.1.9 VDET for Java

VDET for Java [46], which stands for Java Code, is a Transformer-based IDE extension that detects vulnerabilities in Java code. In this system the source code is used as input, and fine-tuning is performed on the JavaBERT model to create a model capable of performing multi-label classification with method-level granularity. Thus, the model can predict the presence of multiple vulnerabilities in a single method.

The dataset generated derives from the Juliet Test Suite for Java. Preprocessing was performed to remove comments and to replace function names with neutral names. In order to address the issue of imbalanced classes, the least significant vulnerabilities were removed, leaving 21 CWEs and a total of 134645 samples. Additionally, long code sequences were removed due to BERT architecture limitations, resulting in a final dataset of 115600 samples, each with a maximum size of 512 tokens.

The model achieves 99% accuracy in the conducted experiments, 94% weighted F1 score, 7% mean FNR and 10% mean FPR. Mamede [46] argues that the results are satisfactory but highlights the need to evaluate the model in a real-world scenario and to train it on a larger dataset.

3.1.10 Summary

This section analysed the current state-of-the-art of DL-based vulnerability detection. Each approach has its characteristics, summarised in Table 3.1 for a better overview.

Table 3.1: Fundamental characteristics of the reviewed systems

System	Classification	Programming Language	Detection Granularity	Features Representation	Dataset Type	DL Model
VulDeePecker [42]	Binary	C/C++	Slice-level	Code gadgets	Synthetic and semi-synthetic	BLSTM
μVulDeePecker [89]	Multiclass	C/C++	Slice-level	Code gadgets	Synthetic and semi-synthetic	BLSTM
SySeVR [41]	Binary	C/C++	Slice-level	SyVC, SeVC and vector representations	Synthetic and semi-synthetic	BGRU
Devign [87]	Binary	C	Function-level	CPG	Real-world	GGNN
ReVeal [6]	Binary	C/C++	Function-level	CPG	Real-world	GGNN
First Transformer-based [88]	Multiclass	C/C++	File-level	Source code	Synthetic and semi-synthetic	BERT
ISVSF [84]	Binary	Java	Sentence-level	AST	Synthetic, semi-synthetic and real-world	BLSTM
LineVul [22]	Binary	C/C++	Function and line-level	Source code	Real-world	CodeBERT
VDET for Java [46]	Multi-label	Java	Method-level	Source code	Synthetic	JavaBERT

3.2 Available datasets

Deep-learning software vulnerability detection is a data-driven task. Large-scale datasets with high-quality labelled samples are essential to ensure the quality of the predictions [66]. Datasets can be classified into three categories, based on the type of samples and the way they are labelled:

- Synthetic datasets comprise code and labels artificially created. Many code patterns can be synthesized, and a potential benefit is the generation of vulnerable patterns that rarely occur in real-world code [66]. However, the quality of the labels highly depends on the annotation techniques, and the code patterns are unrealistic. Juliet test suits [4] fall into this category.

- Semi-synthetic datasets contain data that is either artificially generated samples or annotated. For instance, the SARD and NVD datasets modify actual production code to identify vulnerable patterns. While these datasets offer a more realistic representation than purely synthetic datasets, they still fall short of capturing the full intricacies of real-world vulnerabilities due to the inherent simplifications and abstractions applied [6].

- Real-world datasets are collected from real-world sources such as GitHub open-source projects. The samples are actual production code, and the labels are manually annotated or directly obtained from vulnerability-related commits. However, the reliability of the labels is not guaranteed, and these datasets usually present issues with data imbalance, as the number of vulnerable samples is much smaller than the number of non-vulnerable samples [10].

Most of the research conducted in the field of software vulnerability detection focus on detecting vulnerabilities in C/C++ code, as shown in Section 3.1, so most publicly available datasets

[4] https://samate.nist.gov/SARD/test-suites

are related to this language. Zhang *et al.* [84] and Mamede [46] mention the lack of a real-world dataset for vulnerability detection in Java. Even though Zhang *et al.* [84] created a fine-grained dataset for Java, they did not make it publicly available. Currently, the only available resource at our disposal is the Juliet Test Suit for Java from the National Institute of Standards and Technology (NIST) [5] . This synthetic dataset alone is not desirable for our purposes, as it limits the generalizability of the models trained on it.

3.3 Security tools available on GitHub Marketplace

GitHub Marketplace is a platform that allows developers to discover, install, and publish GitHub Apps, OAuth Apps, and GitHub Actions [31]. The automation provided is the key benefit of the Actions, as it allows developers to schedule security tasks on their repositories, alleviating the effort of running the tools manually. This can improve the development process's efficiency and speed while reducing the risk of security vulnerabilities.

GitHub Marketplace currently has over 900 security-related Actions [28]. These tools can be used to analyse code, search repositories for sensitive data leaks, and find vulnerabilities in various programming languages' source code. However, most tools undertake static analysis, and not all are free. To the best of our knowledge, there is currently no GitHub Action that employs deep learning models for vulnerability detection.

[5]https://samate.nist.gov/SARD/test-suites/111

Chapter 4

Problem and proposed solution

This chapter outlines the problem statement and the proposed solution. In section 4.1, we highlight the unsolved issues drawn from a comprehensive review of the existing literature. Section 4.2 presents the latent opportunities that we have identified, revealing the potential for innovation and progress. Section 4.3 specifies the proposed solution, including the research questions that will guide our work. Finally, section 4.4 outlines the system requirements and the evaluation criteria for measuring the proposed solution's performance and usability.

4.1 Research Gap

Chapter 3 delves into the current state-of-the-art of DL-based vulnerability detection, providing an overview of the existing systems and frameworks. While the field of Deep Learning-based vulnerability detection has made substantial progress, several research gaps still exist that present opportunities for future work.

Although several DL-based vulnerability detection systems have been developed and refined over recent years, most of these have been designed with a focus on C/C++ code. Despite the widespread use of Java in software development, **there is a marked lack of equivalent tools for detecting vulnerabilities in Java code**. Moreover, while Transformer-based systems utilise BERT variants like CodeBERT [20] and JavaBERT [12], which are pre-trained on general-purpose code, **no existing studies have investigated the impact of utilising models pre-trained on vulnerability-related code for vulnerability detection**.

This scarcity of research in DL-based Java vulnerability detection might be attributable to another significant issue: the **lack of publicly available datasets for Java code, especially comprehensive real-world datasets**. These datasets are instrumental for researchers in developing and evaluating vulnerability detection systems. Zhang *et al.* [84] have created a dataset for their ISVSF system, but it is not publicly available, limiting its utility for the broader research community. The current standard, the Juliet Test Suite for Java, is synthetic and has limited utility for training models that can generalise effectively to real-world situations [60].

Finally, as of the time of writing, **no DL-based GitHub Action for vulnerability detection exists**. This gap presents a significant opportunity for innovation and development in this space. Developing the first Deep Learning-based GitHub Action for Java vulnerability detection could have a substantial impact, allowing developers to shift left security, more effectively automate the vulnerability detection process, and further enhance the security of their Java-based software systems.

4.2 Existing opportunities

Given the aforementioned issues, the following opportunities can be identified:

1. **Creation of a curated vulnerability dataset from the Juliet Test Suite for Java**. This dataset must contain a reasonable number of samples and be representative of various vulnerabilities. Proper labelling and data preprocessing are required to ensure the dataset's quality and results' validity. Analysing VDET's data preprocessing pipeline is a good starting point to foresee possible improvements.

2. **Creation of a Java real-world evaluation test set** To assess the practical applicability of our models, it is crucial to create a test set that mirrors real-world conditions. This test set should be composed of Java projects from repositories like GitHub, with the corresponding buggy and fixed code, and representative of the vulnerabilities present in the training dataset.

3. **Development of a Transformer model capable of detecting vulnerabilities in Java source code**. Multi-label and multiclass classification configurations should be used to compare the results to the state-of-the-art models. Fine-tuning should be employed on a model containing knowledge about Java vulnerability-related code.

4. **Proposal of the first GitHub Action that leverages Deep Learning for vulnerability detection**. This tool must be user-friendly and provide meaningful vulnerability reports that can be understood by developers, even those without security expertise. Fast execution time is also required so developers' workflow is not hampered.

4.3 Proposed solution and research questions

Given the opportunities presented in the section, this dissertation aims to design and implement a robust, practical, and efficient Transformer-based GitHub action. This tool, named the Java Transformer-based Automated Scanner (J-TAS), is aimed at detecting vulnerabilities in Java source code and providing developers with clear insights into the vulnerabilities' root causes. The previous work on VDET's [46] development will serve as a starting point for developing the proposed solution.

The following research questions will guide the work developed in this dissertation:

- **RQ1** How do different dataset normalisation techniques influence the performance of the vulnerability detection models?

  Data pre-processing is a crucial step in developing accurate deep-learning models. First, we aim to investigate the impact of different normalisations of the Juliet dataset on the performance of a JavaBERT-based multi-label classification model. Additionally, we want to understand how the performance of the models varies when evaluated on each other's test set, identifying potential bias in the datasets.

- **RQ2** How does a multiclass architecture compare to a multi-label architecture in terms of performance?

  The problem of vulnerability detection can be framed as binary, multiclass, or multi-label classification, depending on goals and data available. We aim to investigate the impact of the multiclass and multi-label architectures on the performance of the vulnerability detection models and identify the most suitable approach in our context.

- **RQ3** How does the performance of a model pre-trained on Java buggy-related code compare to JavaBERT when applied to the same vulnerability detection task?

  We aim to investigate the impact of domain-specific pre-training on the performance of the vulnerability detection models. This involves comparing a model pre-trained on Java buggy code with JavaBERT regarding their effectiveness in detecting vulnerabilities.

- **RQ4** How can a DL model be integrated into a GitHub Action for real-time vulnerability analysis?

  Finally, we aim to explore the challenges of integrating the best-performing model into a GitHub Action and analyse the behaviour of our solution.

4.4 System requirements and evaluation

To be considered successful, the proposed solution must meet certain criteria. Table 4.1 lists the defined functional requirements in order of priority. These requirements describe the functionality of the tool.

Table 4.1: Functional requirements of the proposed solution

FR1	The tool must be able to detect various types of vulnerabilities in Java source code.
FR2	The tool must be able to scan multiple Java files in a repository.
FR3	The tool must be able to present the vulnerabilities reported in a clear and concise manner.
FR4	The tool should be able to give an explanation of the vulnerabilities detected.
FR5	The tool should be able to give feedback regarding errors.

The non-functional requirements identified are listed in Table 4.2. These requirements specify the tool's performance and usability.

Table 4.2: Non functional requirements of the proposed solution

NFR1	The tool must be able to execute in a reasonable amount of time.
NFR2	The tool must provide accurate results (accuracy > 80%, FNR < 10%, FPR < 10%).
NFR3	The tool should be easy to adapt to other programming languages.

The tool will be evaluated using the aforementioned requirements, and the results will be analyzed and compared to other existing deep learning-based vulnerability detection tools. Unmet requirements will be considered tool limitations and serve as the foundation for future work.

Chapter 5

J-TAS implementation

This chapter presents the development and implementation of J-TAS, our Deep-Learning-based tool for vulnerability detection in Java code. Section 5.1.1 starts by describing the steps taken to build the dataset used to train and evaluate our vulnerability detection models. This section details the creation of a synthetic dataset and elaborates on constructing a real-world test set consisting of vulnerable Java code. Moreover, it addresses our data-gathering procedure to train our masked language models (MLMs). Section 5.2 dives into the architecture and training process of both the MLMs and the vulnerability detection models, discussing the best approaches and hyperparameters used. Finally, Section 5.3 describes the development and implementation of J-TAS GitHub Action. This tool, developed as a practical application of our trained vulnerability detection model, is designed to automate the analysis of Java code repositories on GitHub, flagging potential vulnerabilities in the codebase.

5.1 Dataset creation

5.1.1 Dataset for vulnerability detection

A robust and high-quality dataset plays a pivotal role in the success of vulnerability detection methodologies, as it is a data-driven problem [10]. A well-curated dataset not only forms the foundation for training and evaluating detection models but also enables researchers and practitioners to gain a deeper understanding of the nature of vulnerabilities and devise appropriate countermeasures. Therefore, the process of gathering high-quality data and conducting meticulous preprocessing is paramount in the development of accurate and reliable vulnerability detection systems.

The Data Quality model defined in the standard ISO/IEC 25012 [33] describes the main data quality aspects that should be considered when evaluating a dataset. One of the main categories is Inherent Data Quality, which refers to the data's intrinsic characteristics that contribute to its overall quality. These characteristics include *accuracy*, *completeness*, *consistency*, *credibility* and *currentness*. Table 5.1 presents the definitions of these characteristics, as established in the standard. By adhering to these principles, we strive to create a dataset that can serve as a reliable foundation for training and evaluating our vulnerability detection models.

Table 5.1: ISO/IEC 25012 inherent data quality characteristics.

Characteristic	Definition
Accuracy	The degree to which the data has attributes that correctly represent the true value of the intended attribute of a concept or event in a specific context of use.
Completeness	The degree to which subject data associated with an entity has values for all expected attributes and related entity instances in a specific context of use.
Consistency	The degree to which data has attributes that are free from contradiction and are coherent with other data in a specific context of use.
Credibility	The degree to which data has attributes that are regarded as true and believable by users in a specific context of use.
Currentness	The degree to which data has attributes that are of the right age in a specific context of use.

In the context of Java vulnerability detection, it is worth noting that the availability of suitable datasets is limited, as mentioned in Section 3.2. We have selected the Juliet Test Suite for Java version 1.3[1] as the starting point to construct our dataset for the vulnerability detection models, describing each step in the following sections. This test suite comprises a vast collection of Java test cases, containing 28,882 files encompassing vulnerable and non-vulnerable examples across 112 different CWEs. While this dataset has certain limitations due to its synthetic nature, it remains the only publicly available Java test suite designed explicitly for accessing the capabilities of vulnerability detection tools.

5.1.1.1 VDET's dataset analysis

Initially, we analysed VDET's data processing pipeline and final dataset to investigate which findings are helpful and which improvements could be made.

VDET's dataset is constructed from the Juliet Test Suite for Java, and offers method-level granularity, with each method labelled as vulnerable or not, and also which CWE identifier is associated. It consists of 115600 total samples, with 34115 vulnerable methods and 81485 non-vulnerable. It is worth noting that the dataset is highly imbalanced, which can lead to biased results. In total, there are 21 different CWEs present. The dataset is split into three subsets: training, validation and test, with 80%, 10% and 10% of the samples, respectively.

The following issues were identified in VDET's dataset:

I.1. Insufficient normalisation: In the data normalisation process, only the method names were modified. Methods that contained the prefixes/suffixes "bad" and "good" were replaced with a neutral expression ("method"), while the remaining code was left unchanged. This approach was deemed insufficient as it failed to fully address potential biases in the dataset.

[1] https://samate.nist.gov/SARD/test-suites/111

In subsequent research, the authors investigated the presence of bias by calculating Pointwise Mutual Information (PMI) [61] scores to evaluate bias in the dataset. The analysis revealed that "bad" and "good" tokens still present in the code exhibited strong correlations with vulnerable and non-vulnerable methods, respectively. Additionally, tokens that contain the id of the CWE were found to be highly correlated with the CWE labels. These findings suggest that the model may rely on these tokens to classify the methods rather than effectively learning the underlying vulnerabilities. Unfortunately, the normalised dataset was not made available, preventing a more in-depth analysis.

I.2. Presence of duplicates: The dataset contains 47131 duplicated samples - rows with identical code, CWE-ID and vulnerability values. This represents a significant portion of the dataset, accounting for 40.8% of the total samples. Such a high percentage of duplicated data raises concerns regarding data uniqueness and can potentially lead to biased results.

Croft et al. [9] found that some software vulnerability datasets contain high data duplication rates, leading to data leakage, where models achieve inflated performance when evaluated using standard test setups. Removing these duplicates decreased the evaluation performance by up to 82%. Therefore, it is crucial to address data duplication issues to ensure accurate and reliable performance evaluations of vulnerability detection models.

I.3. Contradicting vulnerability labels: Upon removing the duplicated samples, our analysis revealed that 4988 methods with identical code and CWE-ID labels were assigned contradicting vulnerability labels. This means the same method was labelled as vulnerable in some instances and non-vulnerable in others.

Inconsistencies in data labelling can significantly impact the learning process, as the model may encounter conflicting information during training. Consequently, the model's ability to accurately distinguish between vulnerable and non-vulnerable methods may be compromised. Moreover, contradicting vulnerability labels can lead to challenges when evaluating the model's performance. Without a clear and consistent ground truth, it becomes difficult to ascertain the model's effectiveness in correctly identifying vulnerabilities.

I.4. Random train/validation/test split: The traditional split method randomly selects samples for each subset according to the desired split ratio. However, this random selection can lead to an imbalanced distribution of the target variable across the subsets. In the context of vulnerability detection, this can result in a disproportionate number of vulnerable methods and the corresponding CWE-ID in each subset. A stratified split, on the other hand, would ensure equal proportions of the target variable in all splits, which means that the model is trained and evaluated on the same distribution of labels.

In conclusion, this analysis highlighted several issues that need to be addressed in order to create a robust and reliable dataset for vulnerability detection from the Juliet Test Suite for Java. As our research progresses, these findings will serve as valuable guidelines in the following data preprocessing stage, described in detail in the following sections. By incorporating the necessary adjustments and improvements based on the lessons learned from VDET's dataset, we aim to create a dataset that overcomes these challenges and produces more reliable and meaningful results in our vulnerability detection model.

5.1.1.2 Data parsing and code labelling

The labelling process involves assigning appropriate class labels to instances within the dataset, and the granularity of the labels is a crucial factor in the success of the detection model. Section 3.1 presented several systems with different granularity levels for vulnerability detection, and we decided to use method-level granularity, as done in VDET's work [46]. Assigning labels at the method level offers a reasonable trade-off between granularity and manageability. It allows us to capture the presence of vulnerabilities within individual methods, which are crucial units of code in software development, and offer a much more granular approach than file-level labelling. Moreover, method-level labelling simplifies the annotation process by reducing the complexity and time required for manual annotation compared to more granular approaches like line-level labelling.

To process the Juliet Test Suite for Java and achieve the desired granularity level, we have developed a script in Python that uses the *javalang* library[2] to parse and extract all the methods from the test cases. The Juliet Test Suite is organised into different packages, each containing test cases for a specific CWE. For example, test cases for the CWE-15 are located inside the package CWE15_External_Control_of_System_or_Configuration_Setting, in several java files which start with the CWE id and its shortened CWE entry name (e.g. CWE15_External_Control_of_System_or_Configuration_Setting__Environment_01.java). The test cases contain "bad" (vulnerable) and "good" (non-vulnerable) methods, which are named accordingly.

By leveraging the information provided in the Juliet Test Suite documentation, including the tests' naming convention, test design, and regex pattern for method names, our script automatically labels the extracted methods as vulnerable or non-vulnerable and with the corresponding CWE-ID based on the method's name and the package it is located, respectively. All the data is stored in a CSV file, which ends up with 207892 samples. The file contains the following columns: *id* (an incremental identifier for each row), *code* (the methods code text), *CWE* (CWE-ID associated to that method), *isVulnerable* (a boolean, indicating whether the method is vulnerable or not), *filename* (the full file path, including package), *method_name* (the method's name), *startline* and *endline* (method's start and end line, respectively).

5.1.1.3 Data cleaning and normalisation

The data cleaning process is a crucial step in the data preprocessing pipeline, as it ensures that the data is in a suitable format for the model to learn from. The cleaning process involves removing any unnecessary information and normalising the data to ensure consistency.

[2] https://github.com/c2nes/javalang

First, an initial filtering process is executed to retain only methods that are relevant to vulnerability analysis, adhering to the guidelines provided in the Juliet documentation. Consequently, only methods whose names contain the words "bad" or "good" were preserved for further analysis. The primary good methods, named strictly "good", were removed from the dataset as they do not contribute to the learning process of vulnerability detection models. These methods are only used to call secondary good methods, which are the ones that contain the actual code to be learned. This filtering process resulted in the removal of 61484 samples from the dataset. Figure 5.1 illustrates the distribution of methods post this filtering stage.

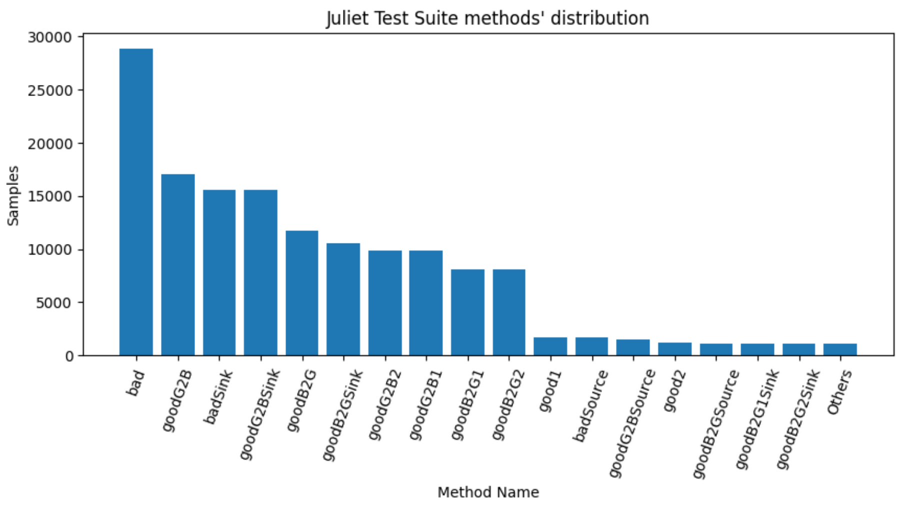


Figure 5.1: Juliet Test Suite methods' distribution. Column 'Others' groups 26 methods with less than 1000 samples.

Upon completing the initial filtering step, we conducted a thorough examination of data integrity and consistency. Specifically, we checked for missing values and duplicates within the dataset. The dataset has no missing values, so we guarantee *completeness*. It is also free from contradicting labels, ensuring *consistency*. However, we identified 52309 duplicate samples - entries that shared the same values in the columns *code*, *CWE*, and *isVulnerable*. In order to ensure data accuracy and address issue I.2., these duplicate entries were removed from the dataset.

The data normalisation process began by removing all the comments from the code, given they neither influence code execution nor contribute to vulnerabilities. Additionally, all whitespace characters were replaced with a single space, reducing the overall length of the code. Afterwards, two different normalisation pipelines were executed, to address the I.1. issue. This resulted in the creation of two normalised datasets, ND1 and ND2. The specific techniques utilised during the code normalisation process are detailed in Table 5.2.

Table 5.2: ND1 and ND2 normalisation steps per code entity.

Entity	ND1	ND2
Strings	Entire String contents were replaced with the word "string".	
Methods	Vulnerability-related method names in declarations and calls were replaced by a neutral expression, *method_X*.	Vulnerabilty-related method names were mapped and replaced by random strings.
Classes	Not normalised.	Test cases' classes were mapped and replaced by random strings.
Variables and Objects	Variable and Object names that contained the tokens "bad", "good" or "CWE" were replaced with *var_X* and *Obj_X*, respectively.	

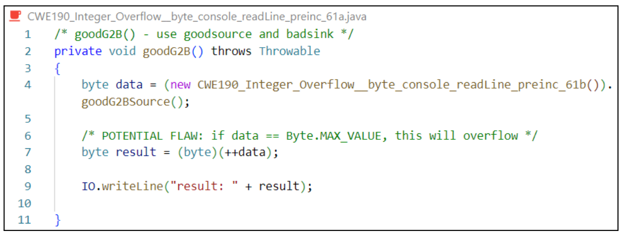
ND1 analysis

The ND1 normalisation pipeline begins by addressing text within strings since we observed that certain test cases' strings contained the id or name of the CWE being tested. The second step focus on the normalisation of vulnerability-related method names. Unlike VDET, which only replaced the method names in the method declaration, we also replaced the method names in the method calls that occur within the code. Finally, we normalised variable and object names that contain any vulnerability-related tokens. The normalisation technique for these names is similar to that for method names.

The numbering schema used to create neutral names assigns a number to each unique vulnerable method name encountered within a sample. This numbering scheme is reset when processing another sample to mitigate bias. If a direct mapping of method names to their neutral counterparts, such as assigning *bad* to *method_1* and *good* to *method_2* were employed, the inherent bias would persist.

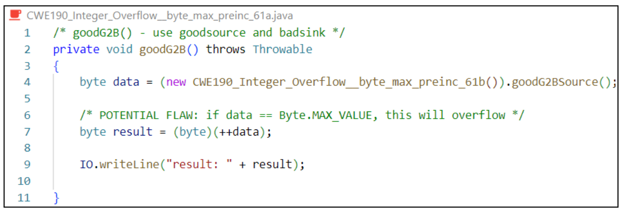
While these normalisation steps eliminate the presence of "CWE", "bad" and "good" tokens in the code and thereby remove possible bias, they inadvertently lead to duplicate rows and contradicting labels within the dataset. Upon closer analysis of these samples, we sought to uncover the root causes of these issues.

The duplication of rows can be attributed to the presence of methods with similar code but different object names and method calls, as depicted in Figure 5.2. Both *goodG2B* methods possess identical code structures but call *goodG2BSource* methods from distinct classes. After normalising the methods and variable names, both rows are identical.



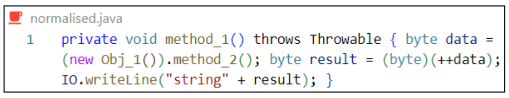
CWE190_Integer_Overflow__byte_console_readLine_preinc_61a.java

```java
/* goodG2B() - use goodsource and badsink */
private void goodG2B() throws Throwable
{
    byte data = (new CWE190_Integer_Overflow__byte_console_readLine_preinc_61b()).
    goodG2BSource();

    /* POTENTIAL FLAW: if data == Byte.MAX_VALUE, this will overflow */
    byte result = (byte)(++data);

    IO.writeLine("result: " + result);

}
```



CWE190_Integer_Overflow__byte_max_preinc_61a.java

```java
/* goodG2B() - use goodsource and badsink */
private void goodG2B() throws Throwable
{
    byte data = (new CWE190_Integer_Overflow__byte_max_preinc_61b()).goodG2BSource();

    /* POTENTIAL FLAW: if data == Byte.MAX_VALUE, this will overflow */
    byte result = (byte)(++data);

    IO.writeLine("result: " + result);

}
```



normalised.java

```java
private void method_1() throws Throwable { byte data =
(new Obj_1()).method_2(); byte result = (byte)(++data);
IO.writeLine("string" + result); }
```

Figure 5.2: Example of two methods which have the same normalised code. The two figures on the left show two *goodG2B* methods from different classes, and the figure on the right shows the normalised code of both.

The contradicting labels arise due to the normalisation of scenarios involving *good-source/bad-sink* and *bad-source/good-sink* methods. In the example illustrated in Figure 5.3, the normalisation process renders the code of the *badSource* method identical to that of *goodB2GSource*, and the same applies to *bad* and *goodG2B* methods.

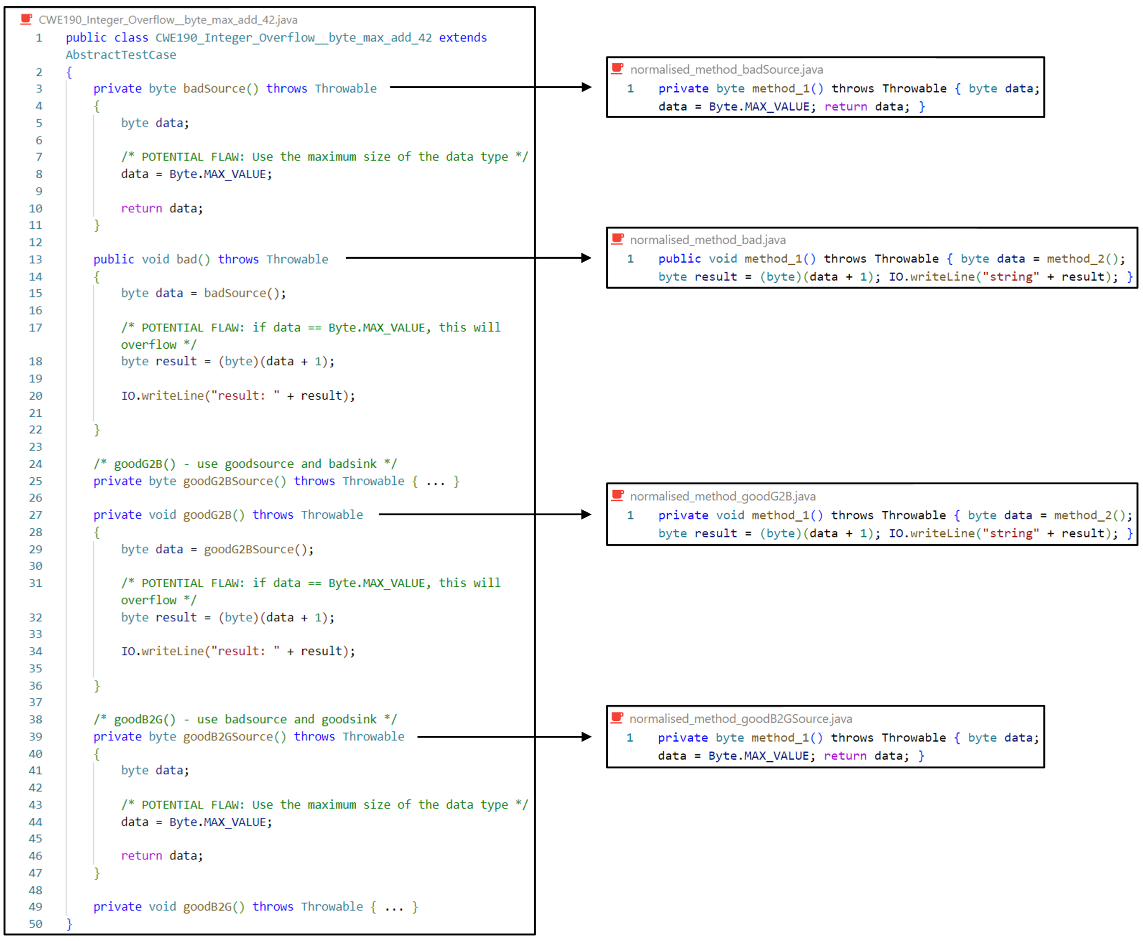
We eliminated the duplicated samples by retaining only the first occurrence, which removed 37213 extra samples from our dataset. However, the contradicting labels were not rectified during the ND1 normalisation process. These findings underscore the need for further enhancements to the normalisation procedure.

ND2 analysis

In this second normalisation pipeline, we sought to address the issues identified in the previous pipeline. The string, variable and object names normalisation steps are identical to those of ND1. The key difference lies in normalising method names and introducing a new normalisation step for class names.

As evidenced in Figure 5.1, the Juliet Test Suite for Java is mainly composed of *good-source/bad-sink* and *bad-source/good-sink* scenarios. The ND1 method normalisation step failed to capture the semantic relationships between the methods and classes involved in these scenarios, as different methods were frequently mapped to the same neutral name.

ND2 pipeline introduces a mapping process that assigns vulnerability-related method names and test class names to random neutral counterparts. Given the dataset size, these counterparts are unique strings of lengths 3 and 4, respectively, the minimum length required to generate unique names. It is important to note that the normalisation of method names depends on the test class, resulting in the same method name being mapped to different random strings based on the test class to which it belongs. For example, the *goodG2B* method from the


CWE190_Integer_Overflow__byte_max_add_42.java
normalised_method_badSource.java
normalised_method_bad.java
normalised_method_goodG2B.java
normalised_method_goodB2GSource.java

```
public class CWE190_Integer_Overflow__byte_max_add_42 extends
AbstractTestCase
{
    private byte badSource() throws Throwable
    {
        byte data;

        /* POTENTIAL FLAW: Use the maximum size of the data type */
        data = Byte.MAX_VALUE;

        return data;
    }

    public void bad() throws Throwable
    {
        byte data = badSource();

        /* POTENTIAL FLAW: if data == Byte.MAX_VALUE, this will
        overflow */
        byte result = (byte)(data + 1);

        IO.writeLine("result: " + result);

    }

    /* goodG2B() - use goodsource and badsink */
    private byte goodG2BSource() throws Throwable { ... }

    private void goodG2B() throws Throwable
    {
        byte data = goodG2BSource();

        /* POTENTIAL FLAW: if data == Byte.MAX_VALUE, this will
        overflow */
        byte result = (byte)(data + 1);

        IO.writeLine("result: " + result);

    }

    /* goodB2G() - use badsource and goodsink */
    private byte goodB2GSource() throws Throwable
    {
        byte data;

        /* POTENTIAL FLAW: Use the maximum size of the data type */
        data = Byte.MAX_VALUE;

        return data;
    }

    private void goodB2G() throws Throwable { ... }
}
```

```
private byte method_1() throws Throwable { byte data;
data = Byte.MAX_VALUE; return data; }
```

```
public void method_1() throws Throwable { byte data = method_2();
byte result = (byte)(data + 1); IO.writeLine("string" + result); }
```

```
private void method_1() throws Throwable { byte data = method_2();
byte result = (byte)(data + 1); IO.writeLine("string" + result); }
```

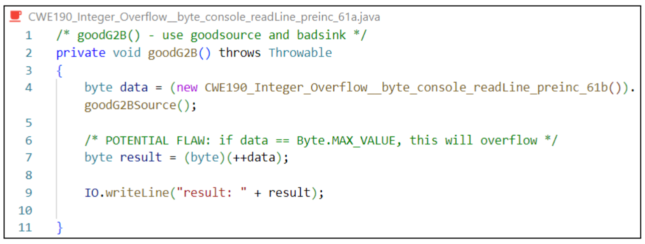
```
private byte method_1() throws Throwable { byte data;
data = Byte.MAX_VALUE; return data; }
```

Figure 5.3: Example of contradicting normalised methods.

CWE190_Integer_Overflow__byte_console_readLine_preinc_61a class is mapped to *hapv* whereas the *goodG2B* method from the *CWE190_Integer_Overflow__byte_max_preinc_61a* class is mapped to *hHVU*.

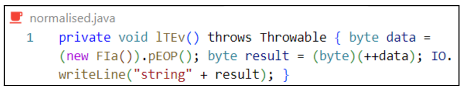
The described mapping process allowed us to replace susceptible method and class names with fixed neutral names. This is done in their declarations and when they are referenced within other methods, whether they are from the same or a different class. As a result, both the structural and semantic relationships intrinsic to the code are preserved.

All the normalisation steps of the ND2 pipeline effectively eliminate the presence of "CWE", "bad" and "good" tokens in the code. Unlike ND1 normalisation, no duplicate rows or contradicting labels were introduced. Thus ND2 contains more samples than ND1 and captures the dependency between methods in data flow tests. Figure 5.4 presents the new normalisation of the same methods illustrated in Figure 5.2



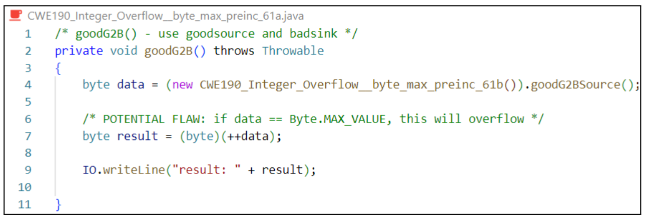
CWE190_Integer_Overflow__byte_console_readLine_preinc_61a.java

```java
/* goodG2B() - use goodsource and badsink */
private void goodG2B() throws Throwable
{
    byte data = (new CWE190_Integer_Overflow__byte_console_readLine_preinc_61b()).
    goodG2BSource();

    /* POTENTIAL FLAW: if data == Byte.MAX_VALUE, this will overflow */
    byte result = (byte)(++data);

    IO.writeLine("result: " + result);

}
```



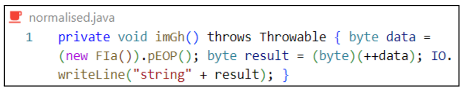
normalised.java

```java
private void lTEv() throws Throwable { byte data =
(new FIa()).pEOP(); byte result = (byte)(++data); IO.
writeLine("string" + result); }
```



CWE190_Integer_Overflow__byte_max_preinc_61a.java

```java
/* goodG2B() - use goodsource and badsink */
private void goodG2B() throws Throwable
{
    byte data = (new CWE190_Integer_Overflow__byte_max_preinc_61b()).goodG2BSource();

    /* POTENTIAL FLAW: if data == Byte.MAX_VALUE, this will overflow */
    byte result = (byte)(++data);

    IO.writeLine("result: " + result);

}
```



normalised.java

```java
private void imGh() throws Throwable { byte data =
(new FIa()).pEOP(); byte result = (byte)(++data); IO.
writeLine("string" + result); }
```

Figure 5.4: Example of two methods which have the same normalised code in ND1 but not in ND2. Figures on the right show the normalised code of the respective method on the left figure.

5.1.1.4 Data filtering

After applying the normalisation pipelines ND1 and ND2, we proceed to address two significant challenges in our dataset: the presence of long sequences and the issue of class imbalance. Despite the already reduced dataset size, these issues must be addressed to ensure the model's learning process is not compromised. Both Long Sequence Removal (LSR) and Non-significant data removal (NSDR) steps followed VDET's approach.

Long Sequence Removal

In order to mitigate the impact of long sequences, the LSR stage involved removing methods that exceed the maximum token limit of 512 imposed by the BERT models. By eliminating these lengthy sequences, we ensure compatibility with the chosen model architecture and prevent any potential loss of information due to token truncation. This step is crucial to ensure that the models can learn from the data without limitations. Figure 5.5 illustrates each datasets' code tokens count distribution before and after the LSR step.

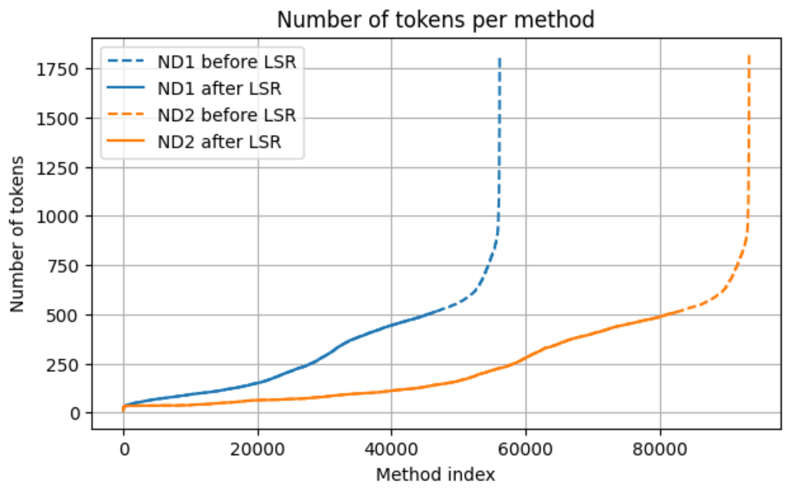


Figure 5.5: ND1 and ND2 code tokens count distribution before and after LSR step.

Non-significant data removal

Succeeding the LSR step, the NSDR stage was performed. Figure 5.6 illustrates the imbalance problem in the datasets before NSDR. To determine the significance of each CWE label, we calculated the mean number of samples per CWE. This allowed us to establish a threshold value representing the minimum number of required examples per CWE (400 for ND1, and 750 for ND2). Any CWEs with fewer samples than this stipulated threshold are deemed non-significant and subsequently removed from the dataset. By performing this step, we aim to achieve a more balanced distribution of samples across the different CWE labels. This ensures that each CWE category has sufficient examples for effective model training and evaluation. Despite the remaining CWE samples still presenting imbalance issues, it is less severe than the original datasets.

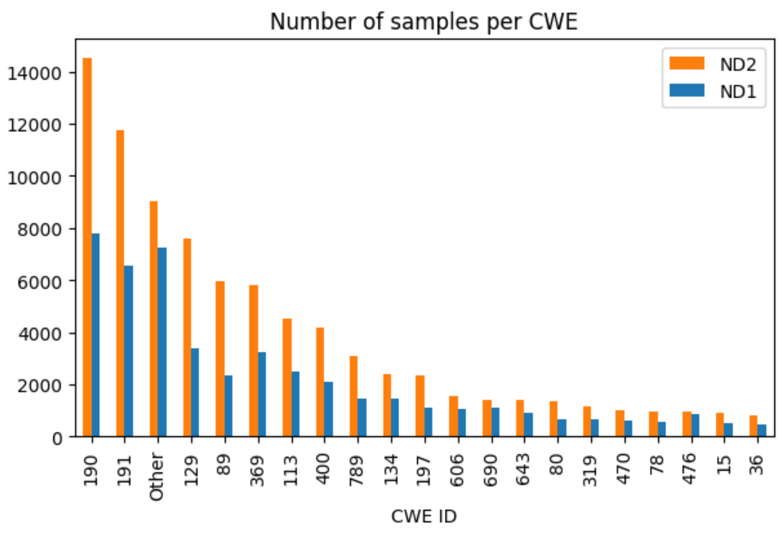


Figure 5.6: Distribution of samples per CWE in ND1 and ND2 before NSDR. Column "Other" aggregates 90 CWEs.

Table 5.3 presents a comprehensive summary of our datasets after completing all preprocessing stages. For comparison, we have also included VDET's publicly available dataset. It is important to note that VDET's distribution of vulnerable samples differs from the one presented by Mamede *et al.* [46] in their work.

Table 5.3: Overview of VDET and our preprocessed datasets.

Dataset	Total samples	Vulnerable samples	Non-vulnerable samples	Vulnerability ratio	Total CWEs
VDET	115600	34115	81485	0.42	21[a]
ND1	41216	17567	23649	0.43	24[b]
ND2	74426	27176	47250	0.37	21[c]

[a] VDET supported CWEs: 15, 23, 36, 78, 80, 89, 90, 113, 129, 134, 190, 191, 197, 319, 369, 400, 470, 606, 643, 690, 789
[b] ND1 supported CWEs: 15, 36, 78, 80, 81, 83, 89, 90, 113, 129, 134, 190, 191, 197, 319, 369, 400, 470, 476, 563, 606, 643, 690, 789
[c] ND2 supported CWEs: 15, 36, 78, 80, 89, 90, 113, 129, 134, 190, 191, 197, 319, 369, 400, 470, 476, 606, 643, 690, 789

5.1.1.5 Data partitioning

In the final stage of the dataset creation pipeline, we split the datasets into distinct training, validation, and test sets. This step allows us to train our models, fine-tune their hyperparameters, and evaluate their performance on unseen data. Before splitting, we created the columns with the target labels for our models.

In the multi-label context, we create a single label column named *one-hot*. This column is the result of applying one-hot encoding to the combined *CWE* and *isVulnerable* columns (for instance, a value pair like [89, 'True']), transforming the categorical variables into a format that could be provided to machine learning. Consequently, we obtain a one-hot vector, whose length is dictated by the total count of CWEs in the dataset, incremented by two, accounting for the 'True' and 'False' values in the *isVulnerable* column.

For the multiclass models, we introduced the *encoded_CWE* column, which contains the value *0* if the method is not vulnerable, and the *encoded CWE-ID* otherwise. The output layer of the model contains as many neurons as there are classes, with each neuron representing the probability of the instance belonging to that particular class. Therefore, it is essential to encode the labels in a format that the model can process. This encoding is achieved by mapping each CWE-ID to a unique integer value within the $[1, NC]$ range, where *NC* denotes the number of unique CWE-IDs in the dataset.

Initially, we apply an 80/20 train/test split to each dataset. This ratio strikes a balance between providing substantial data for training our models while reserving a reasonable portion for evaluation, and is empirically the best division [25]. We further split the test sets into two equal subsets, resulting in a 50/50 validation/test split. This additional split is essential as it allows us to have a dedicated validation set for fine-tuning our models and selecting the best-performing ones.

We employ the *StratifiedShuffleSplit*[3] class from the *scikit-learn* library to perform these splits

[3] https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.StratifiedShuffleSplit.html

and address VDET's I.4. issue. This class enables us to split our data while preserving the stratification of the target variable, *one-hot*, ensuring that the proportions of different labels are maintained in each subset. In the case of the ND1 dataset, we straightforwardly perform the splits based on the *one-hot* column. However, in the ND2 dataset, we take an additional step to accommodate the splitting of data flow tests. For this purpose, we introduce a new column called *pair_hash*, which assigns a unique identifier to all the methods belonging to the same test case. The split is then performed based on these unique *pair_hash* values, while stratification is ensured using the *encoded_CWE* column as the target variable. Methods from the same test case are grouped within the same split, enabling a more coherent evaluation process.

5.1.2 Creating a real-world evaluation dataset

The datasets produced in the preceding section are designed to train and evaluate our models. However, they do not represent real-world vulnerability scenarios, as they were built from synthetic test cases. In order to assess the real-world capabilities of our models, it is crucial to test them on non-synthetic data. To this end, we have created a real-world evaluation set: a collection of vulnerable real-world methods and their patched counterparts. This set was constructed using the CVEfixes tool [4], whose source code is available on their GitHub repository[4].

The CVEfixes tool is a resource designed to automate the collection of vulnerabilities and their corresponding fixes from open-source software. It operates by scanning all published JSON vulnerability feeds from the National Vulnerability Database[5], from 2002 up to the most recent one on the date of collection, and retrieving pertinent information such as CVE-ID, CWE-ID and associated reference links. All open-source projects that were reported in CVE records in the NVD within this timeframe and had publicly available git repositories are fetched and used to build the vulnerability dataset. The most recent CVEfixes vulnerability dataset dump covers all published CVEs up to 27 August 2022, with a total of 7637 CVEs in 209 different CWEs types, and containing the source code before and after the fix of 29309 files and 98250 functions [50].

We adapted the CVEfixes source code to scrape only Java-related repositories. This resulted in a real-world Java vulnerability dataset with 5170 total methods, covering published CVEs up to 15 April 2023, totalling 315 CVEs across 73 CWEs types. In order to maintain consistency with previous works and to enable an effective comparison of real-world model performance, we curated our dataset further. Our filtering criteria focused on retaining only those CWE-IDs present in VDET, ND1, and ND2 datasets. To reduce noise and improve the relevance of our data, we excluded methods associated with tests, as they do not directly encapsulate the vulnerability and its corresponding fix. Table 5.4 shows the final dataset statistics.

It is important to note that this dataset is not ideal as it contains few samples and does not cover all the CWEs supported by our models' training datasets. While VDET, ND1, and ND2 covered more than 20 CWEs, this test set only shares 5 CWEs with them. This presents a limitation regarding the range of vulnerabilities that can be tested.

[4]https://github.com/secureIT-project/CVEfixes
[5]https://nvd.nist.gov/vuln/data-feeds

Table 5.4: Distribution of samples per CWE in the real-world test set

CWE-ID	Total samples	Vulnerable samples
78	20	9
89	515	222
90	4	1
113	6	2
400	64	23

Despite these limitations, our dataset remains a significant resource. Given the existing tools and resources, it is the most comprehensive dataset of real-world Java vulnerabilities we could create, offering an opportunity to assess and compare our models' real-world capabilities.

5.1.3 Java buggy code corpus for domain adaptation

To train our custom MLM model, which can subsequently be fine-tuned to generate vulnerability detection models, we required a dataset encompassing both vulnerable and non-vulnerable methods. To this end, we curated two distinct datasets from the Learning-Fixes [73] data, available on their website[6]. This data contain pairs of buggy Java methods and their respective fixes, extracted from 787178 bug-fixing commits mined from the GitHub Archive [32]. The selection of this data source was motivated by its rich collection of real-world Java code with buggy methods and corresponding fixes, which provides a valuable foundation for training a model to recognize and understand the nuances of vulnerable code structure and its fixes.

The first dataset was created by merging Learning-Fixes' BFP_{small} and BFP_{medium} datasets, which we named $BFP_{combined}$, and contains 123805 bug-fix method pairs (BFPs). Instead of using the pre-defined splits provided, which contain the abstracted version of the code, a custom 80/20 train/validation split of the pairs was created using the original source code of the methods. This decision was driven by the desire to train the model on the raw, unabstracted code, thereby enabling it to learn the vocabulary of both vulnerable and fixed Java code.

For the second dataset, we used the Extracted Bug-Fix Pairs data, which contains all the BFPs extracted from the bug-fixing commits. Out of the 2.3 million BFPs, we retained only those associated with single file and single method commits, with a maximum of 512 tokens. This was done to manage the substantial size of the original dataset, making it more manageable and efficient for training purposes. The resulting dataset, named BFP_{single} comprises 149227 BFPs, and was subjected to the same 80/20 train/validation split as the previous dataset.

5.2 Model development and training

This section explains the development of our model's custom architecture and the training process. Initially, we leveraged the architecture of VDET's model as a baseline, creating two new

[6]https://sites.google.com/view/learning-fixes/data

multi-label models trained on the ND1 and ND2 datasets. Afterwards, we opted for a multiclass approach, a strategy widely used in other DL vulnerability detection, as highlighted in section 3.1.

In the final phase of our experiments, we fine-tuned two masked language models, from BERT, on Java vulnerable code. These pre-trained models were used for transfer learning on the ND1 and ND2 datasets, fine-tuning both multi-label and multiclass models for vulnerability detection.

All fine-tuning experiments were conducted using the free version of Google Colab, which provides access to GPUs, depending on availability. By leveraging CUDA-enabled GPUs in the training process, we accelerated the fine-tuning phase, reducing the overall training time and increasing the efficiency of our experiments.

5.2.1 Model architecture

The notebook containing the code for VDET's model architecture, training and evaluation is promptly available in VDET's GitHub repository[7]. We used this code as the starting point to develop our models, making the necessary changes to adapt it to our datasets and experiments. Our notebook, containing the code for the models' architecture, training and evaluation, is presented in Listing A.1 in the Appendix.

5.2.1.1 Baseline multi-label model

The multi-label vulnerability classifier model was developed using the same architecture as VDET's model, except for the output layer, which has to be modified to accommodate the number of labels in each of our new datasets. The model's architecture, implemented using PyTorch, is presented in Listing 5.1.

To define our customized architecture, we first created a new class, *VulnerabilityClassifier*, which inherits from the *torch.nn.Module*[8] class. In the constructor (*__init__* method), the JavaBERT model checkpoint is loaded, as it is the model we will finetune. A Dropout layer is added to the model, with a probability of 0.1, to prevent overfitting. Finally, the output Linear layer is defined, with the number of labels corresponding to the number of labels in the dataset. This layer is also four times the size of a single BERT's hidden layer (768), as we will use the concatenation of the last four hidden layers to obtain the final output. We chose to use the concatenation of the last four hidden layers as it was the best-performing strategy in BERT's and VDET's experiments.

```python
N_CLASSES = 21 # Number of labels in the dataset
model_checkpoint = 'CAUKiel/JavaBERT' # Model checkpoint from Hugging Face

class VulnerabilityClassifier(torch.nn.Module):
    DROPOUT_PROB = 0.1

    def __init__(self):
```

[7] https://github.com/TQRG/VDET-for-Java
[8] https://pytorch.org/docs/stable/generated/torch.nn.Module.html

```python
        super(VulnerabilityClassifier, self).__init__()
        self.model = transformers.AutoModel.from_pretrained(model_checkpoint)
        self.dropout = torch.nn.Dropout(self.DROPOUT_PROB)
        self.linear = torch.nn.Linear(768 * 4, N_CLASSES)
        self.step_scheduler_after = 'batch'


    def forward(self, ids, mask):
        cls_hs = torch.stack(self.model(ids, attention_mask=mask)["hidden_states"])

        cls_4hs = torch.cat((cls_hs[-1],
                             cls_hs[-2],
                             cls_hs[-3],
                             cls_hs[-4]), -1)[:, 0]

        output_dropout = self.dropout(cls_4hs)
        return self.linear(output_dropout)
```

Listing 5.1: Baseline model architecture.

In the *forward* method, we define the model's logic. It accepts the input ids and attentions masks of the code samples and processes them through the model. Afterwards, the output of the last four hidden layers is concatenated and passed through the Dropout layer. Finally, the output of the Dropout layer is passed through the Linear layer, obtaining the final output of the model.

In order to obtain the final predictions, we have to apply an activation function to the model output. We picked the Sigmoid function since we are dealing with a multi-label classification issue. It returns a number between 0 and 1 for each label, representing the likelihood of the sample belonging to that label. The final predictions are obtained using a 0.5 threshold to the Sigmoid function output, meaning that if a label's probability is greater than 50%, the sample is classified as belonging to that label.

5.2.1.2 Transitioning to a multiclass architecture

In multi-label classification, each instance can be assigned to one or more classes, meaning the classes are not mutually exclusive. Mamede *et al.* [46] reasoning for adopting a multi-label approach was that it enables the model to predict multiple CWEs for a single method. However, our datasets do not reflect this scenario, as each method is exclusively linked to a single CWE-ID. Furthermore, the inherent structure of the labels in the multi-label datasets allows the model to generate incompatible or contradictory predictions. For instance, the model may concurrently predict both "True" (vulnerable) and "False" (not vulnerable) for the same method, or it may predict only a CWE label without the associated vulnerability status label, or vice-versa. These are all scenarios we end up confirming in our experiments.

Given these issues associated with the employed multi-label approach, we decided to switch to a multiclass architecture. This approach suits our datasets better and provides more precise and unambiguous predictions, as the model is forced to predict a single label for each method.

The architecture of the multiclass model is identical to the multi-label model, with the only necessary change being the adjustment of the output layer to accommodate the number of classes in our dataset. Specifically, the output layer has to be a Linear layer with the number of classes corresponding to the number of unique values in the *encoded_CWE* column, as it is the target variable.

To generate the final predictions, we simply extract the index of the maximum value in the model output list, which corresponds to the predicted class. While a Softmax function could be applied to the model output to obtain the probabilities of each class, this step is unnecessary, particularly during the training phase. The training process focuses on identifying the correct classes rather than analyzing the probabilities associated with each one.

5.2.2 Loss functions

Loss functions, also known as cost functions, play a key role in neural network training. They measure the error between the predicted and the actual values, providing a metric to optimise during training. The goal of the training process is to minimise the value of the loss function to achieve the best results possible.

Cross-Entropy loss, a popular loss function in neural network training, has a unique characteristic that makes it particularly effective: it penalises models that are overconfident about their predictions. The greater the difference between the predicted and the actual value, the greater the loss [58]. This is useful in our vulnerability detection context, as it is critical to minimise the number of false positives and false negatives.

In the multi-label scenario, we chose the Binary Cross-Entropy (BCE) loss function, following the same approach as VDET. The Binary Cross-Entropy (BCE) with Logits Loss function, as implemented in PyTorch[9], combines a Sigmoid layer and the BCE loss in one single class, making it more numerically stable. Regarding the multiclass model, we opted for the Cross-Entropy loss function, which combines a LogSoftmax layer and the NLLLoss (Negative Log Likelihood Loss) in one single class[10]. The LogSoftmax activation is used to normalise the output of the model, obtaining a probability distribution over the classes. The NLLLoss function is then used to calculate the loss between the predicted and the actual class.

5.2.3 Optimizers

In the training of deep learning models, the optimizer is responsible for updating the model's parameters based on the gradients of the loss function. The optimizer aims to minimize the loss function, thereby improving the model's performance.

[9]https://pytorch.org/docs/stable/generated/torch.nn.BCEWithLogitsLoss.html
[10]https://pytorch.org/docs/stable/generated/torch.nn.CrossEntropyLoss.html

The Hugging Face library provides support for the AdaFactor [63] and AdamW [45] optimizers, both of which are extensions of the original Adam optimizer [37]. Despite Adafactor being more memory-efficient, we opted for AdamW, as its weight decay regularization can lead to better generalization and faster convergence.

The learning rate (LR) is a hyperparameter that controls the step size of the optimizer during training. A higher LR can speed up the training process but can also lead to the model overshooting the optimal solution. On the other hand, a lower LR makes smaller updates to the model's weights, meaning that it will take more iterations to converge, but often leads to a more accurate and reliable model, as it reduces the risk of overshooting and allows the model to fine-tune its parameters for optimal performance. In our experiments, we used a learning rate of $5e^{-5}$ (the same provided in VDET's notebook), lower than the default value of $1e^{-3}$.

Additionally, a linear LR scheduler was employed to adjust the learning rate during training. This scheduler increases the LR linearly from 0 to the provided value over a warmup period and then decreases it linearly to 0 over the remaining training steps. The warmup period allows the model to start learning slowly, reducing the risk of significant errors early in the training process, and the gradual decay allows the model to fine-tune its parameters for optimal performance.

5.2.4 Training the vulnerability classifier models

Before feeding the data to the model, we have to tokenize it, converting the code samples into a format the model can understand, and group it into batches so data can be processed in parallel. The tokenizer, provided by the model checkpoint we are using, is responsible for tokenizing the code samples and ensuring they have the desired length by either padding or truncating them. In the case of BERT models, the maximum length of the input is 512 tokens, so [PAD] tokens are added to the end of the code samples that are shorter than 512 tokens, and the longer ones are truncated. Since we preprocessed the datasets to ensure that all code samples have a maximum of 512 tokens, no truncation should be necessary.

Afterwards, the data is grouped into batches of size 12, the maximum we could fit into the GPU memory. In order to reduce the number of tokens and subsequentially increase training speed, a smart batching strategy from Chris McCormick tutorial [47] is applied. The code samples are sorted by length and then grouped into batches with similar lengths, reducing the number of [PAD] tokens and, thus, reducing the total number of tokens the model has to process.

With the training and validation data ready, we can start the training cycle, which is carried out for 10 epochs. This process is divided into two phases per epoch, training and validation, which are implemented with *train_fn* and *eval_fn* functions present in Listing A.1 in the Appendix. The training phase involves settings the model to train mode, loading the inputs onto the GPU from acceleration, and passing the inputs through the model. The loss is then calculated, and the gradients are computed and back-propagated through the model. Finally, the optimizer is used to update the model's parameters. The validation phase is similar, except that the model is set to evaluation mode, the gradients are not computed and back-propagated, and the optimizer is not updated. Each epoch can take up to an hour to complete depending on the dataset used.

A checkpoint strategy is used to ensure efficient model development. At the end of each epoch, we serialize and store the current states of the model, optimizer and scheduler. This serialized object contains all the necessary information to load the model and continue the training from that epoch onwards. Additionally, we save the training and validation loss and accuracy for each epoch. This strategy not only guarantees that we have a record of the model's progress but also provides flexibility in terms of experiment management. It allows us to stop the training process at any time, resume it later, and compare the performance of different models by loading their best-performing checkpoints.

5.2.5 Custom masked language modelling with Java buggy code

Masked Language Modelling (MLM) is a training technique used NLP where a model is trained to predict certain masked tokens in a the given input. This approach, popularized by models such as BERT, allows the model to learn a rich understanding of the language structure and semantics, as it needs to understand the context around the masked token to make accurate predictions. Vulnerabilities also revolve around the context of the code, as the presence of certain tokens in a method can be a strong indicator of the presence of a vulnerability. Therefore, we hypothesize that a model trained to predict masked tokens in buggy code will be able to learn a rich representation of the context around vulnerabilities, and thus be able to detect them.

Inspired by the work of JavaBERT [12], we decided to develop our own MLM model for Java, fine-tuning BERT on domain-specific data, in our case, Java code with vulnerabilities.

The fine-tuning process was performed using the Hugging Face Transformers library, following the steps described in the "Fine-tuning a masked language model" guide[15]. We chose to fine-tune the *bert-base-cased* model since Java is a case-sensitive language, and it was the best performing model in JavaBERT experiments [12].

We trained two distinct instances, each corresponding to one of the datasets described in section 5.1.3, resulting in the creation of the $BFP_{combined}$ and BFP_{single} masked language models. Prior to the training phase, the data underwent tokenization, converting the code into tokens the model can interpret and learn from. These tokens were then exposed to random masking, where selected tokens were replaced with a [MASK] token. Using the *DataCollatorForLanguageModelling* class[11], this was executed with a probability of 15% (default value). The masking technique is a fundamental aspect of MLM, as it compels the model to predict the masked tokens based on their context, thereby promoting a deeper understanding of the language structure and semantics.

The training procedure was executed over five epochs using HF's Trainer API, and the notebook containing the code for this process and the training logs are presented in Appendix A.2. The most pertinent training arguments and hyperparameters are summarised in Table 5.5. It is noteworthy to mention that we employed mixed precision training, denoted by the *fp16* flag, a technique that leverages half-precision floating point numbers (16-bit) as opposed to the conventional 32-bit ones for the representation of the model's weights and activations. This technique

[11]https://huggingface.co/docs/transformers/main_classes/data_collator#transformers.DataCollatorForLanguageModeling

reduces the memory requirements of the model, allowing for larger batch sizes, and significantly reduces the training time [16].

Table 5.5: Training arguments used for the fine-tuning of the MLM models.

Paramter	Value
FP16	*True*
Training batch size	32
Evaluation batch size	32
Optimizer	*AdamW*
Learning rate	$2e^{-5}$
Weight decay	0.01

Finally, to use these fine-tuned models for transfer learning and create new vulnerability detection models, we simply have to replace the model checkpoint (line 2 in Listing 5.1) with the checkpoint of the desired fine-tuned model. These models are available in the following repositories on the Hugging Face model hub: *up201806461/bert-java-bfp_combined*[12] and *up201806461/bert-java-bfp_single*[13].

5.2.6 Overview of the trained models

To comprehensively understand the vulnerability detection models' capabilities, we fine-tuned JavaBERT [12] and our MLMs on VDET, ND1 and ND2 datasets. The resulting models are named following the format [Dataset]-[MLM]-[Task]. For example, the $BFP_{combined}$ model fined-tuned on the ND2 dataset for the multiclass task is denoted as *ND2-*$BFP_{combined}$*-Multiclass*.

The experimental process led to the creation of three multi-label models, specifically *VDET-JavaBERT-Multilabel*, *ND1-JavaBERT-Multilabel* and *ND2-JavaBERT-Multilabel*. The *VDET-JavaBERT-Multilabel* model corresponds to the one developed by Mamede *et al.* [46]. However, considering the discrepancies between the publicly available dataset and the one presented in their work, we decided to train a new model to showcase a fair comparison to the multiclass models.

For the multiclass task, we developed nine different models. These correspond to all MLMs, including JavaBERT, BFP_{single} and $BFP_{combined}$, each fine-tuned on the VDET, ND1, and ND2 datasets.

5.2.7 Evaluation procedure

After training the vulnerability detection models we chose the optimal checkpoint, based on their learning curves, for evaluation. Since our goal is to minimize the loss function, the checkpoint with the lowest loss value, positioned before any indication of overfitting, was selected. Section A.1.2 of the Appendix depicts the learning curves of all models, with the best checkpoints highlighted.

[12]https://huggingface.co/up201806461/bert-java-bfp_combined
[13]https://huggingface.co/up201806461/bert-java-bfp_single

To illustrate this process, consider Figure 5.7, which presents the learning curves of the *ND2-$BFP_{combined}$-Multiclass* model. The optimal checkpoint, epoch 7, is highlighted by the dotted vertical line. Between epoch 4 and 7, the validation loss stabilizes on its lowest value. Beyond epoch 7, while the training loss continued to decrease and training accuracy kept improving, the validation loss started increasing, and the validation accuracy plateaued, which indicates overfitting.

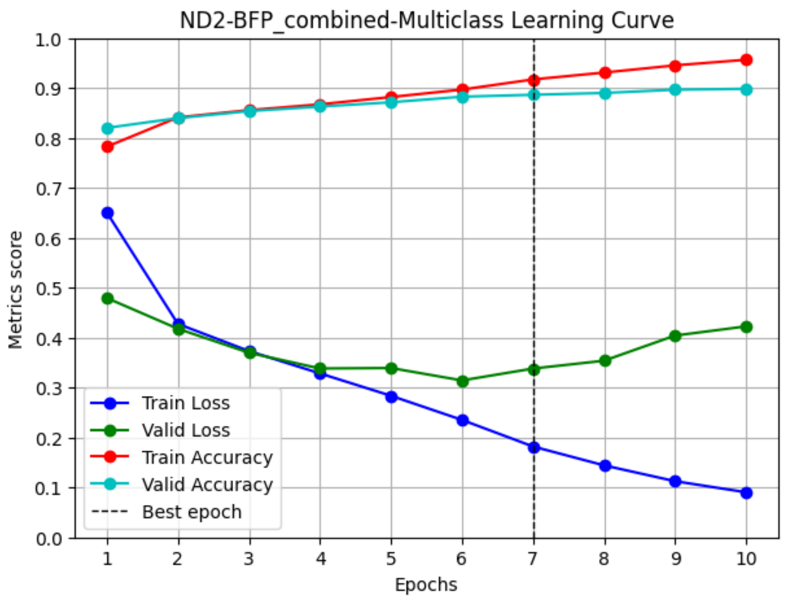


Figure 5.7: *ND2-$BFP_{combined}$-Multiclass* model's learning curve

All these models were then evaluated on the test set of their respective datasets, as well as the real-world test set, to determine their practical applicability. To scrutinize potential bias in the models, we performed cross-validation, evaluating each JavaBERT multi-label model on the test sets of the other two datasets.

The models' performance is assessed using the metrics described in Section 2.8, including accuracy, precision, recall and F1-score, as well as FPR and FNR. Given the imbalance nature of the datasets used in this research, accuracy is not a reliable metric to evaluate the models' performance. Instead, we focus on the weighted average precision, recall and F1-score values. It is important to note that function used to calculate the accuracy, *accuracy_score*[14] from sklearn computes the subset accuracy in multi-label classification contexts. This means that for a sample to be considered correctly classified, the set of predicted labels must exactly correspond to the set of actual labels in the ground truth.

The evaluation methodology adopted allows for a comparison of the performance of different models when fine-tuned on the same dataset. This offers insights into the transfer learning capabilities of the MLMs and their real-world effectiveness. It also allows us to compare the models'

[14] `https://scikit-learn.org/stable/modules/generated/sklearn.metrics.accuracy_score.html`

performance and any biases across different datasets. In addition, by comparing models with different architectures, we obtain a more comprehensive understanding of which model architecture is best suited for detecting vulnerabilities in Java code.

5.3 Developing J-TAS Action

In the process of developing our Java vulnerability detection Action, the initial requirement was a script that could effectively utilise our vulnerability detection models and present the results in a user-friendly and comprehensible format. To this end, we created a Python script that examines Java files in a directory and then generates a SARIF file with the results. We use the *ND2-BFP*$_{single}$*-Multiclass* model for the analysis due to its superior performance compared to the other models, as discussed later in Section 6.4.

The culmination of this process was the creation of a Action that utilizes the script. This Action is designed to seamlessly integrate into any workflow, thereby facilitating the detection of vulnerabilities in Java code. The J-TAS Action, along with the source code for the script, is readily accessible for use via our public GitHub repository[15].

5.3.1 Analysis script

Our Python script begins by parsing all Java files located within a specified directory and extracting all methods within these files. This is accomplished by utilising the *javalang* library.

Following the parsing phase, the script then proceeds to load the pre-trained vulnerability detection model and iterates over the methods for analysis. Each method is normalised, by removing unnecessary elements like comments and extra whitespace, and then tokenised. Finally, the preprocessed code is subjected to the analysis of the model, which outputs the most probable label.

If a given method is predicted as vulnerable (indicated by the label being a CWE-ID) and the probability of the prediction surpasses 50%, the method is added to a list of results. After analysing all methods, the script generates a SARIF file that encapsulates the results.

By default, this script analyses all Java files of a repository. However, users can provide specific files or directories as input for analysis to enhance versatility. This functionality is especially beneficial in large repositories, where analysing the entire repository instead of only the modified files could hamper the developer's workflow due to the time consumed.

Moreover, the script is designed with a strong emphasis on modularity, facilitating the integration of new models. By replacing the model utilised and adjusting the dictionary that maps the model's output to the corresponding label, the script can be updated with new models that cover more vulnerabilities or target different programming languages with minimal effort.

[15]`https://github.com/andrenasx/CVE-2015-10034`

5.3.2 SARIF support

The Static Analysis Results Interchange Format (SARIF) is an open standard, managed by the OASIS SARIF Technical Commitee[16], that defines a standard output format for static analysis tools. Its primary purpose is to facilitate the interchange of static analysis results among various tools and services, reducing the complexity of integrating the results of various tools into common workflows [53].

SARIF employs a JSON schema to outline the structure of the output file. It includes information about the tool that generated the results, the analysis run, and the vulnerabilities identified, and it can even pinpoint the location of the vulnerability within the source code. Despite its original design for static analysis tools, we can conveniently adapt the results of our deep-learning classification model to the SARIF format, as it outputs the vulnerabilities detected for each method. This facilitates the integration of our tool into existing workflows that support SARIF.

GitHub supports a subset of the SARIF 2.1.0 schema for code scanning. Developers can upload SARIF files to a GitHub repository, and GitHub parses the file and displays the results as part of the code scanning interface, in the *Security* tab of the repository. To utilise GitHub Actions for uploading a third-party SARIF file to a repository, we simply need to define a workflow that employs the *upload-sarif* action from GitHub's CodeQL [30]. This aligns perfectly with our desired Action workflow, as we can use the *upload-sarif* action to upload the SARIF file generated by our script to the repository.

Listing 5.2 presents an example of a SARIF file generated by our script. The *tool* object contains information about J-TAS and a *rules* list, which catalogues the identified vulnerabilities. Each entry contains information about the vulnerability, such as the CWE-ID, name, description, and a link to the respective CWE page on the MITRE website so that developers can learn more about the vulnerability.

Further down the file, we encounter the results list, which enumerates the specific vulnerabilities detected in the Java code for each file analysed by our tool. Each result is mapped to a rule to identify the CWE present and includes the prediction probability calculated by our model. Additionally, the *locations* list details the location of the vulnerability within the source code, specifying the file path and the lines of code where the vulnerability is located (identifying the method).

```
{
 "version": "2.1.0",
 "$schema": "https://raw.githubusercontent.com/oasis-tcs/sarif-spec/master/
     ↪ Schemata/sarif-schema-2.1.0.json",
 "runs": [
   {
    "tool": {
      "driver": {
        "name": "J-TAS",
```

[16] https://www.oasis-open.org/committees/sarif/

```
        "version": "0.1.0",
        "informationUri": "https://github.com/andrenasx/J-TAS",
        "rules": [
          {
            "id": "J-TAS/CWE-89",
            "name": "Improper Neutralization of Special Elements used in an SQL
                ↪ Command ('SQL Injection')",
            "shortDescription": {
              "text": "The product constructs all or part of an SQL command using
                  ↪ externally-influenced input from an upstream component, but it
                  ↪ does not neutralize or incorrectly neutralizes special elements
                  ↪ that could modify the intended SQL command when it is sent to a
                  ↪ downstream component."
            },
            "fullDescription": {
              "text": "Without sufficient removal or quoting of SQL syntax in user-
                  ↪ controllable inputs, the generated SQL query can cause those
                  ↪ inputs to be interpreted as SQL instead of ordinary user data.
                  ↪ This can be used to alter query logic to bypass security checks,
                  ↪  or to insert additional statements that modify the back-end
                  ↪ database, possibly including execution of system commands. SQL
                  ↪ injection has become a common issue with database-driven web
                  ↪ sites. The flaw is easily detected, and easily exploited, and as
                  ↪  such, any site or product package with even a minimal user base
                  ↪  is likely to be subject to an attempted attack of this kind.
                  ↪ This flaw depends on the fact that SQL makes no real distinction
                  ↪  between the control and data planes."
            },
            "helpUri": "https://cwe.mitre.org/data/definitions/89.html",
            "help": {
              "text": "For additional information on this weakness, visit the CWE-89
                  ↪ page on the MITRE website: https://cwe.mitre.org/data/
                  ↪ definitions/89.html",
              "markdown": "For additional information on this weakness, visit the [
                  ↪ CWE-89 page](https://cwe.mitre.org/data/definitions/89.html) on
                  ↪ the MITRE website."
            },
            "defaultConfiguration": {
              "level": "warning"
            },
            "properties": {
              "tags": [
                "security",
                "CWE-89"
              ]
            }
          },
          {
            "id": "J-TAS/CWE-789",
            "name": "Memory Allocation with Excessive Size Value",
            "shortDescription": {
              "text": "The product allocates memory based on an untrusted, large size
                  ↪ value, but it does not ensure that the size is within expected
                  ↪ limits, allowing arbitrary amounts of memory to be allocated."
            },
            "helpUri": "https://cwe.mitre.org/data/definitions/789.html",
            "help": {
              "text": "For additional information on this weakness, visit the CWE-789
                  ↪ page on the MITRE website: https://cwe.mitre.org/data/
                  ↪ definitions/789.html",
              "markdown": "For additional information on this weakness, visit the [
                  ↪ CWE-789 page](https://cwe.mitre.org/data/definitions/789.html)
                  ↪ on the MITRE website."
            },
            "defaultConfiguration": {
              "level": "warning"
            },
            "properties": {
              "tags": [
                "security",
                "CWE-789"
              ]
            }
          }
        ]
      }
    },
    "results": [
      {
        "ruleId": "J-TAS/CWE-89",
        "message": {
          "text": "CWE-89 predicted with 71.84% probability."
        },
        "locations": [
          {
            "physicalLocation": {
              "artifactLocation": {
                "uri": "src/app/database/UsersDao.java",
                "uriBaseId": "%SRCROOT%"
              },
              "region": {
                "startLine": 224,
                "endLine": 254
              }
            }
          }
        ]
      },
```

```json
        {
          "ruleId": "J-TAS/CWE-789",
          "message": {
            "text": "CWE-789 predicted with 54.18% probability."
          },
          "locations": [
            {
              "physicalLocation": {
                "artifactLocation": {
                  "uri": "src/app/controllers/UsersController.java",
                  "uriBaseId": "%SRCROOT%"
                },
                "region": {
                  "startLine": 31,
                  "endLine": 41
                }
              }
            }
          ]
        }
      ]
    }
  ]
}
```

Listing 5.2: J-TAS SARIF file example

5.3.3 Creating a Docker container Action

GitHub has three types of Actions: JavaScript, Docker container, and composite actions. JavaScript actions, as the name suggests, are designed to run JavaScript code. Since our models were implemented using Python libraries, this type of Action is unsuitable for our use case. Composite actions are a collection of workflow run steps and primarily execute shell scripts. While it would be possible to run a Python script using a composite action, all the necessary files would need to be included in the repository, which is not possible in our case, as the size of our binarized model file (1.2GB) exceeds GitHub's Large File Storage limit of 1GB per account.

Consequently, we opted for a Docker container Action. This type of Action allows us to construct a custom environment with all the necessary pre-installed dependencies and include the model within the image. We followed GitHub's comprehensive "Creating a Docker container Action" guide [27] to develop our Action.

To create a Docker container Action, we need to create a Dockerfile that defines the container's environment. Listing 5.3 presents the Dockerfile for our Action. The first line specifies the base image for the container, which, in our case, is an official Python image. We utilize the *3.9-slim* tag, a lightweight version of the image that includes only the essential packages required for Python to run. Subsequent lines install the dependencies required for the script and copy all the necessary action files (scripts and model) into the container. Line 7 defines a command to remove all cache to help reduce the image size. Lastly, we define the entrypoint for the container. This bash script executes the Python script and copies the resulting SARIF file to the GitHub workspace inside the container, so it can later be uploaded to the repository.

```
 1  FROM python:3.9-slim
 2
 3  COPY requirements.txt /gaction/requirements.txt
 4
 5  RUN cd /gaction && \
 6     pip install -r requirements.txt && \
 7     rm -rf /var/lib/apt/lists/*
 8
 9  COPY . /gaction
10
11  ENTRYPOINT ["/gaction/entrypoint.sh"]
```

Listing 5.3: J-TAS Dockerfile.

Building an image as lightweight as possible is crucial, as it will be downloaded every time the Action is run, affecting workflow times. This is why we use the slim version of the Python image, group the installation of dependencies in a single instruction, remove installation cache, and only copy the necessary files to the container image.

After creating the Dockerfile, we must build the image and push it to a container registry. Our image is hosted on our repository[17] on Docker Hub, a cloud-based registry service that allows us to store and distribute Docker images. We can then reference the image in our Action workflow file, as shown in Listing 5.4. All the information regarding usage and suggestions to integrate this Action into your workflow is included in the Action's README file presented in Listing B.1 in Appendix, which is displayed in the Action's page on the GitHub Marketplace.

5.3.4 J-TAS Action practical demonstration

To demonstrate J-TAS' usage in practice, we decided to create a dedicated test repository containing known vulnerabilities. First, we examined the outputs generated by the *ND2-BFP$_{single}$-Multiclass* model. The goal was to select a specific CVE from the set of vulnerabilities that our model had correctly predicted and that we could easily analyse. This scrutiny led us to the selection of *CVE-2015-10034*[18].

Subsequently, we forked the parent commit of the fix commit associated with this specific CVE, creating a demo repository with vulnerable code. Our Action was configured to analyse the files modified in the fix commit exclusively, so we could later compare the results with the actual vulnerabilities present in the code.

[17] https://hub.docker.com/repository/docker/up201806461/j-tas/general
[18] https://nvd.nist.gov/vuln/detail/CVE-2015-10034

```yaml
name: 'J-TAS'
description: 'Analyses Java files for vulnerabilities'

inputs:
  paths:
      description: 'Paths to analyse (relative to the repository root, separated by
          ↪ spaces)'
      required: false
      default: ''
  files:
    description: 'Files to analyse (paths relative to the repository root, separated
        ↪  by spaces)'
    required: false
    default: ''

runs:
  using: 'docker'
  image: 'docker://up201806461/j-tas:latest'
  args:
    - ${{ inputs.paths }}
    - ${{ inputs.files }}
```

Listing 5.4: J-TAS Action yaml file.

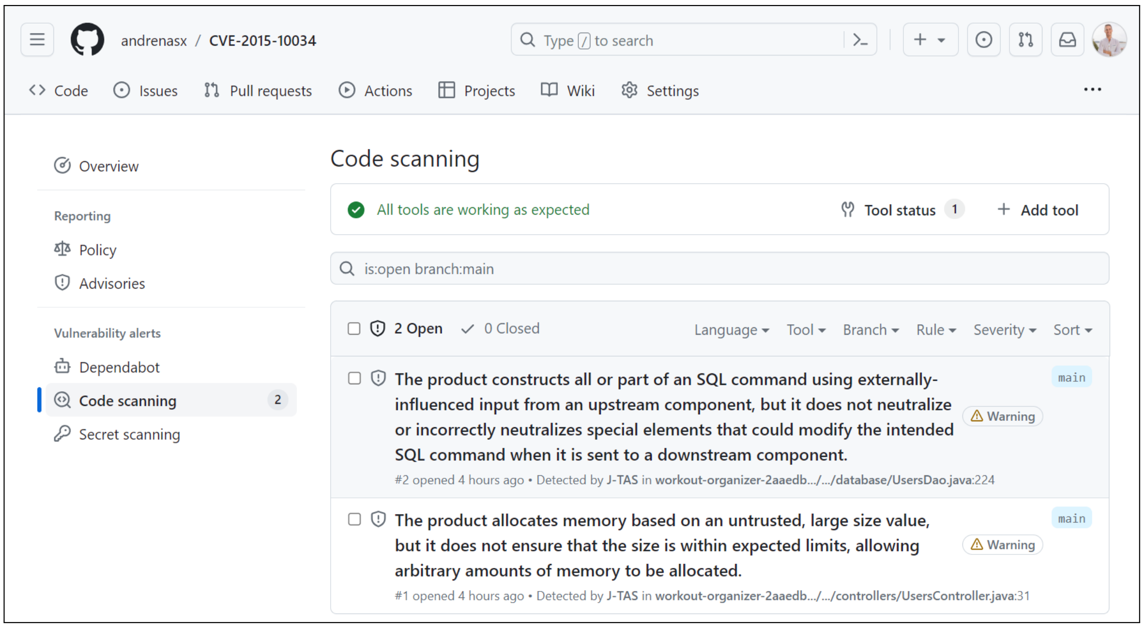
The workflow file used to configure the analysis is presented in Listing 5.5. The first step uses GitHub's checkout Action[19] to clone the repository onto the virtual machine. The second step runs the J-TAS Action, with the files designated for analysis specified using the *files* parameter. The final step involves uploading the SARIF file, generated by the Action, to the repository using the *upload-sarif* Action[20] from GitHub's CodeQL.

The code scanning tab of the repository displays a list of the SARIF file's parsed vulnerabilities, as shown in Figure 5.8. The title of each result is a short description of the CWE to provide a quick insight into the vulnerability. Developers are able to filter the results by several criteria in order to show the vulnerabilities that are most relevant to them.

[19] https://github.com/actions/checkout
[20] https://github.com/github/codeql-action/tree/main/upload-sarif

```yaml
on: [push]
name: J-TAS Action Demo

jobs:
 analysis:
  runs-on: ubuntu-latest

  permissions:
   actions: read
   contents: read
   security-events: write

  steps:
   - name: Checkout this repo code
     uses: actions/checkout@v3

   - name: Run J-TAS
     uses: andrenasx/J-TAS@main
     with:
      files: 'workout-organizer-2aaedb7f69ea398a0217493fefe8f19b31ee0775/src/app/
          ↪ controllers/AccountController.java workout-organizer-2
          ↪ aaedb7f69ea398a0217493fefe8f19b31ee0775/src/app/controllers/
          ↪ Application.java workout-organizer-2
          ↪ aaedb7f69ea398a0217493fefe8f19b31ee0775/src/app/controllers/
          ↪ UsersController.java workout-organizer-2
          ↪ aaedb7f69ea398a0217493fefe8f19b31ee0775/src/app/database/ExerciseDao.
          ↪ java workout-organizer-2aaedb7f69ea398a0217493fefe8f19b31ee0775/src/
          ↪ app/database/GymsDao.java workout-organizer-2
          ↪ aaedb7f69ea398a0217493fefe8f19b31ee0775/src/app/database/UsersDao.
          ↪ java workout-organizer-2aaedb7f69ea398a0217493fefe8f19b31ee0775/src/
          ↪ app/database/WorkoutDao.java'

   - name: Upload J-TAS results
     uses: github/codeql-action/upload-sarif@v2
     with:
      sarif_file: results.sarif
      category: j-tas
```

Listing 5.5: Test repository's workflow file.


andrenasx / CVE-2015-10034

Type / to search

Code · Issues · Pull requests · Actions · Projects · Wiki · Settings

Overview

Reporting
Policy
Advisories

Vulnerability alerts
Dependabot
Code scanning 2
Secret scanning

Code scanning

All tools are working as expected — Tool status 1 — Add tool

is:open branch:main

2 Open ✓ 0 Closed — Language · Tool · Branch · Rule · Severity · Sort

The product constructs all or part of an SQL command using externally-influenced input from an upstream component, but it does not neutralize or incorrectly neutralizes special elements that could modify the intended SQL command when it is sent to a downstream component. — main — Warning
#2 opened 4 hours ago • Detected by J-TAS in workout-organizer-2aaedb.../.../database/UsersDao.java:224

The product allocates memory based on an untrusted, large size value, but it does not ensure that the size is within expected limits, allowing arbitrary amounts of memory to be allocated. — main — Warning
#1 opened 4 hours ago • Detected by J-TAS in workout-organizer-2aaedb.../.../controllers/UsersController.java:31

Figure 5.8: J-TAS analysis results displayed in the code scanning tab.

Upon selecting a specific vulnerability to examine, the complete details of the analysis become accessible. Figure 5.9 shows the details of the second alert in our test repository. On the right-hand side of the page, developers can dismiss the code alert or create an issue to rectify it. It also shows the alert's severity, tags and CWE, all properties we defined in the SARIF file, and the branch where the vulnerability was detected. The centre of the page displays the code viewer with the vulnerable method highlighted and the probability of the predicted CWE as calculated by our model. Finally, on the bottom section of the page, we find information regarding the tool that generated the results, the matched rule, and the help text, which contains a link to the respective CWE page.

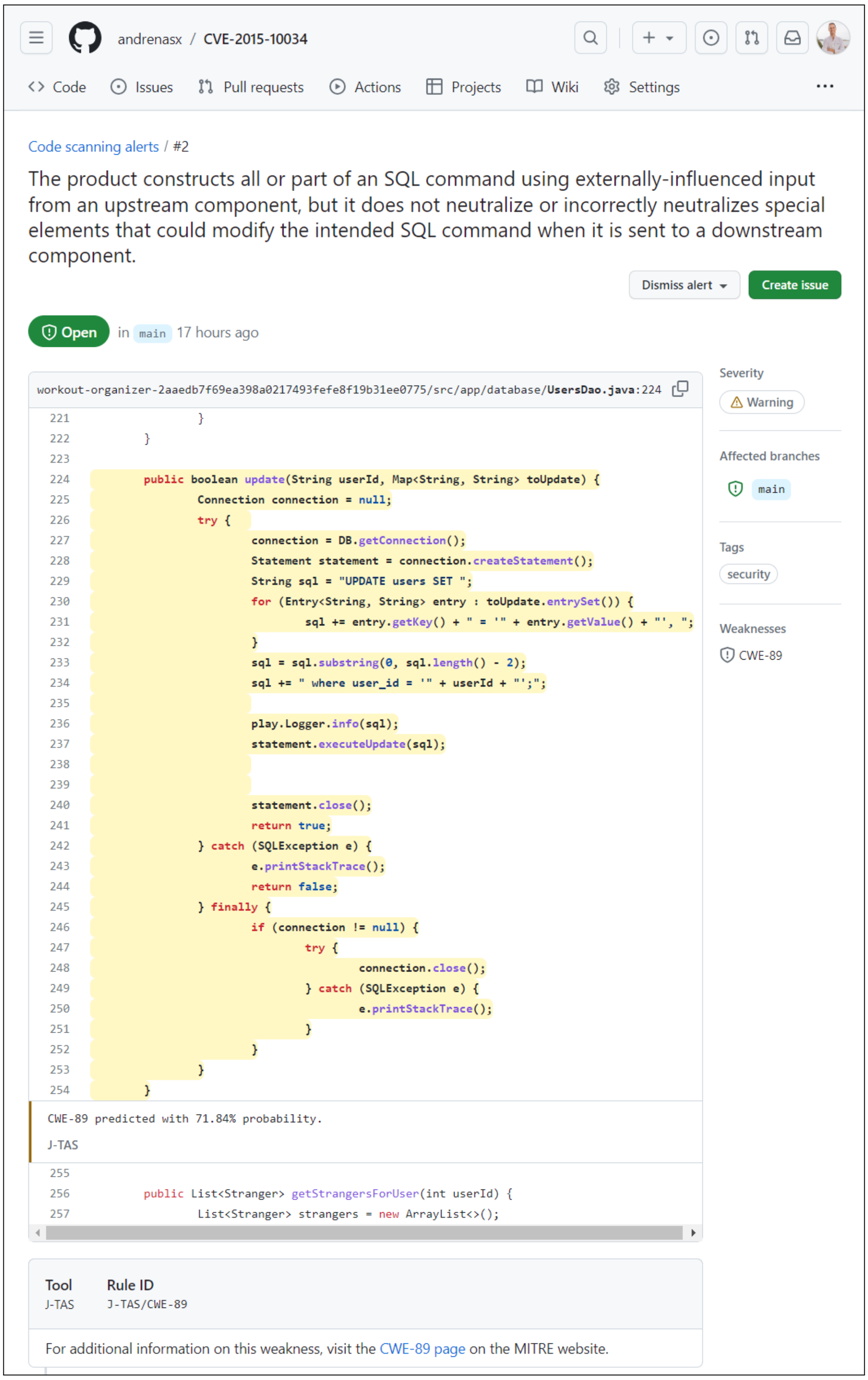
This example demonstrates our successful exploitation of GitHub's code scanning interface to display the results derived from our deep learning vulnerability detection model, effectively bridging the gap between DL techniques and practical software security auditing.

5.3.5 J-TAS evaluation procedure

To assess J-TAS's performance, we designed a process that takes into account both the model's prediction capabilities and the efficiency of the GitHub Action, using the test repository described in the previous section.

The first part of the evaluation process involves comparing the vulnerabilities detected with the actual vulnerabilities present in the code. Our Action analysed all methods in the files that were changed in the *CVE-2015-10034* fix commit. However, it's worth noting that our real-world test set comprises only those methods that were explicitly modified in the fix commit. This led to the analysis of additional methods which were not part of our test set. To account for these extra methods, we carried out a manual review and labeled them as not vulnerable. With the complete ground truth, we compared the results of the Action with the actual vulnerabilities present in the code, allowing us to determine the tool's ability to correctly identify vulnerabilities in real-world code.

The second stage of the evaluation process involves assessing the Action's efficiency. The *Actions* tab of the repository displays the time taken by the Action to run, even detailed by each step. In order to generate a robust measurement of the Action's efficiency, we executed the workflow three times and registered the time consumed by each step. From this data, we were able to calculate the average time taken for each step and the overall time taken by the Action to run.



The product constructs all or part of an SQL command using externally-influenced input from an upstream component, but it does not neutralize or incorrectly neutralizes special elements that could modify the intended SQL command when it is sent to a downstream component.

workout-organizer-2aaedb7f69ea398a0217493fefe8f19b31ee0775/src/app/database/**UsersDao.java**:224

```
221                 }
222         }
223 
224         public boolean update(String userId, Map<String, String> toUpdate) {
225                 Connection connection = null;
226                 try {
227                         connection = DB.getConnection();
228                         Statement statement = connection.createStatement();
229                         String sql = "UPDATE users SET ";
230                         for (Entry<String, String> entry : toUpdate.entrySet()) {
231                                 sql += entry.getKey() + " = '" + entry.getValue() + "', ";
232                         }
233                         sql = sql.substring(0, sql.length() - 2);
234                         sql += " where user_id = '" + userId + "';";
235 
236                         play.Logger.info(sql);
237                         statement.executeUpdate(sql);
238 
239 
240                         statement.close();
241                         return true;
242                 } catch (SQLException e) {
243                         e.printStackTrace();
244                         return false;
245                 } finally {
246                         if (connection != null) {
247                                 try {
248                                         connection.close();
249                                 } catch (SQLException e) {
250                                         e.printStackTrace();
251                                 }
252                         }
253                 }
254         }
```

CWE-89 predicted with 71.84% probability.

J-TAS

```
255 
256         public List<Stranger> getStrangersForUser(int userId) {
257                 List<Stranger> strangers = new ArrayList<>();
```

Tool	Rule ID
J-TAS	J-TAS/CWE-89

For additional information on this weakness, visit the CWE-89 page on the MITRE website.

Figure 5.9: J-TAS specific result details.

Chapter 6

Results and Discussion

This chapter provides a comprehensive presentation and insightful discussion of the results obtained from the experiments conducted in this dissertation. The outcomes of these experiments, designed to address our research questions, are presented in Sections 6.1 through 6.4. Section 6.5 delves into in-depth analysis and discussion of the findings regarding the several datasets, models, and the final tool developed. In order to provide a comprehensive assessment, Section 6.6 examines the potential threats to the validity of our findings. Finally, Section 6.7 focuses on evaluating the tool developed in this dissertation, considering the functional and non-functional requirements defined in Section 4.4.

6.1 RQ1: How do different dataset normalisation techniques influence the performance of the vulnerability detection models?

To answer this research question, we compared the results of *ND1-JavaBERT-Multilabel* and *ND2-JavaBERT-Multilabel* with the results of the model developed by Mamede *et al.* [46] which we named *VDET-JavaBERT-Multilabel*.

The performance results of our models, as illustrated in Table 6.1, reveal noteworthy findings. The *VDET-JavaBERT-Multilabel* model demonstrates superior performance across most metrics, with an impressive weighted F1-score of 94%, precision of 95%, and recall of 93%, suggesting an effective balance between precision and recall. Further, it achieves the lowest mean FNR at 6.84%, the only model below 10%.

The models trained on our normalised datasets, *ND1-JavaBERT-Multilabel* and *ND2-JavaBERT-Multilabel*, show fairly comparable performances. Both models achieve similar recall rates of 88%, with the ND1 model slightly outperforming the ND2 model in terms of the F1-score and precision, while the ND2 model excels in having a lower mean FNR. These models do not fall short of the *VDET-JavaBERT-Multilabel* in all metrics, except for the mean FNR which shows a significant difference.

Table 6.1: *JavaBERT*-based multi-label models' evaluation results on the test set.

Model	Subset Accuracy	(W) F1-score	(W) Precision	(W) Recall	Mean FNR	Mean FPR
VDET-JavaBERT-Multilabel	85.75%	94%	95%	93%	6.84%	0.90%
ND1-JavaBERT-Multilabel	76.30%	92%	96%	88%	24.61%	0.50%
ND2-JavaBERT-Multilabel	77.95%	89%	91%	88%	17.41%	1.15%

To assess the real-world capabilities of the models, we also evaluate them on our real-world test set. The results are presented in Table 6.2. In this scenario all models show very similar performance between them, but is is evident that these models do not perform as well as they do on their respective test sets.

The Subset Accuracy for all models is extremely low, ranging from 2.63% to 3.45%, which indicates that the models have a hard time correctly predicting the complete set of labels for a given instance. The recall rates are roughly 60% lower than those obtained on the test sets, resulting in a significant impact on the F1-score, which now stands at around 30%. This indicates that the models cannot correctly predict the labels for most instances in the real-world test set.

Moreover, all models experience a significant increase in mean FPR compared to the test sets. While the mean FNR shows a slight increase for the *ND2-JavaBERT-Multilabel* model, the *ND1-JavaBERT-Multilabel* model manages to decrease it. However, the *VDET-JavaBERT-Multilabel* model encounters a significant increase in FNR.

Table 6.2: *JavaBERT*-based multi-label models' evaluation results on the real-world test set.

Model	Subset Accuracy	(W) F1-score	(W) Precision	(W) Recall	Mean FNR	Mean FPR
VDET-JavaBERT-Multilabel	2.63%	31%	68%	29%	25.56%	7.30%
ND1-JavaBERT-Multilabel	3.45%	30%	68%	29%	21.10%	6.27%
ND2-JavaBERT-Multilabel	3.45%	31%	62%	29%	23.64%	7.10%

Finding 1: *All models achieve similar results in the real-world test set.*

Finding 2: *The models' performance drops significantly when evaluated on the real-world test set.*

Furthermore, to examine the models' generalisation capabilities, we cross-evaluated the *JavaBERT*-based models, assessing their performance on the test sets derived from the other datasets. This also allows us to identify potential bias in the models introduced by the dataset used, and validate whether the results are inflated by that or not. The results of this analysis are summarised in Table 6.3.

Table 6.3: *JavaBERT*-based multi-label models' cross-evaluation results.

Model	Test set	Subset Accuracy	(W) F1-score	(W) Precision	(W) Recall	Mean FNR	Mean FPR
VDET-JavaBERT-Multilabel	ND1	47.85%	71% (-23%)	80%	68%	31.67% (+24.83%)	3.34% (+2.44%)
	ND2	28.73%	55% (-39%)	70%	51%	49.07% (+42.23%)	4.77% (+3.87%)
ND1-JavaBERT-Multilabel	VDET	60.18%	80% (-12%)	84%	78%	26.84% (+2.23%)	1.98% (+1.48%)
	ND2	49.03%	75% (-17%)	84%	70%	37.20% (+12.59%)	1.42% (+0.92%)
ND2-JavaBERT-Multilabel	VDET	65.52%	82% (-7%)	83%	81%	25.50% (+8.09%)	2.15% (+1%)
	ND1	79.25%	91% (+2%)	93%	89%	21.48% (+4.07%)	0.79% (+0.36%)

Note: **(W) F1-score**, **Mean FNR** and **Mean FPR** columns also show the difference of the values compared to the evaluation of the models on their own test set (values in Tables 6.1)

When the *VDET-JavaBERT-Multilabel* model is evaluated on the ND1 and ND2 test sets, we observe a serious decrease in performance. The weighted F1-score falls to 71% and 55% on the ND1 and ND2 test sets, respectively, which are substantially lower than the 94% achieved on the VDET test set.

On the other hand, both *ND1-JavaBERT-Multilabel* and *ND2-JavaBERT-Multilabel* models show promising generalization potential. The *ND1-JavaBERT-Multilabel* model exhibits consistent performance on both the VDET and ND2 test sets, with weighted F1-scores of 80% and 75%, respectively. Still, it represents a considerable drop in performance compared to the 92% achieved on the ND1 test set.

The *ND2-JavaBERT-Multilabel* model shines in this evaluation with weighted F1-scores of 82% and 91% on the VDET and ND1 test sets, respectively. The results on the ND1 test set are reasonably close to the ones achieved on its native ND2 test set.

> ***Finding 3:*** *The model trained on the ND2 dataset has the best generalisation capabilities.*

6.2 RQ2: How does a multiclass architecture compare to a multi-label architecture in terms of performance?

Before transitioning to a multiclass architecture, we analysed the multi-label models' predictions more profoundly to better understand their performance. This led to an interesting finding: the models output conflicting predictions. In our multi-label context, we define a "conflicting prediction" as one where the model predicts a sample as both "True" (vulnerable) and "False" (not vulnerable). Additionally, a "conflicting prediction" can also arise when the model only predicts the CWE label without the associated vulnerability status label or vice-versa. Table 6.4 presents the number of conflicting predictions for each multi-label model on the test set and real-world test set.

All multi-label models output a considerable number of conflicting predictions, mainly when evaluated on the real-world test set. In this case, the number of conflicting predictions is more than

Table 6.4: *JavaBERT*-based multi-label models' number of conflicting predictions on the test set and real-world evaluation set.

Model	Number of test samples	Number of conflicting predictions
	Test set evaluation	
VDET-JavaBERT-Multilabel	11527	692 (6.00%)
ND1-JavaBERT-Multilabel	4126	717 (17.37%)
ND2-JavaBERT-Multilabel	8064	587 (7.28%)
	Real-world set evaluation	
VDET-JavaBERT-Multilabel	609	206 (33.82%)
ND1-JavaBERT-Multilabel	609	252 (41.22%)
ND2-JavaBERT-Multilabel	609	224 (36.78%)

one-third of the total test samples for all models. The *ND1-JavaBERT-Multilabel* model produces the highest number of conflicting predictions on both test sets.

> ***Finding 4:*** *The multi-label models produce a considerable number of conflicting predictions, especially when evaluated on the real-world test set.*

Adapting the models to a multiclass architecture led to interesting performance changes compared to their multi-label counterparts. Table 6.5 illustrates the results of our JavaBERT-based multiclass models on the test set and real-world evaluation set. Since multiclass models only predict a single label per sample, they inherently do not produce conflicting predictions.

Table 6.5: *JavaBERT*-based multiclass models' evaluation results on the test set and real-world evaluation set.

Model	Accuracy	(W) F1-score	(W) Precision	(W) Recall	Mean FNR	Mean FPR
			Test set evaluation			
VDET-JavaBERT-Multiclass	90.61%	91% (-3%)	91% (-4%)	91% (-2%)	15.98% (+9.14%)	0.92% (+0.02%)
ND1-JavaBERT-Multiclass	86.17%	86% (-6%)	88% (-8%)	86% (-2%)	32.31% (+7.7%)	0.73% (+0.23%)
ND2-JavaBERT-Multiclass	87.14%	87% (-2%)	88% (-3%)	87% (-1%)	25.04% (+7.63%)	0.87% (-0.28%)
			Real-world set evaluation			
VDET-JavaBERT-Multiclass	55.34%	45% (+14%)	49% (-19%)	55% (+26%)	22.84% (-2.72%)	4.47% (-2.63%)
ND1-JavaBERT-Multiclass	55.67%	42% (+12%)	46% (-22%)	56% (+27%)	20.14% (-0.96%)	3.98% (-2.29%)
ND2-JavaBERT-Multiclass	54.19%	42% (+11%)	34% (-28%)	54% (+25%)	23.01% (-0.63%)	4.41% (-2.69%)

Note: **(W) F1-score**, **(W) Precision**, **(W) Recall**, **Mean FNR** and **Mean FPR** columns also show the difference of the values compared to the multi-label counterparts of the models (values in Tables 6.1 and 6.2)

Upon evaluating the test set, all multiclass models exhibited a dip in F1-score and registered concerning increases in the mean FNR values. The variation of the mean FPR is negligible. Mirroring the patterns observed in the multi-label scenario, VDET's multiclass model achieves better results across all metrics.

On the other hand, the evaluation of the real-world test set shows a clear improvement across the metrics. The F1-score values show increases in the 8%-11% range, but, unfortunately, they are still below 50%. Furthermore, the mean FPR of all models drops below 5% for all models. A significant change noticed in this scenario is the higher balance between precision and recall, due to significant increases in recall coupled with decreases in precision.

The models trained on our normalised datasets have the same F1-score, but *ND1-JavaBERT-Multiclass* has higher precision than *ND2-JavaBERT-Multiclass* (12% more). VDET's multiclass model achieves slightly better results than these models.

> ***Finding 5:*** *When evaluated on the real-world test set, the multiclass models show a significant performance improvement compared to the multi-label models.*

6.3 RQ3: How does the performance of a model pre-trained on Java buggy-related code compare to *JavaBERT* when applied to the same vulnerability detection task?

The $BFP_{combined}$ and BFP_{single} MLMs were developed to help us answer this question and analyse the impact of transfer learning on the performance of the models. The results of the multi-label vulnerability detection models pre-trained on these MLMs are presented in Table 6.6 for both the test set and real-world evaluation set.

When evaluated on the test set, the multiclass $BFP_{combined}$ and BFP_{single}-based models exhibit F1-scores that are on par with their JavaBERT-based counterparts. In terms of Mean FNR, all BFP models see a noticeable improvement, with decreases ranging from 0.85% to 7.67%. The decrease in mean FPR can be deemed negligible.

The performance dynamics shift after examining the results from the real-world test set. While the variation of the mean FNR and FPR are also negligible, the F1-scores register a slight decrease, with the $BFP_{combined}$-based models showing the most significant drop. This shift is characterized by a decrease in recall, offset to some extent by the increase in precision, except for the ND1-based models, which experience a decrease in both metrics. For example, the *ND2-*BFP_{single}*-Multiclass* model experiences a 4% decrease in recall compared to its JavaBERT-based counterpart but compensates with an impressive 16% increase in precision.

Nonetheless, when analysing the per-class predictions on the real-world test set for the multiclass models, presented in Tables C.15 to C.47, it becomes apparent that $BFP_{combined}$-based models have a higher count of true positives regarding vulnerabilities, with the exception of models trained on VDET's dataset. Table 6.7 illustrates this observation.

Table 6.6: $BFP_{combined}$ and BFP_{single}-based multiclass models' evaluation results on the test set and real-world evaluation set

Model	Accuracy	(W) F1-score	(W) Precision	(W) Recall	Mean FNR	Mean FPR
			Test set evaluation			
VDET-$BFP_{combined}$-Multiclass	90.63%	91% (0%)	92% (+1%)	91% (0%)	8.31% (-7.67%)	0.59% (-0.33%)
VDET-BFP_{single}-Multiclass	90.56%	91% (0%)	92% (+1%)	91% (0%)	9.02% (-6.96%)	0.63% (-0.29%)
ND1-$BFP_{combined}$-Multiclass	85.44%	86% (0%)	88% (0%)	85% (-1%)	31.46% (-0.85%)	0.70% (-0.03%)
ND1-BFP_{single}-Multiclass	86.17%	86% (0%)	88% (0%)	86% (0%)	29.85% (-2.46%)	0.70% (-0.03%)
ND2-$BFP_{combined}$-Multiclass	88.98%	89% (+2%)	89% (+1%)	89% (+2%)	20.71% (-4.33%)	0.80% (-0.07%)
ND2-BFP_{single}-Multiclass	87.39%	88% (+1%)	88% (0%)	87% (0%)	24.12% (-0.92%)	0.84% (-0.03%)
			Real-world set evaluation			
VDET-$BFP_{combined}$-Multiclass	46.14%	41% (-4%)	59% (+10%)	46% (-9%)	22.63% (-0.21%)	4.43% (-0.04%)
VDET-BFP_{single}-Multiclass	49.59%	43% (-2%)	54% (+5%)	50% (-5%)	22.33% (-0.51%)	4.32% (-0.15%)
ND1-$BFP_{combined}$-Multiclass	44.83%	38% (-4%)	43% (-3%)	45% (-11%)	20.45% (+0.31%)	4.16% (+0.18%)
ND1-BFP_{single}-Multiclass	42.36%	38% (-4%)	37% (-9%)	42% (-14%)	15.77% (-4.37%)	3.99% (+0.01%)
ND2-$BFP_{combined}$-Multiclass	49.10%	41% (-1%)	45% (+11%)	49% (-5%)	22.90% (-0.11%)	4.49% (+0.08%)
ND2-BFP_{single}-Multiclass	49.75%	42% (0%)	50% (+16%)	50% (-4%)	22.82% (-0.19%)	4.51% (+0.10%)

Note: **(W) F1-score**, **(W) Precision**, **(W) Recall**, **Mean FNR** and **Mean FPR** columns also show the difference of the values compared to the JavaBERT-based multiclass counterparts of the models (values in Tables 6.5)

Table 6.7: Multiclass models' number of correctly predicted vulnerabilities in the real-world test set

Model	Vulnerable samples TPs
VDET-JavaBERT-Multiclass	11
VDET-$BFP_{combined}$-Multiclass	6
VDET-BFP_{single}-Multiclass	10
ND1-JavaBERT-Multiclass	1
ND1-$BFP_{combined}$-Multiclass	3
ND1-BFP_{single}-Multiclass	5
ND2-JavaBERT-Multiclass	0
ND2-$BFP_{combined}$-Multiclass	2
ND2-BFP_{single}-Multiclass	8

Among all models, those trained on the ND2 dataset benefit most from the transition to *BFP*-based MLMs, with the *ND2-BFP_{single}-Multiclass* model achieving the most commendable results. However, it's important to note that the count of detected vulnerabilities remains considerably low in comparison to the total of 257 vulnerable samples present in the real-world test set.

Finding 6: *$BFP_{combined}$ and BFP_{single}-based model trained on our normalised datasets detect more vulnerabilities than their JavaBERT-based counterparts.*

6.4 RQ4: How can a DL model be integrated into a GitHub Action for real-time vulnerability analysis?

Our study's fourth research question (RQ4) was inherently technical, intending to explore the feasibility of integrating a Deep Learning analysis tool into a GitHub action. This section presents the practical results of this integration by evaluating the J-TAS Action on our test repository, as explained in Section 5.3.5.

The files subjected to analysis contain 17 non-vulnerable methods and 18 methods proven vulnerable to CWE-89. However, our Action could only classify one of the vulnerable methods correctly. The remaining methods were misclassified as non-vulnerable, except for one method incorrectly identified as vulnerable to CWE-789, although with a relatively low probability of just 54.18%. This translates into a substantial false negative rate (FNR) of 94% for the CWE-89 class, coupled with a high false positive rate (FPR) of 89% for the 'Not vulnerable' class.

Finding 7: *In practice, J-TAS confuse the classes, presenting high false positive and negative rates.*

Regarding the Action's runtime, Table 6.8 presents the average duration of the crucial steps of the Action's workflow. The total runtime of the Action averages 1 minute and 24 seconds, with the most time-consuming stage being pulling the Docker image to the runner, which takes an average of 46 seconds (55% of the total runtime). The time taken by the J-TAS analysis step is the only one that can vary between commits, as it is linearly proportional to the number of code tokens requested for analysis. In this specific case, the analysis covered 35 methods and processed a total of 8923 tokens, taking an average of 23 seconds (27% of the total runtime).

Table 6.8: Average duration of the J-TAS Action's workflow steps

Workflow Step	**Average duration**
Pull J-TAS Docker image	46s
Checkout repo code	1s
Run J-TAS analysis	23s
Upload SARIF	7s

6.5 Analysis and Discussion

6.5.1 RQ1 Analysis

Through **RQ1**, we found out that the model trained on the VDET dataset achieves the best performance on the test datasets, and that all models achieve similar results on our real-world test set (**Finding 1**). The cross-evaluation of the multi-label models reveals that the models trained on the ND2 dataset have the best generalisation capabilities in the test sets (**Finding 3**), and validates the inflated performance of VDET's model in the test set due to bias. However, *VDET-JavaBERT-Multilabel* still surpasses them on the real-world test set. This performance benchmark is achieved despite the dataset's shortcomings, such as duplicate entries, conflicting labels, and the bias due to insufficient code normalisation (confirmed by Mamede Mamede *et al.* [46]), VDET's models' slighty higher performance persists in the multiclass scenario, as seen in **RQ2** and **RQ3** results.

Our second fundamental discovery (**Finding 2**) highlights the performance drop of the multi-label models when evaluated on the real-world test set. This behaviour is also consistently corroborated in the multiclass scenarios, revealing a consistent challenge across different model configurations. Additionally, the results per class of the several models shown in Appendix C allow for a more in-depth analysis of the models' real-world performance. All models struggle to predict the CWEs, with the results heavily skewed towards the 'Not vulnerable' class.

We primarily attribute this performance gap to the synthetic nature of the dataset used to train the models, which does not recreate the complex and diverse patterns of real-world Java vulnerabilities. This finding underscores the need for models trained on more representative datasets, ideally constructed from real-world codebases, to enhance their performance and generalisation capabilities in practical applications.

6.5.2 RQ2 Analysis

For **RQ2**, we present the results of the multiclass models, which are compared to their multi-label counterparts. Despite the significant improvement of the multiclass models' performance on the real-world test set compared to the multi-label models (**Finding 4**), the models still struggle to achieve a satisfactory F1-score.

However, we argue that a direct comparison between our multi-label and multiclass models cannot be made. The multi-label datasets are labelled according to [CWE-ID, Vulnerability status], while the multiclass datasets are labelled with the encoded CWE-ID in case of a vulnerability or 0 (Not vulnerable) otherwise. This difference in labelling leads to a different learning process for the models, as the multi-label models are trained to predict both the vulnerability status and the CWE-ID and, therefore, also learn the CWE fix-patterns in the non-vulnerable methods.

The reason that we considered multiclass the best approach for our vulnerability detection task is the precise and unambiguous predictions that it provides. As demonstrated in Table 6.4, multi-label models output a considerable number of conflicting predictions, especially when evaluated on the real-world test set (**Finding 5**). If we were to implement a vulnerability detection tool based on these models, we would have been compelled to discard the conflicting predictions, leading to a significant increase in false negatives.

6.5.3 RQ3 Analysis

This research question allowed us to analyse the impact of transfer learning on the performance of the vulnerability detection models, by fine-tuning MLMs trained on vulnerability-related code. While JavaBERT [12] was trained on a large corpus of general-purpose Java code, comprised of almost 3 million files, we trained the $BFP_{combined}$ and BFP_{single} models on only 198088 and 236040 vulnerability related Java methods, respectively.

Remarkably, even with the significantly smaller MLM training datasets, the BFP-based models still manage to demonstrate comparable performance to the JavaBERT-based models on the test set and even exhibit a more balanced precision-recall trade-off when applied to the real-world test set (**Finding 6**). This finding is particularly encouraging, as it demonstrates the potential of transfer learning to enhance the performance of vulnerability detection models. Fine-tuning a MLM trained on a smaller, domain-specific (vulnerability-related) dataset can yield a vulnerability detection model as robust as one fine-tuned on a larger, more generic dataset.

This insight leads us to hypothesize that training MLMs on larger and vulnerability-specific datasets can further enhance the vulnerability detection model's performance. Therefore, while our models have demonstrated promising results, there is potential for even more significant improvements in the future.

Given these final results and the insights gained from the analysis of the models' performance, we considered the *ND2-*BFP_{single}*-Multiclass* model to be the best fit for our vulnerability detection tool.

6.5.4 RQ4 Analysis

Analysing the results from RQ4, it is evident that the performance of the J-TAS Action regarding vulnerability detection leaves much to be desired. This outcome aligns with the performance of the underlying vulnerability detection model, as documented in our earlier findings. Given the model's poor detection performance, it was obvious that the GitHub Action would struggle to identify vulnerabilities effectively.

Finding 7 reveals that our tool still has a long way to go before it can be considered a viable vulnerability detection tool. As such, we position the current implementation of the J-TAS Action as a proof-of-concept rather than a market-ready product. It demonstrates the technical feasibility of integrating a Deep Learning model into a GitHub Action, laying the groundwork for future improvements in the model performance.

In terms of runtime, the J-TAS Action exhibits good performance. The time taken by the J-TAS analysis step is reasonably short, considering the number of methods processed. However, the stage of pulling the J-TAS Docker image to the runner should be optimised to improve the

Action's total runtime. Specifically, efforts to reduce the Docker image size would likely yield a substantial decrease in the time taken to pull the image, speeding up the overall process.

6.6 Threats to validity

The validity of the experiments and findings described in this thesis may be subject to the following threats:

Reliance on synthetic data source: The dataset used for training and evaluating the models is derived from the synthetic Juliet Test Suite for Java. While this suite provides a structured and controlled environment for vulnerability detection research, it does not emulate real-world Java vulnerability patterns. Consequently, this could limit the applicability and effectiveness of our models in real-world scenarios.

Reduced sample size and imbalanced data: The ND1 and ND2 datasets constructed suffer from reduced sample size and skewed data distribution across the CWEs. A smaller sample can limit the diversity of patterns the model can learn from, potentially hindering its ability to generalize to unseen data. Furthermore, the class imbalance can bias the model towards the over-represented classes, undermining its performance on under-represented vulnerability types.

Real-world test set data quality Our real-world test set, used for model evaluation, consists of merely 609 methods and covers a limited array of CWEs. With only 5 CWEs represented in this test set, we encounter a significant constraint regarding the spectrum of vulnerabilities that can be examined. This limited coverage may compromise the comprehensive evaluation of the real-world performance of the models.

Furthermore, this dataset was automatically labelled CVE's fix commit, which does not guarantee the absolute reliability of the labels. There is the possibility of noise in the dataset which could result in misclassification or misinterpretation of the vulnerabilities, thereby affecting the accuracy of the model evaluation.

Absence of hyperparameter tuning: No specific hyperparameter tuning was conducted when training the vulnerability detection models and MLMs. This might result in suboptimal model configurations, potentially affecting the models' performance. Without thoroughly exploring the hyperparameter space, the models' ability to achieve their best performance may be compromised.

Evaluation strategy variability: Utilizing a more rigorous evaluation technique, such as k-fold cross-validation, instead of just train/test split, would provide a more stable and reliable estimate of each model's capabilities.

6.7 J-TAS requirements validation

Section 4.4 detailed a set of functional and non-functional requirements, outlining the critical characteristics and capabilities that the vulnerability detection tool should possess to be considered a viable solution. These requirements guided the development of the J-TAS Action, setting clear objectives to achieve. This section evaluates the extent to which the J-TAS Action successfully meets the requirements.

6.7.1 Functional Requirements

6.7.1.1 FR1: Detecting various types of vulnerabilities

The model incorporated into our tool can detect 21 distinct CWEs in Java source code. This reflects a wide variety of vulnerabilities and addresses the requirement for thorough vulnerability detection. Thus, **the tool successfully fulfils FR1** by offering broad-spectrum detection capabilities.

6.7.1.2 FR2: Scanning multiple java files

By default, the J-TAS Action scans all Java files within a repository. This functionality thoroughly assesses the entire codebase, thereby enhancing overall security. Additionally, the tool allows developers to specify individual files or directories for focused analysis, offering flexibility in its operation. Therefore, **the tool meets FR2** effectively.

6.7.1.3 FR3: Presenting results clearly

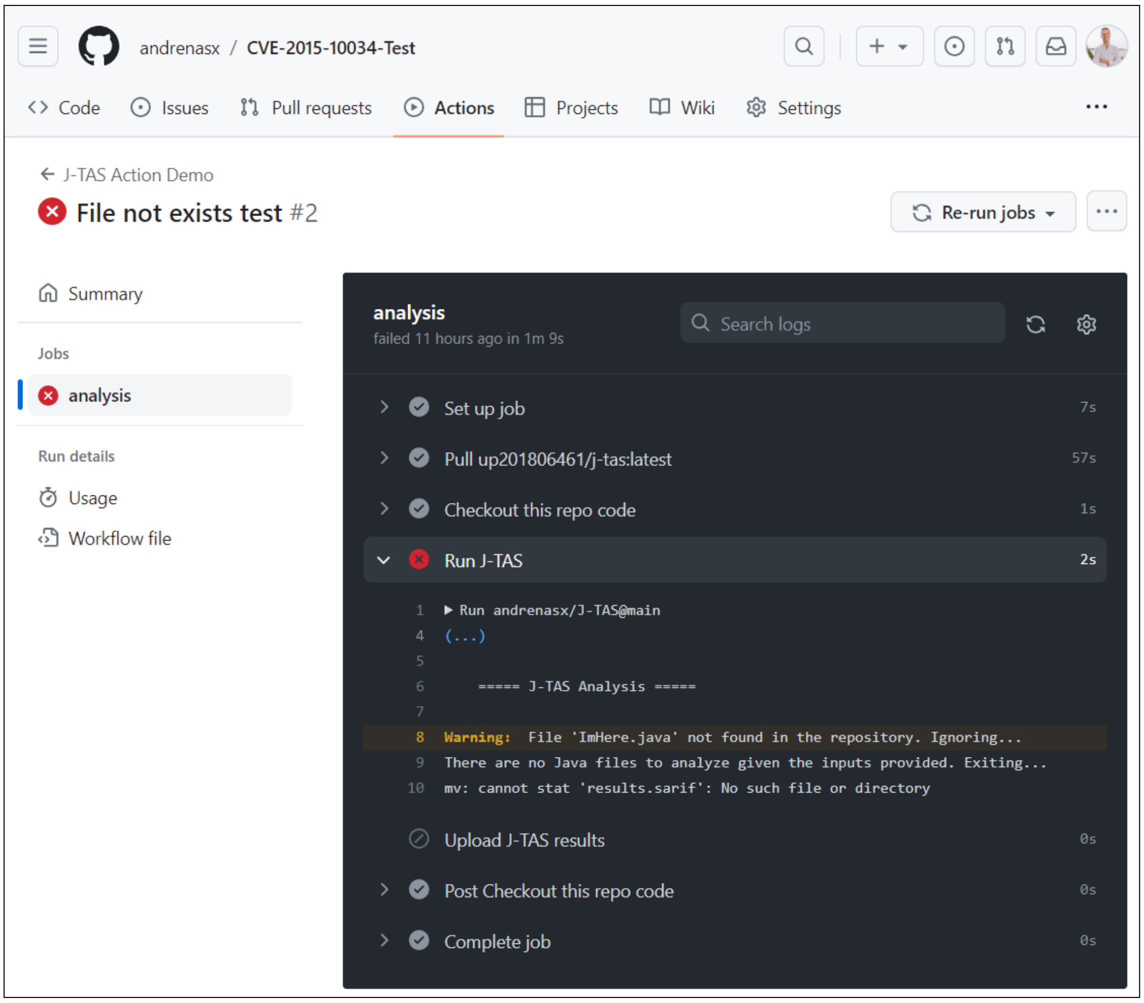
Our tool presents the analysis results in an accessible and intuitive manner, as shown in Figures 5.8 and 5.9. The outcomes are displayed under the Security tab of the respective GitHub repository. Users can effortlessly filter and navigate the vulnerabilities using the associated CWE-ID tag. Upon selecting a specific code alert, users are provided with the vulnerable method highlighted and the predicted percentage of the detected CWE, promoting transparency and understanding. Therefore, **the J-TAS Action meets the FR3** with a clear and concise result presentation.

6.7.1.4 FR4: Providing vulnerability explanations

Each code-scanning alert contains a description of the detected CWE. We also provide a link to the respective CWE page on the MITRE website to facilitate a deeper understanding. This allows developers to understand better the vulnerabilities found and how to address them. Therefore, **our tool successfully fulfils the FR4**.

6.7.1.5 FR5: Providing error feedback

The J-TAS Action is designed to handle input errors and provide constructive feedback. If the input files provided are either non-existent or are not Java files, or if the directories specified do not exist, the tool promptly issues an error message and aborts the analysis. Figure 6.1 shows the warning message issued by the tool when the input file does not exist. Similar messages are issued in the other cases as well. This feedback allows users to rectify their errors and helps in the seamless functioning of the tool, thus **successfully satisfying FR5**.



```
▶ Run andrenasx/J-TAS@main
(...)

    ===== J-TAS Analysis =====

Warning:  File 'ImHere.java' not found in the repository. Ignoring...
There are no Java files to analyze given the inputs provided. Exiting...
mv: cannot stat 'results.sarif': No such file or directory
```

Figure 6.1: J-TAS analysis results displayed in the code scanning tab.

6.7.2 Non-functional Requirements

6.7.2.1 NFR1: Fast execution time

Our tool, J-TAS Action, has been optimized for efficient execution. During our tests, the entire analysis process for 35 methods was completed in under a minute and a half. Notably, the analysis script's actual runtime accounted for only 22 seconds of this total time. Thus, we consider that **the J-TAS Action meets the NFR1**.

6.7.2.2 NFR2: Accuracy of the results

Achieving high accuracy in vulnerability detection is crucial for the effectiveness of our tool. Despite the ambitious target of over 80% accuracy, the current model employed in J-TAS Action achieves a modest accuracy of roughly 50% in the real-world scenario. Regarding the FPR and FNR metrics, we defined a 10% threshold for both. The tool performs well in the former, with mantaining a low FPR of under 5%. However, the FNR is significantly higher, with a value of 50%. The low FPR means the tool is cautious and minimizes the risk of false alarms. Unfortunately, **the tool fails to fulfil the NFR2 requirements of high accuracy and low FNR**.

6.7.2.3 NFR3: Adaptability to other programming languages

The J-TAS script was designed with a modular approach, which makes it highly adaptable to other programming languages. While the current version is Java-specific, the tool's design allows for the straightforward integration of new multiclass models for other languages. This adaptability enables a versatile and scalable solution, making **the tool effectively meet NFR3**.

Chapter 7

Conclusions and Future work

7.1 Conclusions

The expanding digital universe and the increasing sophistication of cyber threats have made software vulnerabilities a growing concern. Among the escalating threat landscape, the shift-left approach in the Software Development Life Cycle has emerged as an effective strategy, promoting proactive vulnerability detection in the early stages of software development. Simultaneously, the introduction of Deep Learning has revolutionized cybersecurity, offering significant advantages over traditional rule-based systems, such as enhanced pattern recognition capabilities, adaptability, and the ability to learn from vast, complex datasets.

In response to these trends, this dissertation embarked on an ambitious multidisciplinary journey to construct a Deep Learning-based tool for detecting software vulnerabilities. This process entailed constructing comprehensive datasets, training and evaluating Deep Learning models, and ultimately integrating the most promising model into a GitHub Action.

Our work started by creating the ND1 and ND2 datasets, meticulously curated to facilitate model training and evaluation. Afterwards, we created and trained two masked language modelling models that, alongside JavaBERT, served as the foundation for our vulnerability detection models, successfully exploring the impact of transfer learning. Finally, our efforts led to integrating our best-performing model into the J-TAS Action, a GitHub Action designed to facilitate vulnerability detection of Java code during development.

However, using synthetic datasets for training proved to be a substantial limitation. The synthetic nature limits their ability to accurately mimic real-world Java vulnerability patterns, ultimately impairing our models' performance and generalization capabilities. The shortcomings were manifested in the real-world scenario performance of the models and the GitHub Action, with a modest accuracy of roughly 50% and a significant false negative rate of 23%.

Despite these limitations, our work stands as a proof-of-concept for integrating a Deep Learning vulnerability detection model into a GitHub Action. It demonstrates that despite the complexities and challenges involved, such a tool can be built and provide value to developers. This dissertation, therefore, contributes a tangible artefact to the shift-left paradigm in SDLC and the broader effort to harness the power of Deep Learning for software security. It affirms that the combination of Deep Learning and DevOps can yield innovative solutions, even as it reveals the challenges and imperfections that must be addressed to fully realize the potential of such integrations.

7.2 Contributions

This dissertation's research results in the following contributions to the fields of software engineering, software security, and deep learning:

A curated vulnerability detection dataset derived from the Juliet Test Suite for Java. This dataset contains vulnerable and non-vulnerable Java methods and the corresponding CWE-ID, and is labelled accordingly for multi-label or multiclass classification.

An unique real-world Java test set enriched with both vulnerable Java code snippets and their corresponding fixes. This dataset, with CWE labeling, provides a valuable resource for future research and model development in the field of software vulnerability detection.

Two masked language models pre-trained on Java vulnerability-related code that provide a robust foundation for transfer learning to downstream tasks, such as vulnerability detection and bug-fixing patch generation.

Multi-label and multiclass Transformer-based vulnerability detection models trained on our dataset, and capable of detecting more than 20 different CWEs in Java methods.

The first Deep Learning-based GitHub Action for vulnerability detection , named J-TAS, representing a significant step towards automated and efficient software vulnerability detection.

7.3 Future work

The work undertaken in this dissertation has laid a strong foundation for creating a DL-based Action. However, the limitations identified throughout the process have highlighted areas for further investigation and improvement to make the tool viable for practical applications.

Given the synthetic character of the datasets used in this study, the first step in future research includes the creation of a real-world dataset. Real-world vulnerabilities exhibit more diverse and complicated patterns than synthetic data can capture, boosting the model's generalisation capabilities and performance improvements. This robust dataset could be created by mining GitHub commits associated with Java vulnerabilities.

The second area for exploration concerns model development. A new masked language model should be created, training it on more data from the Bug-Fix Pairs dataset, for a more rigorous evaluation of the effects of domain adaptation to a vulnerability-related Java corpus.

Following this, it would be necessary to develop new vulnerability detection models using the newly trained MLMs and train them on the real-world dataset. The process of creating these new models could also involve refining the model architecture and performing hyperparameter tuning to ensure optimal performance.

Lastly, future work should aim to improve the GitHub Action. This includes efforts to compress the image size, thereby enhancing the efficiency and usability of the Action. Furthermore, the Action might be improved by not just finding vulnerabilities but also suggesting possible fixes for them. By providing developers with potential solutions, the Action would serve as a more comprehensive vulnerability detection and resolution tool.

References

[1] *Advances and Applications in Deep Learning*. BoD – Books on Demand, December 2020. Google-Books-ID: 7a4tEAAAQBAJ.

[2] Daniel Beck, Gholamreza Haffari, and Trevor Cohn. Graph-to-Sequence Learning using Gated Graph Neural Networks. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 273–283, Melbourne, Australia, July 2018. Association for Computational Linguistics.

[3] Shai Ben-David, John Blitzer, Koby Crammer, and Fernando Pereira. Analysis of Representations for Domain Adaptation. In *Advances in Neural Information Processing Systems*, volume 19. MIT Press, 2006.

[4] Guru Bhandari, Amara Naseer, and Leon Moonen. CVEfixes: Automated Collection of Vulnerabilities and Their Fixes from Open-Source Software. In *Proceedings of the 17th International Conference on Predictive Models and Data Analytics in Software Engineering (PROMISE '21)*, page 10. ACM, 2021.

[5] Quang-Cuong Bui, Riccardo Scandariato, and Nicolás E. Díaz Ferreyra. Vul4J: A Dataset of Reproducible Java Vulnerabilities Geared Towards the Study of Program Repair Techniques. In *2022 IEEE/ACM 19th International Conference on Mining Software Repositories (MSR)*, pages 464–468, May 2022. ISSN: 2574-3864.

[6] Saikat Chakraborty, Rahul Krishna, Yangruibo Ding, and Baishakhi Ray. Deep Learning based Vulnerability Detection: Are We There Yet?, September 2020. arXiv:2009.07235 [cs].

[7] B. Chess and G. McGraw. Static analysis for security. *IEEE Security & Privacy*, 2(6):76–79, November 2004. Conference Name: IEEE Security & Privacy.

[8] Kyunghyun Cho, Bart van Merrienboer, Dzmitry Bahdanau, and Yoshua Bengio. On the Properties of Neural Machine Translation: Encoder-Decoder Approaches, October 2014. arXiv:1409.1259 [cs, stat].

[9] Roland Croft, M. Ali Babar, and Mehdi Kholoosi. Data Quality for Software Vulnerability Datasets, January 2023. arXiv:2301.05456 [cs].

[10] Roland Croft, Yongzheng Xie, and Muhammad Ali Babar. Data Preparation for Software Vulnerability Prediction: A Systematic Literature Review. *IEEE Transactions on Software Engineering*, pages 1–1, 2022. Conference Name: IEEE Transactions on Software Engineering.

[11] Debeshee Das, Noble Saji Mathews, and Sridhar Chimalakonda. Exploring Security Vulnerabilities in Competitive Programming: An Empirical Study. In *The International Conference*

on Evaluation and Assessment in Software Engineering 2022, pages 110–119, Gothenburg Sweden, June 2022. ACM.

[12] Nelson Tavares de Sousa and Wilhelm Hasselbring. JavaBERT: Training a transformer-based model for the Java programming language, October 2021. arXiv:2110.10404 [cs].

[13] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding, May 2019. arXiv:1810.04805 [cs].

[14] Meng Joo Er, Rajasekar Venkatesan, and Ning Wang. An online universal classifier for binary, multi-class and multi-label classification. In *2016 IEEE International Conference on Systems, Man, and Cybernetics (SMC)*, pages 003701–003706, October 2016.

[15] Hugging Face. Fine-tuning a masked language model - Hugging Face NLP Course. `https://huggingface.co/learn/nlp-course/chapter7/3`. Last accessed on 9th June 2023.

[16] Hugging Face. Performance and Scalability: How To Fit a Bigger Model and Train It Faster. `https://huggingface.co/docs/transformers/v4.13.0/en/performance`. Last accessed on 9th June 2023.

[17] Jiahao Fan, Yi Li, Shaohua Wang, and Tien N. Nguyen. A C/C++ Code Vulnerability Dataset with Code Changes and CVE Summaries. In *Proceedings of the 17th International Conference on Mining Software Repositories*, MSR '20, pages 508–512, New York, NY, USA, September 2020. Association for Computing Machinery.

[18] Wei Fang, Yupeng Chen, and Qiongying Xue. Survey on Research of RNN-Based Spatio-Temporal Sequence Prediction Algorithms. *Journal on Big Data*, 3(3):97–110, 2021.

[19] Michael Felderer, Matthias Büchler, Martin Johns, Achim D. Brucker, Ruth Breu, and Alexander Pretschner. Security Testing. In *Advances in Computers*, volume 101, pages 1–51. Elsevier, 2016.

[20] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. CodeBERT: A Pre-Trained Model for Programming and Natural Languages, September 2020. arXiv:2002.08155 [cs].

[21] Brian Fitzgerald and Klaas-Jan Stol. Continuous software engineering: A roadmap and agenda. *Journal of Systems and Software*, 123:176–189, January 2017.

[22] Michael Fu and Chakkrit Tantithamthavorn. LineVul: A Transformer-based Line-Level Vulnerability Prediction. In *2022 IEEE/ACM 19th International Conference on Mining Software Repositories (MSR)*, pages 608–620, May 2022. ISSN: 2574-3864.

[23] Felix A. Gers, Nicol N. Schraudolph, and Jürgen Schmidhuber. Learning precise timing with lstm recurrent networks. *The Journal of Machine Learning Research*, 3(null):115–143, March 2003.

[24] Seyed Mohammad Ghaffarian and Hamid Reza Shahriari. Software Vulnerability Analysis and Discovery Using Machine-Learning and Data-Mining Techniques: A Survey. *ACM Computing Surveys*, 50(4):56:1–56:36, August 2017.

[25] Afshin Gholamy, V. Kreinovich, and O. Kosheleva. Why 70/30 or 80/20 Relation Between Training and Testing Sets: A Pedagogical Explanation. 2018.

[26] Anthony Gillioz, Jacky Casas, Elena Mugellini, and Omar Abou Khaled. Overview of the Transformer-based Models for NLP Tasks. In *2020 15th Conference on Computer Science and Information Systems (FedCSIS)*, pages 179–183, September 2020.

[27] GitHub. Creating a Docker container action. https://docs.github.com/en/actions/creating-actions/creating-a-docker-container-action. Last accessed on 13th June 2023.

[28] GitHub. GitHub Marketplace · Actions to improve your workflow. https://github.com/marketplace?category=security&type=actions. Last accessed on 12th June 2023.

[29] GitHub. Understanding GitHub Actions. https://docs.github.com/en/actions/learn-github-actions/understanding-github-actions. Last accessed on 12th June 2023.

[30] GitHub. Uploading a SARIF file to GitHub. https://docs.github.com/en/code-security/code-scanning/integrating-with-code-scanning/uploading-a-sarif-file-to-github. Last accessed on 13th June 2023.

[31] GitHub. About GitHub Marketplace. https://docs.github.com/en/get-started/customizing-your-github-workflow/exploring-integrations/about-github-marketplace, Jan 2023. Last accessed on 30th January 2023.

[32] Ilya Grigorik. GH Archive. https://www.gharchive.org/. Last accessed on 9th June 2023.

[33] Fernando Gualo, Moisés Rodriguez, Javier Verdugo, Ismael Caballero, and Mario Piattini. Data quality certification using ISO/IEC 25012: Industrial experiences. *Journal of Systems and Software*, 176:110938, June 2021.

[34] Sepp Hochreiter and Jürgen Schmidhuber. Long Short-Term Memory. *Neural Computation*, 9(8):1735–1780, November 1997.

[35] J.J. Hopfield. Neural networks and physical systems with emergent collective computational abilities. January 1982.

[36] Seokmo Kim, R. Young Chul Kim, and Young B. Park. Software Vulnerability Detection Methodology Combined with Static and Dynamic Analysis. *Wireless Personal Communications*, 89(3):777–793, August 2016.

[37] Diederik P. Kingma and Jimmy Ba. Adam: A Method for Stochastic Optimization, January 2017. arXiv:1412.6980 [cs].

[38] Kiprono Elijah Koech. Cross-Entropy Loss Function. https://towardsdatascience.com/cross-entropy-loss-function-f38c4ec8643e, July 2022. Last accessed on 14th June 2023.

[39] Yujia Li, Daniel Tarlow, Marc Brockschmidt, and Richard Zemel. Gated Graph Sequence Neural Networks, September 2017. arXiv:1511.05493 [cs, stat].

[40] Zhen Li, Deqing Zou, Jing Tang, Zhihao Zhang, Mingqian Sun, and Hai Jin. A Comparative Study of Deep Learning-Based Vulnerability Detection System. *IEEE Access*, PP:1–1, July 2019.

[41] Zhen Li, Deqing Zou, Shouhuai Xu, Hai Jin, Yawei Zhu, and Zhaoxuan Chen. SySeVR: A Framework for Using Deep Learning to Detect Software Vulnerabilities. *IEEE Transactions on Dependable and Secure Computing*, 19(4):2244–2258, July 2022. arXiv:1807.06756 [cs, stat].

[42] Zhen Li, Deqing Zou, Shouhuai Xu, Xinyu Ou, Hai Jin, Sujuan Wang, Zhijun Deng, and Yuyi Zhong. VulDeePecker: A Deep Learning-Based System for Vulnerability Detection. In *Proceedings 2018 Network and Distributed System Security Symposium*, 2018. arXiv:1801.01681 [cs].

[43] Tianyang Lin, Yuxin Wang, Xiangyang Liu, and Xipeng Qiu. A survey of transformers. *AI Open*, 3:111–132, January 2022.

[44] Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. RoBERTa: A Robustly Optimized BERT Pretraining Approach, July 2019. arXiv:1907.11692 [cs].

[45] Ilya Loshchilov and Frank Hutter. Decoupled Weight Decay Regularization, January 2019. arXiv:1711.05101 [cs, math].

[46] Cláudia Raquel Botelho Sobral Mamede. *A Transformer-Based IDE Plugin for Vulnerability Detection*. PhD thesis, University of Porto, July 2022. Accepted: 2022-09-07T17:52:37Z.

[47] Chris McCormick. Smart Batching Tutorial - Speed Up BERT Training. https://mccormickml.com/2020/07/29/smart-batching-tutorial. Last accessed on 11th June 2023.

[48] Stuart Millar. Vulnerability Detection in Open Source Software: The Cure and the Cause. 2017.

[49] MITRE. FAQs | CVE. https://www.cve.org/ResourcesSupport/FAQs#pc_introcve_nvd_relationship, Jan 2023. Last accessed on 30th January 2023.

[50] Leon Moonen and Linas Vidziunas. CVEfixes Dataset: Automatically Collected Vulnerabilities and Their Fixes from Open-Source Software. https://zenodo.org/record/7029359, August 2022. Last accessed on 6th June 2023.

[51] NIST. NVD - Statistics. https://nvd.nist.gov/vuln/search/statistics?form_type=Basic&results_type=statistics&search_type=all&isCpeNameSearch=false, Jan 2023. Last accessed on 30th January 2023.

[52] NIST. NVD - Vulnerability Metrics. https://nvd.nist.gov/vuln-metrics/cvss, Jan 2023. Last accessed on 30th January 2023.

[53] OASIS. Static Analysis Results Interchange Format (SARIF) Version 2.1.0. 2020.

[54] Sinno Jialin Pan and Qiang Yang. A Survey on Transfer Learning. *IEEE Transactions on Knowledge and Data Engineering*, 22(10):1345–1359, October 2010. Conference Name: IEEE Transactions on Knowledge and Data Engineering.

[55] Samarjeet Patil. *Automated Vulnerability Detection in Java Source Code using J-CPG and Graph Neural Network*. PhD thesis, University of Twente, February 2021.

[56] Claudia Perlich. Learning Curves in Machine Learning. March 2009.

[57] Neil Perry, Megha Srivastava, Deepak Kumar, and Dan Boneh. Do Users Write More Insecure Code with AI Assistants?, December 2022. arXiv:2211.03622 [cs].

[58] Bala Pryia. Cross-Entropy Loss: Everything You Need to Know. `https://www.pinecone.io/learn/cross-entropy-loss/`. Last accessed on 11th June 2023.

[59] Roshan N. Rajapakse, Mansooreh Zahedi, M. Ali Babar, and Haifeng Shen. Challenges and solutions when adopting DevSecOps: A systematic review. *Information and Software Technology*, 141:106700, January 2022.

[60] Aya El-Rahman Kamal El-Deen Ramadan, Ahmed Bahaa, and Amr Ghoneim. A Systematic Literature Review on Software Vulnerability Detection Using Machine Learning Approaches. *FCI-H Informatics Bulletin*, 4(1):1–9, January 2022. Publisher: Helwan University, Faculty of Computers and Artificial Intelligence.

[61] François Role and Mohamed Nadif. HANDLING THE IMPACT OF LOW FREQUENCY EVENTS ON CO-OCCURRENCE BASED MEASURES OF WORD SIMILARITY - A Case Study of Pointwise Mutual Information:. In *Proceedings of the International Conference on Knowledge Discovery and Information Retrieval*, pages 226–231, Paris, France, 2011. SciTePress - Science and and Technology Publications.

[62] Victor Sanh, Lysandre Debut, Julien Chaumond, and Thomas Wolf. DistilBERT, a distilled version of BERT: smaller, faster, cheaper and lighter, February 2020. arXiv:1910.01108 [cs].

[63] Noam Shazeer and Mitchell Stern. Adafactor: Adaptive Learning Rates with Sublinear Memory Cost, April 2018. arXiv:1804.04235 [cs, stat].

[64] Ajay Shrestha and Ausif Mahmood. Review of Deep Learning Algorithms and Architectures. *IEEE Access*, 7:53040–53065, 2019. Conference Name: IEEE Access.

[65] Sima Siami-Namini, Neda Tavakoli, and Akbar Siami Namin. The Performance of LSTM and BiLSTM in Forecasting Time Series. In *2019 IEEE International Conference on Big Data (Big Data)*, pages 3285–3292, December 2019.

[66] Tim Sonnekalb, Thomas S. Heinze, and Patrick Mäder. Deep security analysis of program code: A systematic literature review. *Empirical Software Engineering*, 27(1):2, January 2022.

[67] Murugiah Souppaya, Karen Scarfone, and Donna Dodson. Secure Software Development Framework (SSDF) Version 1.1: Recommendations for Mitigating the Risk of Software Vulnerabilities. Technical Report NIST 800-218, National Institute of Standards and Technology, Gaithersburg, MD, February 2022.

[68] Ralf C. Staudemeyer and Eric Rothstein Morris. Understanding LSTM – a tutorial into Long Short-Term Memory Recurrent Neural Networks, September 2019. arXiv:1909.09586 [cs].

[69] Benjamin Steenhoek, Md Mahbubur Rahman, Richard Jiles, and Wei Le. An Empirical Study of Deep Learning Models for Vulnerability Detection, December 2022. arXiv:2212.08109 [cs].

[70] Martin Sundermeyer, Hermann Ney, and Ralf Schlüter. From Feedforward to Recurrent LSTM Neural Networks for Language Modeling. *IEEE/ACM Transactions on Audio, Speech, and Language Processing*, 23(3):517–529, March 2015. Conference Name: IEEE/ACM Transactions on Audio, Speech, and Language Processing.

[71] Karthik Chandra Swarna, Noble Saji Mathews, Dheeraj Vagavolu, and Sridhar Chimalakonda. A Mocktail of Source Code Representations, March 2022. arXiv:2106.10918 [cs].

[72] Tyler W. Thomas, Madiha Tabassum, Bill Chu, and Heather Lipford. Security During Application Development: an Application Security Expert Perspective. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*, pages 1–12, Montreal QC Canada, April 2018. ACM.

[73] Michele Tufano, Cody Watson, Gabriele Bavota, Massimiliano Di Penta, Martin White, and Denys Poshyvanyk. An Empirical Study on Learning Bug-Fixing Patches in the Wild via Neural Machine Translation, May 2019. arXiv:1812.08693 [cs].

[74] Deedra Vargo, Lin Zhu, Briana Benwell, and Zheng Yan. Digital technology use during COVID-19 pandemic: A rapid review. *Human Behavior and Emerging Technologies*, 3(1):13–24, 2021. _eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/hbe2.242.

[75] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention Is All You Need, December 2017. arXiv:1706.03762 [cs].

[76] Tom Viering and Marco Loog. The Shape of Learning Curves: A Review. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 45(6):7799–7819, June 2023. Conference Name: IEEE Transactions on Pattern Analysis and Machine Intelligence.

[77] Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Remi Louf, Morgan Funtowicz, Joe Davison, Sam Shleifer, Patrick von Platen, Clara Ma, Yacine Jernite, Julien Plu, Canwen Xu, Teven Le Scao, Sylvain Gugger, Mariama Drame, Quentin Lhoest, and Alexander Rush. Transformers: State-of-the-Art Natural Language Processing. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, pages 38–45, Online, October 2020. Association for Computational Linguistics.

[78] Bolun Wu and Futai Zou. Code Vulnerability Detection Based on Deep Sequence and Graph Models: A Survey. *Security and Communication Networks*, 2022:e1176898, October 2022. Publisher: Hindawi.

[79] Jiajie Wu. Literature review on vulnerability detection using NLP technology, April 2021. arXiv:2104.11230 [cs].

[80] Yonghui Wu, Mike Schuster, Zhifeng Chen, Quoc V. Le, Mohammad Norouzi, Wolfgang Macherey, Maxim Krikun, Yuan Cao, Qin Gao, Klaus Macherey, Jeff Klingner, Apurva Shah, Melvin Johnson, Xiaobing Liu, Łukasz Kaiser, Stephan Gouws, Yoshikiyo Kato, Taku Kudo, Hideto Kazawa, Keith Stevens, George Kurian, Nishant Patil, Wei Wang, Cliff Young, Jason Smith, Jason Riesa, Alex Rudnick, Oriol Vinyals, Greg Corrado, Macduff Hughes, and Jeffrey Dean. Google's Neural Machine Translation System: Bridging the Gap between Human and Machine Translation, October 2016. arXiv:1609.08144 [cs].

[81] Fabian Yamaguchi, Nico Golde, Daniel Arp, and Konrad Rieck. Modeling and Discovering Vulnerabilities with Code Property Graphs. In *2014 IEEE Symposium on Security and Privacy*, pages 590–604, May 2014. ISSN: 2375-1207.

[82] Shudong Yang, Xueying Yu, and Ying Zhou. LSTM and GRU Neural Network Performance Comparison Study: Taking Yelp Review Dataset as an Example. In *2020 International Workshop on Electronic Communication and Artificial Intelligence (IWECAI)*, pages 98–101, June 2020.

[83] Sakib Zargar. *Introduction to Sequence Learning Models: RNN, LSTM, GRU*. April 2021.

[84] Haibin Zhang, Yifei Bi, Hongzhi Guo, Wen Sun, and Jianpeng Li. ISVSF: Intelligent Vulnerability Detection Against Java via Sentence-Level Pattern Exploring. *IEEE Systems Journal*, 16(1):1032–1043, March 2022. Conference Name: IEEE Systems Journal.

[85] Min-Ling Zhang and Zhi-Hua Zhou. A Review On Multi-Label Learning Algorithms. *Knowledge and Data Engineering, IEEE Transactions on*, 26:1819–1837, August 2014.

[86] Wei Zheng, Jialiang Gao, Xiaoxue Wu, Fengyu Liu, Yuxing Xun, Guoliang Liu, and Xiang Chen. The impact factors on the performance of machine learning-based vulnerability detection: A comparative study. *Journal of Systems and Software*, 168:110659, October 2020.

[87] Yaqin Zhou, Shangqing Liu, Jingkai Siow, Xiaoning Du, and Yang Liu. Devign: Effective Vulnerability Identification by Learning Comprehensive Program Semantics via Graph Neural Networks, September 2019. arXiv:1909.03496 [cs, stat].

[88] Noah Ziems and Shaoen Wu. Security Vulnerability Detection Using Deep Learning Natural Language Processing, May 2021. arXiv:2105.02388 [cs].

[89] Deqing Zou, Sujuan Wang, Shouhuai Xu, Zhen Li, and Hai Jin. μVulDeePecker: A Deep Learning-Based System for Multiclass Vulnerability Detection. *IEEE Transactions on Dependable and Secure Computing*, 18(5):2224–2236, September 2021. Conference Name: IEEE Transactions on Dependable and Secure Computing.

Appendix A

Models' implementation details

A.1 Vulnerability detection models

A.1.1 Implementation code

```
# -*- coding: utf-8 -*-
"""vd_model_template.ipynb

Automatically generated by Colaboratory.

# J-TAS

Notebook for training and evaluating the models used for J-TAS

**Instructions:** Update the paths (datasets, any existing checkpoints, etc)
"""

!pip install transformers
!pip install sklearn
!pip install pandas
!pip install numpy

# Commented out IPython magic to ensure Python compatibility.
import numpy as np
import pandas as pd
import pickle
import transformers
import torch
import torch.nn as nn
import random
import time
from ast import literal_eval

from sklearn.metrics import accuracy_score, multilabel_confusion_matrix,
    ↪ confusion_matrix, classification_report, precision_recall_curve, roc_curve
from IPython.display import display
import matplotlib.pyplot as plt
# %matplotlib inline

from google.colab import drive
drive.mount('/content/drive')

DATA_PATH = '/content/drive/My Drive/J-TAS/data' #! Path with the train/val/test
    ↪ splits
MODEL_PATH = '/content/drive/My Drive/J-TAS/models' #! Path to save and load model
    ↪ checkpoints

model_checkpoint = 'up201806461/bert-java-bfp_single' #! BFP_single
                    # up201806461/bert-java-bfp_combined #! BFP_combined
                    # 'CAUKiel/JavaBERT' #! JavaBERT

suffix = #! Suffix for model name, e.g. '-nd2-bfp_single-multiclass'

#! Example for Multiclass, load label dict
label_names = [
    'Not Vuln',
    '15',
    '36',
    '78',
    '80',
    '89',
    '90',
    '113',
    '129',
    '134',
    '190',
    '191',
    '197',
    '319',
    '369',
    '400',
    '470',
    '476',
    '606',
    '643',
    '690',
    '789'
]

labels=[*range(len(label_names))]


```

```python
#! Example for Multilabel
# mlb = pickle.load(open(MODEL_PATH + 'mlb2_nodups.pkl', 'rb'))
# label_names = (mlb.classes_).astype(str)
# labels=[*range(len(label_names))]

"""# Utils functions
Mandatory to run
"""

EPOCHS = 10
BATCH_SIZE = 12
N_CLASSES = 22 #! Update according to dataset

tokenizer = transformers.AutoTokenizer.from_pretrained(model_checkpoint)

class VulnerabilityClassifier(nn.Module):
    DROPOUT_PROB = 0.1

    def __init__(self):
        super(VulnerabilityClassifier, self).__init__()
        self.model = transformers.AutoModel.from_pretrained(model_checkpoint)
        self.dropout = torch.nn.Dropout(self.DROPOUT_PROB)
        self.linear = torch.nn.Linear(768 * 4, N_CLASSES)
        self.step_scheduler_after = 'batch'


    def forward(self, ids, mask):
        cls_hs = torch.stack(self.model(ids, attention_mask=mask)["hidden_states"])

        cls_4hs = torch.cat((cls_hs[-1],
                             cls_hs[-2],
                             cls_hs[-3],
                             cls_hs[-4]), -1)[:, 0]

        output_dropout = self.dropout(cls_4hs)
        return self.linear(output_dropout)


def get_model():
  model = vulnerabilityClassifier()
  return model

def get_optimizer(model):
    opt = torch.optim.AdamW(model.parameters(),
                   lr = 5e-5,
                   eps = 1e-8
                 )

    return opt
```

```

def get_scheduler(optimizer, num_train_steps):
    sch = transformers.get_linear_schedule_with_warmup(
        optimizer, num_warmup_steps=0, num_training_steps=num_train_steps)
    return sch

import datetime

def format_time(elapsed):
    '''Takes a time in seconds and returns a string hh:mm:ss'''
    # Round to the nearest second.
    elapsed_rounded = int(round((elapsed)))

    # Format as hh:mm:ss
    return str(datetime.timedelta(seconds=elapsed_rounded))

## From: https://mccormickml.com/2020/07/29/smart-batching-tutorial/
def tokenize_truncate(tokenizer, text_samples, max_length):
    full_input_ids = []

    # For each training example...
    for text in text_samples:
        # Tokenize the sample.
        input_ids = tokenizer.encode(text=text,              # Text to encode.
                                    add_special_tokens=True, # Do add specials.
                                    max_length=max_length,      # Do Truncate!
                                    truncation=True,         # Do Truncate!
                                    padding=False)           # DO NOT pad.

        # Add the tokenized result to our list.
        full_input_ids.append(input_ids)

    print('DONE. {:>10,} samples\n'.format(len(full_input_ids)))
    return full_input_ids


def build_batches(samples, batch_size):
    # List of batches that we'll construct.
    batch_ordered_text = []
    batch_ordered_labels = []
    batch_original_order = []

    print('Creating batches of size {:}...'.format(batch_size))

    # Loop over all of the input samples...
    while len(samples) > 0:
        # `to_take` is our actual batch size. It will be `batch_size` until
        # we get to the last batch, which may be smaller.
        to_take = min(batch_size, len(samples))

        # Pick a random index in the list of remaining samples to start
        # our batch at.
        select = random.randint(0, len(samples) - to_take)

        # Select a contiguous batch of samples starting at `select`.
        batch = samples[select:(select + to_take)]

        #print("Batch length:", len(batch))

        # Each sample is a tuple--split them apart to create a separate list of
        # sequences and a list of labels for this batch.
        batch_ordered_text.append([s[0] for s in batch])
        batch_ordered_labels.append([s[1] for s in batch])
        batch_original_order.append([s[2] for s in batch])

        # Remove these samples from the list.
        del samples[select:select + to_take]

    print('\t  DONE - Selected {:,} batches.\n'.format(len(batch_ordered_text)))
    return batch_ordered_text, batch_ordered_labels, batch_original_order


def add_padding_per_batch(tokenizer, batch_ordered_text, batch_ordered_labels):
    print('Padding out sequences within each batch...')

    final_input_ids = []
    final_attention_masks = []
    final_labels = []

    # For each batch...
    for (batch_inputs, batch_labels) in zip(batch_ordered_text,
    ↪ batch_ordered_labels):

        # New version of the batch, this time with padded sequences and now with
        # attention masks defined.
        batch_padded_inputs = []
        batch_attn_masks = []

        # First, find the longest sample in the batch.
        # Note that the sequences do currently include the special tokens!
        max_size = max([len(sen) for sen in batch_inputs])

        # For each input in this batch...
        for sen in batch_inputs:

            # How many pad tokens do we need to add?
            num_pads = max_size - len(sen)

            # Add `num_pads` padding tokens to the end of the sequence.
            padded_input = sen + [tokenizer.pad_token_id]*num_pads

            # Define the attention mask--it's just a `1` for every real token
            # and a `0` for every padding token.
            attn_mask = [1] * len(sen) + [0] * num_pads

            # Add the padded results to the batch.
            batch_padded_inputs.append(padded_input)
            batch_attn_masks.append(attn_mask)

        # Our batch has been padded, so we need to save this updated batch.
        # We also need the inputs to be PyTorch tensors, so we'll do that here.
        final_input_ids.append(torch.tensor(batch_padded_inputs))
        final_attention_masks.append(torch.tensor(batch_attn_masks))
        final_labels.append(torch.tensor(np.array(batch_labels))) # if there's problems, remove np.array()

    print('\t DONE. Returning final smart-batched data.')
    # Return the smart-batched dataset!
    return (final_input_ids, final_attention_masks, final_labels)


def smart_batching(tokenizer, max_length, text_samples, labels, batch_size, return_order=False):
    # Tokenize and truncate text_samples; no padding
    full_input_ids = tokenize_truncate(tokenizer, text_samples, max_length)
    original_order = list(range(len(full_input_ids)))

    # Sort the two lists together by the length of the input sequence.
    samples = sorted(zip(full_input_ids, labels, original_order), key=lambda x: len(x[0]))

    # Build batches of contiguous data, starting at random points in samples
    batch_size = batch_size
    batch_ordered_text, batch_ordered_labels, batch_original_order = build_batches(samples, batch_size)

    # Add padding accordingly to batch size
    final_input_ids, final_attention_masks, final_labels = add_padding_per_batch(tokenizer, batch_ordered_text, batch_ordered_labels)

    if return_order:
      return final_input_ids, final_attention_masks, final_labels, [val for sublist in batch_original_order for val in sublist]

    return final_input_ids, final_attention_masks, final_labels

def loss_fn(outputs, labels):
    if labels is None:
        return None
    return nn.CrossEntropyLoss()(outputs, labels) #! Multiclass
    # return nn.BCEWithLogitsLoss()(outputs, labels.float()) #! Multi-label

#! Multiclass
def getAccuracy(preds, labels):
    y_pred = torch.stack(preds).cpu().detach().numpy()
    y_true = torch.stack(labels).cpu().detach().numpy()

    return accuracy_score(y_true, y_pred)

#! Multi-label
# def getAccuracy(preds, labels):
#     prob_preds = torch.stack(preds)
#     prob_preds = prob_preds.cpu().detach().numpy()
#     flabels = torch.stack(labels)
#     flabels = flabels.cpu().detach().numpy()

#     label_predictions = np.zeros((len(preds), N_CLASSES))
#     label_predictions = prob_preds >= 0.5
#     label_predictions = label_predictions.astype(int)

#     # accuracy_score from sklearn calculates subset accuracy
#     return accuracy_score(flabels, label_predictions)

"""Get Confusion Matrix and Classification Report"""

def get_cm_cr(y_true, y_pred):
  cm = multilabel_confusion_matrix(y_true, y_pred, labels=labels)
  cr = classification_report(y_true, y_pred, labels=labels, target_names=
    ↪ label_names)

  return cm, cr

"""Get metrics"""

def get_metrics(cm, y_true, y_pred):
  # accuracy_score from sklearn calculates subset accuracy
  ACC_subset = accuracy_score(y_true, y_pred)

  FNR_total = FPR_total = ACC_total = 0
  classes_metrics = []

  # Calculate metrics per class
  for idx, cm_class in enumerate(cm):
    TN, FP, FN, TP = cm_class.ravel()

    # False positive rate
    FPR = FP/(FP+TN)
    FPR_total += FPR

    # False negative rate
    FNR = 0
    if FN != 0:
      FNR = FN/(FN+TP)
    FNR_total += FNR

    # Accuracy
    ACC = (TP+TN)/(TP+FP+FN+TN)
    ACC_total += ACC

    classes_metrics.append([label_names[idx], ACC, TP, TN, FP, FN, FPR, FNR])


  # Dataframes to display results
  df_classes = pd.DataFrame(classes_metrics, columns=['Classes', 'Accuracy', 'TP',
    ↪ 'TN', 'FP', 'FN', 'FPR', 'FNR'])

  df_global = pd.DataFrame(
      [[ACC_total/N_CLASSES, ACC_subset, FNR_total/N_CLASSES, FPR_total/N_CLASSES
    ↪ ]],
      columns=['Accuracy', 'Subset Accuracy', 'Mean FNR', 'Mean FPR']
      )

  return df_classes, df_global

def train_fn(train_input_ids, train_attn_masks, train_labels, model, optimizer,
    ↪ scheduler):
    print('Starting training... ')

    update_interval = 500
    t0 = time.time()

    train_loss = 0.0
    model.train()

    final_targets = []
    final_outputs = []

    # for each batch
    for step in range(0, len(train_input_ids)):
        # Progress update every, e.g., 100 batches.
        if step % update_interval == 0 and not step == 0:
            # Calculate elapsed time in minutes.
            elapsed = format_time(time.time() - t0)

            # Calculate the time remaining based on our progress.
            steps_per_sec = (time.time() - t0) / step
            remaining_sec = steps_per_sec * (len(train_input_ids) - step)
            remaining = format_time(remaining_sec)

            # Report progress.
            print('  Batch {:>7,}  of  {:>7,}.    Elapsed: {:}.  Remaining: {:}'.format(step, len(train_input_ids), elapsed, remaining))

        ids = train_input_ids[step].to('cuda', dtype = torch.long)
        mask = train_attn_masks[step].to('cuda', dtype = torch.long)
        targets = train_labels[step].to('cuda', dtype = torch.long)

        optimizer.zero_grad()

        outputs = model(ids=ids, mask=mask)

        loss = loss_fn(outputs, targets)
        loss.backward()
        train_loss += loss.item()
        optimizer.step()
        scheduler.step()

        final_targets.extend(targets)
        final_outputs.extend(torch.argmax(outputs, dim=1)) #! Multiclass
        # final_outputs.extend(torch.sigmoid(outputs)) #! Multi-label

    return train_loss, final_outputs, final_targets

def eval_fn(test_input_ids, test_attn_masks, test_labels, model):
    print('\nStarting evaluation...')

    update_interval = 100
    t0 = time.time()

    eval_loss = 0.0

    model.eval()

    final_targets = []
    final_outputs = []

    with torch.no_grad():
       for step in range(0, len(test_input_ids)):
          if step % update_interval == 0 and not step == 0:
              # Calculate elapsed time in minutes.
              elapsed = format_time(time.time() - t0)

              # Calculate the time remaining based on our progress.
              steps_per_sec = (time.time() - t0) / step
              remaining_sec = steps_per_sec * (len(test_input_ids) - step)
              remaining = format_time(remaining_sec)
              # Report progress.
              print('  Batch {:>7,}  of  {:>7,}.    Elapsed: {:}.  Remaining: {:}'.format(step, len(test_input_ids), elapsed, remaining))

          ids = test_input_ids[step].to('cuda', dtype = torch.long)
          mask = test_attn_masks[step].to('cuda', dtype = torch.long)
          targets = test_labels[step].to('cuda', dtype = torch.long)

          outputs = model(ids=ids, mask=mask)

          loss = loss_fn(outputs, targets)

          eval_loss += loss.item()
          final_targets.extend(targets)
          final_outputs.extend(torch.argmax(outputs, dim=1)) #! Multiclass
          # final_outputs.extend(torch.sigmoid(outputs)) #! Multi-label

    return eval_loss, final_outputs, final_targets

def save_checkpoint(epoch, optimizer, scheduler, model, train_loss, test_loss):
  torch.save({
            'epoch': epoch,
            'model_state_dict': model.state_dict(),
            'optimizer_state_dict': optimizer.state_dict(),
            'scheduler_state_dict': scheduler.state_dict(),
            'train_loss': train_loss,
            'test_loss': test_loss
            }, MODEL_PATH + f'model{suffix}.bin')

  print('Saved hs_checkpoint_' + str(epoch) + '.bin')

"""# Training and Validation"""

#! Update splits files
train_dataset = pd.read_csv(DATA_PATH + 'train.csv')
val_dataset = pd.read_csv(DATA_PATH + 'val.csv')

train_input_ids, train_attn_masks, train_labels = smart_batching(tokenizer, 512, train_dataset['code'], train_dataset['encoded_labels'], BATCH_SIZE)
val_input_ids, val_attn_masks, val_labels = smart_batching(tokenizer, 512, val_dataset['code'], val_dataset['encoded_labels'], BATCH_SIZE)

n_train_steps = len(train_input_ids) * EPOCHS

model = get_model()
model.to('cuda')
```

```
optimizer = get_optimizer(model)
scheduler = get_scheduler(optimizer, n_train_steps)

LOAD = False

# Load checkpoint from a previous model
if LOAD:
  checkpoint = torch.load(MODEL_PATH + f'model{suffix}.bin')
  model.load_state_dict(checkpoint['model_state_dict'])
  optimizer.load_state_dict(checkpoint['optimizer_state_dict'])
  scheduler.load_state_dict(checkpoint['scheduler_state_dict'])
  best_eval_loss = checkpoint['test_loss']
  epoch = checkpoint['epoch'] + 1

  del checkpoint

else:
  epoch = 0

while epoch < EPOCHS:
  print('\t\t epoch: ', epoch)

  if epoch > 0: # This is not cross-validation!
    train_input_ids, train_attn_masks, train_labels = smart_batching(tokenizer,
    ↪ 512, train_dataset['code'], train_dataset['encoded_labels'], BATCH_SIZE)
    val_input_ids, val_attn_masks, val_labels = smart_batching(tokenizer, 512,
    ↪ val_dataset['code'], val_dataset['encoded_labels'], BATCH_SIZE)

  train_loss, train_preds, train_true_labels = train_fn(train_input_ids,
    ↪ train_attn_masks, train_labels, model, optimizer, scheduler)
  eval_loss, eval_preds, eval_true_labels = eval_fn(val_input_ids, val_attn_masks,
    ↪ val_labels, model)

  avg_train_loss, avg_val_loss = train_loss / len(train_input_ids), eval_loss / len
    ↪ (val_input_ids)
  train_acc = getAccuracy(train_preds, train_true_labels)
  eval_acc = getAccuracy(eval_preds, eval_true_labels)

  train_info = 'Avg Train loss (loss/batch): ' + str(avg_train_loss) + '\t Train
    ↪ accuracy: ' + str(train_acc) + '\n'
  val_info = 'Avg Valid loss (loss/batch): ' + str(avg_val_loss) + '\t Validation
    ↪ accuracy: ' + str(eval_acc) + '\n\n'

  f = open(MODEL_PATH + f'loss{suffix}.txt', 'a')
  f.write('Epoch ' + str(epoch) + '\n')
  f.write(train_info)
  f.write(val_info)
  print(train_info)
  print(val_info)
  f.close()

  scheduler.step()
  save_checkpoint(epoch, optimizer, scheduler, model, train_loss, eval_loss)
  epoch = epoch + 1

  # Free GPU memory
  del train_loss, train_preds, train_true_labels, eval_loss, eval_preds,
    ↪ eval_true_labels
  torch.cuda.empty_cache()

"""# Evaluation"""

model = get_model()
model.to('cuda')

# Load best model iteration for testing
epoch = 8 #! Update best epoch
checkpoint = torch.load(MODEL_PATH + f'model{suffix}-{epoch}.bin')
model.load_state_dict(checkpoint['model_state_dict'])

def evaluate(testfile):
  test_dataset = pd.read_csv(DATA_PATH + testfile)

  test_input_ids, test_attn_masks, test_labels = smart_batching(tokenizer, 512,
    ↪ test_dataset['code'], test_dataset['encoded_labels'], BATCH_SIZE)
  test_loss, test_preds, test_true_labels = eval_fn(test_input_ids, test_attn_masks
    ↪ , test_labels, model)

  y_true = torch.stack(test_true_labels).cpu().detach().numpy()
  y_pred = torch.stack(test_preds).cpu().detach().numpy()

  # Build confusion matrix and classification report
  cm, cr = get_cm_cr(y_true, y_pred)
  print("\n\n=== Confusion Matrix ===")
  print(cm)

  print("\n\n=== Classification report ===")
  print(cr)

  # Model metrics
  metrics_classes, metrics_global = get_metrics(cm, y_true, y_pred)
  print("\n\n=== Model metrics ===")
  display(metrics_global)

  print("\n\n=== Class metrics ===")
  display(metrics_classes)

"""## Test set"""
```

```

evaluate('test.csv') #! Update test set file

"""## Real-world evaluation set"""

evaluate('test_real.csv') #! Update file
```

Listing A.1: Baseline model architecture.

A.1.2 Training reports and Learning curves

A.1.2.1 *VDET-JavaBERT-Multilabel*

```
Epoch 1
Avg Train loss (loss/batch): 0.029430265961264763 Train accuracy: 0.8174711301414299
Avg Valid loss (loss/batch): 0.01971153401943296 Validation accuracy: 0.8609888687548537

Epoch 2
Avg Train loss (loss/batch): 0.020358640481468234 Train accuracy: 0.8437567579256953
Avg Valid loss (loss/batch): 0.018336528869329273 Validation accuracy: 0.8605574251445336

Epoch 3
Avg Train loss (loss/batch): 0.019557983876922175 Train accuracy: 0.8427619912633537
Avg Valid loss (loss/batch): 0.018268467139974334 Validation accuracy: 0.8580550522046768

Epoch 4
Avg Train loss (loss/batch): 0.018698066150274063 Train accuracy: 0.8452272825569829
Avg Valid loss (loss/batch): 0.017498718805008917 Validation accuracy: 0.8653033048580551

Epoch 5
Avg Train loss (loss/batch): 0.018140183374031824 Train accuracy: 0.8437026945201332
Avg Valid loss (loss/batch): 0.01685358148146252 Validation accuracy: 0.8669427905772715

Epoch 6
Avg Train loss (loss/batch): 0.017805336116949604 Train accuracy: 0.8435296916223347
Avg Valid loss (loss/batch): 0.016827491186383402 Validation accuracy: 0.8594356717577013

Epoch 7
Avg Train loss (loss/batch): 0.017407140307528796 Train accuracy: 0.8430647463345011
Avg Valid loss (loss/batch): 0.01716261771973343 Validation accuracy: 0.8653895935801191

Epoch 8
Avg Train loss (loss/batch): 0.017203546347089656 Train accuracy: 0.8396911898274296
Avg Valid loss (loss/batch): 0.01667831731882669 Validation accuracy: 0.865217016135991

Epoch 9
Avg Train loss (loss/batch): 0.016990605670517824 Train accuracy: 0.8377232818649712
Avg Valid loss (loss/batch): 0.016650922603261707 Validation accuracy: 0.8565018552075244

Epoch 10
Avg Train loss (loss/batch): 0.01683182198186074 Train accuracy: 0.8290515116128195
Avg Valid loss (loss/batch): 0.0165344875146158 Validation accuracy: 0.8565881439295884
```

Listing A.2: *VDET-JavaBERT-Multilabel* Training report

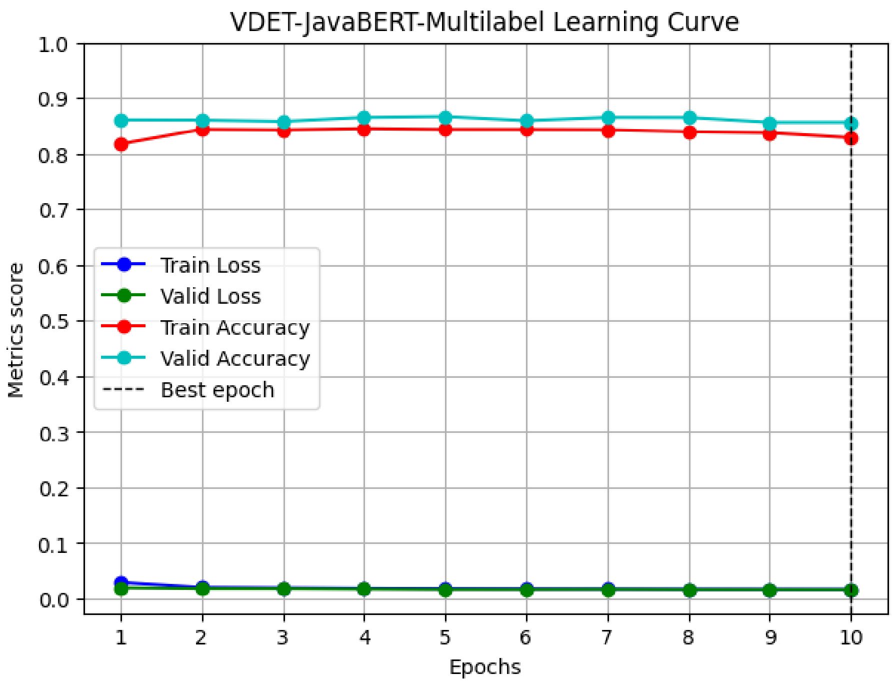


Figure A.1: *VDET-JavaBERT-Multilabel* Learning curve

A.1.2.2 *ND1-JavaBERT-Multilabel*

```
Epoch 1
Avg Train loss (loss/batch): 0.05446443316691834 Train accuracy: 0.6518617263004817
Avg Valid loss (loss/batch): 0.037559579240244834 Validation accuracy: 0.7229762481822588

Epoch 2
Avg Train loss (loss/batch): 0.033300098304653955 Train accuracy: 0.7439331050989184
Avg Valid loss (loss/batch): 0.029538177588811225 Validation accuracy: 0.7605428986912264

Epoch 3
Avg Train loss (loss/batch): 0.030627705773458393 Train accuracy: 0.7496894598115551
Avg Valid loss (loss/batch): 0.028478466907197502 Validation accuracy: 0.7603005332040718

Epoch 4
Avg Train loss (loss/batch): 0.030220905628128077 Train accuracy: 0.748083739812767
Avg Valid loss (loss/batch): 0.028187138230731403 Validation accuracy: 0.7569074163839069

Epoch 5
Avg Train loss (loss/batch): 0.028940753511017575 Train accuracy: 0.7496288666040537
Avg Valid loss (loss/batch): 0.02738313507018384 Validation accuracy: 0.7518177411536597

Epoch 6
Avg Train loss (loss/batch): 0.028157265993632087 Train accuracy: 0.749386493774048
Avg Valid loss (loss/batch): 0.027366053898805507 Validation accuracy: 0.7561803199224431

Epoch 7
Avg Train loss (loss/batch): 0.027588330938406146 Train accuracy: 0.7495076801890508
Avg Valid loss (loss/batch): 0.026879367552959763 Validation accuracy: 0.760785264178381

Epoch 8
Avg Train loss (loss/batch): 0.02715958476802453 Train accuracy: 0.7497803496228073
Avg Valid loss (loss/batch): 0.02660384410280638 Validation accuracy: 0.7559379544352884

Epoch 9
Avg Train loss (loss/batch): 0.026628291594778596 Train accuracy: 0.7492653073590451
Avg Valid loss (loss/batch): 0.026171195883862734 Validation accuracy: 0.7544837615123606

Epoch 10
Avg Train loss (loss/batch): 0.026338824854787397 Train accuracy: 0.7462659435877238
Avg Valid loss (loss/batch): 0.026149388427595198 Validation accuracy: 0.7520601066408144
```

Listing A.3: *ND1-JavaBERT-Multilabel* Training report

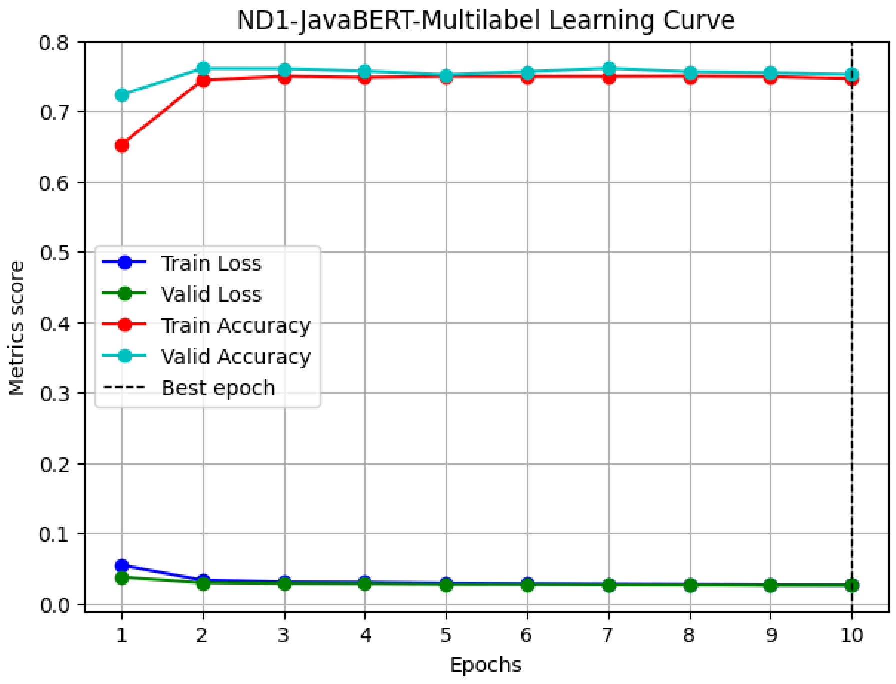


Figure A.2: *ND1-JavaBERT-Multilabel* Learning curve

A.1.2.3 *ND2-JavaBERT-Multilabel*

```
Epoch 1
Avg Train loss (loss/batch): 0.0870989736780973 Train accuracy: 0.47242119728592286
Avg Valid loss (loss/batch): 0.07399994556082636 Validation accuracy: 0.5158187800556822

Epoch 2
Avg Train loss (loss/batch): 0.0626118962689583 Train accuracy: 0.5776002748432535
Avg Valid loss (loss/batch): 0.06008086773880993 Validation accuracy: 0.5957985320172108

Epoch 3
Avg Train loss (loss/batch): 0.05393339112371233 Train accuracy: 0.6333075667783218
Avg Valid loss (loss/batch): 0.05341766482472246 Validation accuracy: 0.6511009870918755

Epoch 4
Avg Train loss (loss/batch): 0.04721169375676458 Train accuracy: 0.6754788284806321
Avg Valid loss (loss/batch): 0.05096292408812416 Validation accuracy: 0.687800556821058

Epoch 5
Avg Train loss (loss/batch): 0.040894855290828765 Train accuracy: 0.7163789401357038
Avg Valid loss (loss/batch): 0.0469009316022358 Validation accuracy: 0.6991900784611491

Epoch 6
Avg Train loss (loss/batch): 0.03504403924561207 Train accuracy: 0.7530876921755562
Avg Valid loss (loss/batch): 0.048269027594591056 Validation accuracy: 0.7248797772715768
```

```

Epoch 7
Avg Train loss (loss/batch): 0.02949740637633103 Train accuracy: 0.7893498239285408
Avg Valid loss (loss/batch): 0.04880702031440069 Validation accuracy: 0.7503163756011136

Epoch 8
Avg Train loss (loss/batch): 0.024120907499703036 Train accuracy: 0.8233788542471872
Avg Valid loss (loss/batch): 0.05383428973542814 Validation accuracy: 0.7637307010883321

Epoch 9
Avg Train loss (loss/batch): 0.018970805557908404 Train accuracy: 0.858335480546251
Avg Valid loss (loss/batch): 0.058679061239664314 Validation accuracy: 0.7749936724879777

Epoch 10
Avg Train loss (loss/batch): 0.015245732027187054 Train accuracy: 0.8809413381430903
Avg Valid loss (loss/batch): 0.0633686306441984 Validation accuracy: 0.7792963806631232

Epoch 11
Avg Train loss (loss/batch): 0.013637254231931741 Train accuracy: 0.8908528729708838
Avg Valid loss (loss/batch): 0.06338834745424782 Validation accuracy: 0.7792963806631232

Epoch 12
Avg Train loss (loss/batch): 0.013697190650242433 Train accuracy: 0.8898050330670789
Avg Valid loss (loss/batch): 0.06337681425098814 Validation accuracy: 0.7792963806631232

Epoch 13
Avg Train loss (loss/batch): 0.013661825912025726 Train accuracy: 0.8912136047410462
Avg Valid loss (loss/batch): 0.06336864499238129 Validation accuracy: 0.7792963806631232
```

Listing A.4: *ND2-JavaBERT-Multilabel* Training report

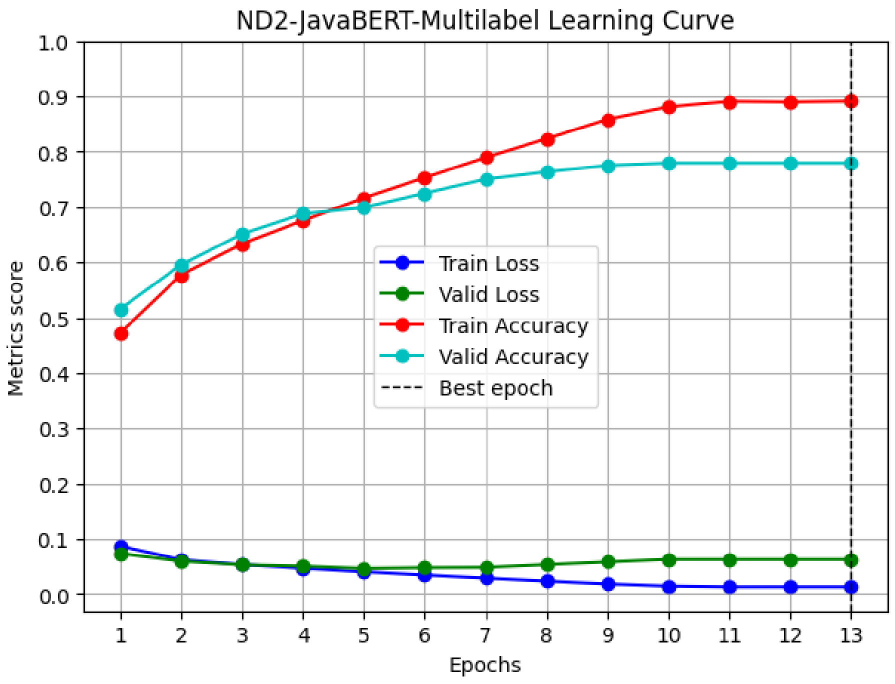


Figure A.3: *ND2-JavaBERT-Multilabel* Learning curve

A.1.2.4 *VDET-JavaBERT-Multiclass*

```
Epoch 1
Avg Train loss (loss/batch): 0.3123069933095296 Train accuracy: 0.8811470092124043
Avg Valid loss (loss/batch): 0.2039515762071215 Validation accuracy: 0.9049098282854431

Epoch 2
Avg Train loss (loss/batch): 0.20634921077646395 Train accuracy: 0.901301846805934
Avg Valid loss (loss/batch): 0.17588120936455426 Validation accuracy: 0.9045646733971869

Epoch 3
Avg Train loss (loss/batch): 0.18930732808098633 Train accuracy: 0.9040482678084858
Avg Valid loss (loss/batch): 0.17019518054081442 Validation accuracy: 0.9066356027267236

Epoch 4
Avg Train loss (loss/batch): 0.17849130166608568 Train accuracy: 0.9038644522295749
Avg Valid loss (loss/batch): 0.1858502643273346 Validation accuracy: 0.9027526102338425

Epoch 5
Avg Train loss (loss/batch): 0.17113227820251453 Train accuracy: 0.9052052246875135
Avg Valid loss (loss/batch): 0.17731698737393947 Validation accuracy: 0.9056864267840193

Epoch 6
Avg Train loss (loss/batch): 0.16490468613377274 Train accuracy: 0.906913628303274
Avg Valid loss (loss/batch): 0.15429003691605145 Validation accuracy: 0.908016222279748
```

```

Epoch 7
Avg Train loss (loss/batch): 0.16138131466546152 Train accuracy: 0.906913628303274
Avg Valid loss (loss/batch): 0.15258931535820858 Validation accuracy: 0.907671067391492

Epoch 8
Avg Train loss (loss/batch): 0.1575116053471737 Train accuracy: 0.9071082565632974
Avg Valid loss (loss/batch): 0.15334729046663065 Validation accuracy: 0.9084476658900682

Epoch 9
Avg Train loss (loss/batch): 0.15447621293023994 Train accuracy: 0.9083300895289996
Avg Valid loss (loss/batch): 0.14920923599281788 Validation accuracy: 0.907929933557684

Epoch 10
Avg Train loss (loss/batch): 0.15168723022946248 Train accuracy: 0.9079948964145149
Avg Valid loss (loss/batch): 0.1500950094051693 Validation accuracy: 0.9039606523427388
```

Listing A.5: *VDET-JavaBERT-Multiclass* Training report

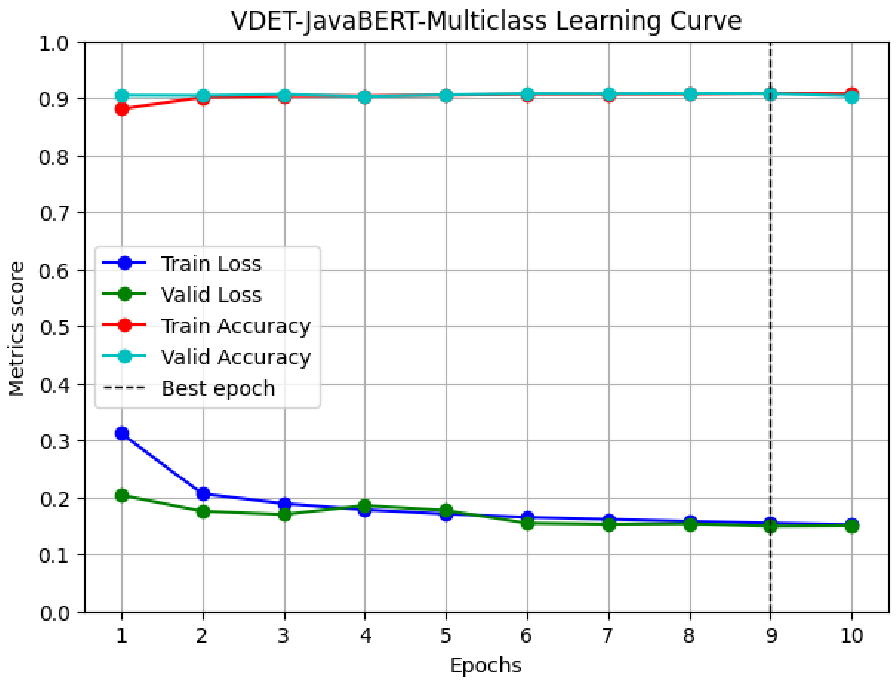


Figure A.4: *VDET-JavaBERT-Multiclass* Learning curve

A.1.2.5 *VDET-BFP_{single}-Multiclass*

```
Epoch 1
Avg Train loss (loss/batch): 0.2729854350781467 Train accuracy: 0.8872345486786903
Avg Valid loss (loss/batch): 0.20460859996319744 Validation accuracy: 0.9004228147381137
```

```

Epoch 2
Avg Train loss (loss/batch): 0.1977043803229348 Train accuracy: 0.9013234721681588
Avg Valid loss (loss/batch): 0.1736100115877961 Validation accuracy: 0.9060315816722755

Epoch 3
Avg Train loss (loss/batch): 0.18055635620250504 Train accuracy: 0.9033886942606288
Avg Valid loss (loss/batch): 0.17875021581004782 Validation accuracy: 0.9042195185089309

Epoch 4
Avg Train loss (loss/batch): 0.170687185672083 Train accuracy: 0.9064054322909909
Avg Valid loss (loss/batch): 0.15792357670301435 Validation accuracy: 0.9101734403313487

Epoch 5
Avg Train loss (loss/batch): 0.1658523434763561 Train accuracy: 0.9057782967864711
Avg Valid loss (loss/batch): 0.16255246947288507 Validation accuracy: 0.9054275606178273

Epoch 6
Avg Train loss (loss/batch): 0.16423138921486305 Train accuracy: 0.9055944812075603
Avg Valid loss (loss/batch): 0.15548002163816632 Validation accuracy: 0.907671067391492

Epoch 7
Avg Train loss (loss/batch): 0.15909966481508683 Train accuracy: 0.9062216167120799
Avg Valid loss (loss/batch): 0.15888718150386566 Validation accuracy: 0.9082750884459401

Epoch 8
Avg Train loss (loss/batch): 0.15690761842253909 Train accuracy: 0.9066541239565763
Avg Valid loss (loss/batch): 0.15520866681369666 Validation accuracy: 0.9061178703943394

Epoch 9
Avg Train loss (loss/batch): 0.154611916905393 Train accuracy: 0.906751438086588
Avg Valid loss (loss/batch): 0.1510558836881452 Validation accuracy: 0.904478384675123

Epoch 10
Avg Train loss (loss/batch): 0.1512562636269542 Train accuracy: 0.9075407638077938
Avg Valid loss (loss/batch): 0.15132980870895724 Validation accuracy: 0.9062904478384675
```

Listing A.6: *VDET-BFP$_{single}$-Multiclass* Training report

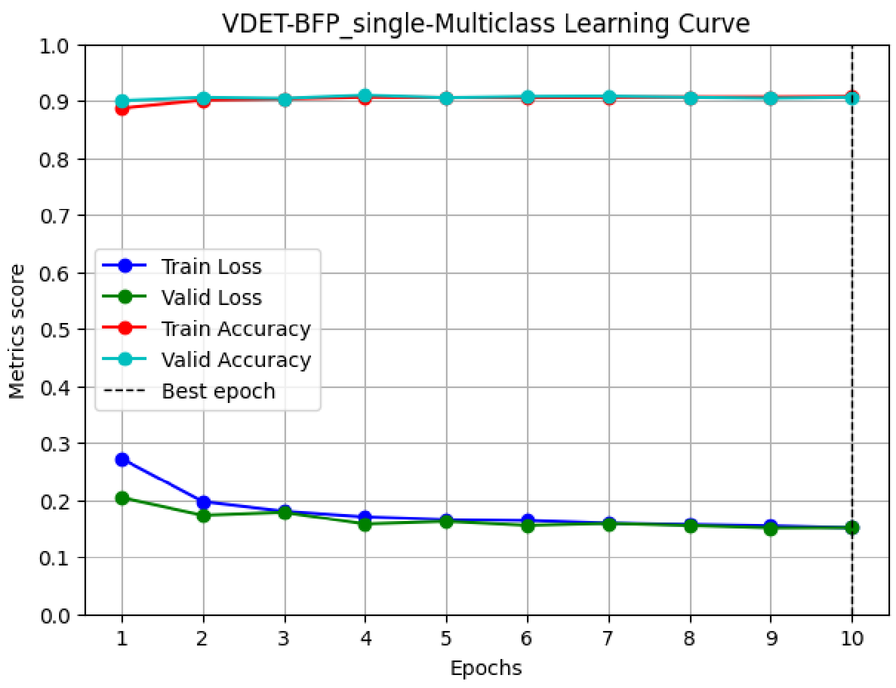


Figure A.5: *VDET-BFP_{single}-Multiclass* Learning curve

A.1.2.6 *VDET-$BFP_{combined}$-Multiclass*

```
Epoch 1
Avg Train loss (loss/batch): 0.2703578482736096 Train accuracy: 0.886564162449721
Avg Valid loss (loss/batch): 0.19647134148727893 Validation accuracy: 0.904305807230995

Epoch 2
Avg Train loss (loss/batch): 0.19189731155761533 Train accuracy: 0.9024155529605121
Avg Valid loss (loss/batch): 0.1753127043767689 Validation accuracy: 0.9054275606178273

Epoch 3
Avg Train loss (loss/batch): 0.18073624866930996 Train accuracy: 0.9039185156351369
Avg Valid loss (loss/batch): 0.1783358096771583 Validation accuracy: 0.9030114764000345

Epoch 4
Avg Train loss (loss/batch): 0.17460743074234072 Train accuracy: 0.905140348600839
Avg Valid loss (loss/batch): 0.16578429741775363 Validation accuracy: 0.9057727155060834

Epoch 5
Avg Train loss (loss/batch): 0.16728705363145302 Train accuracy: 0.9066757493188011
Avg Valid loss (loss/batch): 0.16071427733764887 Validation accuracy: 0.9039606523427388

Epoch 6
Avg Train loss (loss/batch): 0.16362066284556345 Train accuracy: 0.905853985554258
Avg Valid loss (loss/batch): 0.16637158134296573 Validation accuracy: 0.9082750884459401
```

```

Epoch 7
Avg Train loss (loss/batch): 0.16055799607730853 Train accuracy: 0.9064054322909909
Avg Valid loss (loss/batch): 0.1551283580198456 Validation accuracy: 0.908016222279748

Epoch 8
Avg Train loss (loss/batch): 0.1567544139061857 Train accuracy: 0.9077894554733792
Avg Valid loss (loss/batch): 0.15605571859791617 Validation accuracy: 0.9073259125032358

Epoch 9
Avg Train loss (loss/batch): 0.15432809572261644 Train accuracy: 0.9074542623588945
Avg Valid loss (loss/batch): 0.15228946956037878 Validation accuracy: 0.9063767365605315

Epoch 10
Avg Train loss (loss/batch): 0.15187282368134258 Train accuracy: 0.907043380476623
Avg Valid loss (loss/batch): 0.15040146221990777 Validation accuracy: 0.9084476658900682
```

Listing A.7: *VDET-BFP$_{combined}$-Multiclass* Training report

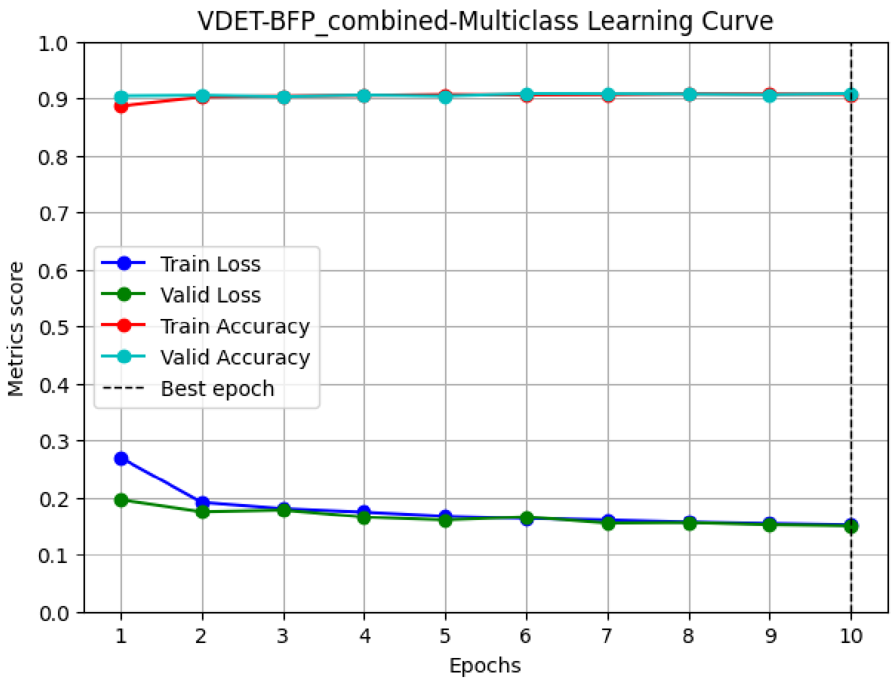


Figure A.6: *VDET-BFP$_{combined}$-Multiclass* Learning curve

A.1.2.7 *ND1-JavaBERT-Multiclass*

```
Epoch 1
Avg Train loss (loss/batch): 0.6009252641779577 Train accuracy: 0.8071393909984229
Avg Valid loss (loss/batch): 0.41948664225327414 Validation accuracy: 0.8420669577874818
```

```

Epoch 2
Avg Train loss (loss/batch): 0.38273029723229085 Train accuracy: 0.8549981802741721
Avg Valid loss (loss/batch): 0.34188250264298065 Validation accuracy: 0.858806404657933

Epoch 3
Avg Train loss (loss/batch): 0.34712747038698644 Train accuracy: 0.859850782482106
Avg Valid loss (loss/batch): 0.3320333314410206 Validation accuracy: 0.8600194080543425

Epoch 4
Avg Train loss (loss/batch): 0.33414046845044787 Train accuracy: 0.862792672570666
Avg Valid loss (loss/batch): 0.32882766250368994 Validation accuracy: 0.8634158175642892

Epoch 5
Avg Train loss (loss/batch): 0.3136754732928878 Train accuracy: 0.8648246997452383
Avg Valid loss (loss/batch): 0.33050261943700987 Validation accuracy: 0.859049005337215

Epoch 6
Avg Train loss (loss/batch): 0.3041422872950228 Train accuracy: 0.8682215212907922
Avg Valid loss (loss/batch): 0.32428466596794814 Validation accuracy: 0.861232411450752

Epoch 7
Avg Train loss (loss/batch): 0.2973475420429636 Train accuracy: 0.8659468640058231
Avg Valid loss (loss/batch): 0.3099052455055286 Validation accuracy: 0.8651140223192625

Epoch 8
Avg Train loss (loss/batch): 0.28734998501108594 Train accuracy: 0.8685854664563872
Avg Valid loss (loss/batch): 0.29713156852251943 Validation accuracy: 0.861475012130034

Epoch 9
Avg Train loss (loss/batch): 0.28183960390477336 Train accuracy: 0.8708601237413564
Avg Valid loss (loss/batch): 0.3049288120775737 Validation accuracy: 0.8537117903930131

Epoch 10
Avg Train loss (loss/batch): 0.27487471989612855 Train accuracy: 0.8730134659711271
Avg Valid loss (loss/batch): 0.30273606222345034 Validation accuracy: 0.8529839883551674
```

Listing A.8: *ND1-JavaBERT-Multiclass* Training report

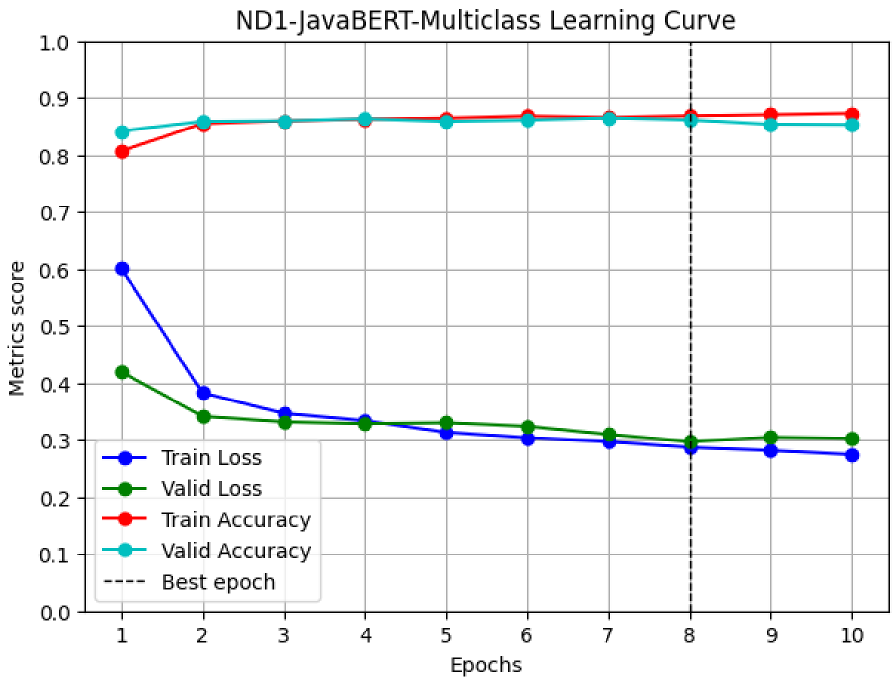


Figure A.7: *ND1-JavaBERT-Multiclass* Learning curve

A.1.2.8 *ND1-BFP_{single}-Multiclass*

```
Epoch 1
Avg Train loss (loss/batch): 0.5362765566528698 Train accuracy: 0.8171478830522868
Avg Valid loss (loss/batch): 0.3916887114379638 Validation accuracy: 0.8493449781659389

Epoch 2
Avg Train loss (loss/batch): 0.35323072364047564 Train accuracy: 0.8572121800315419
Avg Valid loss (loss/batch): 0.3272997380284556 Validation accuracy: 0.8651140223192625

Epoch 3
Avg Train loss (loss/batch): 0.3280856308374681 Train accuracy: 0.8639148368312508
Avg Valid loss (loss/batch): 0.31406609076111636 Validation accuracy: 0.8600194080543425

Epoch 4
Avg Train loss (loss/batch): 0.3118358722728376 Train accuracy: 0.8647337134538396
Avg Valid loss (loss/batch): 0.3130720978302465 Validation accuracy: 0.8558951965065502

Epoch 5
Avg Train loss (loss/batch): 0.3045640728717383 Train accuracy: 0.8651279873832343
Avg Valid loss (loss/batch): 0.3137459318051147 Validation accuracy: 0.8580786026200873

Epoch 6
Avg Train loss (loss/batch): 0.2981256240962485 Train accuracy: 0.8646730559262404
Avg Valid loss (loss/batch): 0.3010933291020698 Validation accuracy: 0.863658418243571
```

```

Epoch 7
Avg Train loss (loss/batch): 0.2893383954586877 Train accuracy: 0.8673723159044038
Avg Valid loss (loss/batch): 0.30012660219322135 Validation accuracy: 0.8597768073750607

Epoch 8
Avg Train loss (loss/batch): 0.2831655682278247 Train accuracy: 0.8676149460148005
Avg Valid loss (loss/batch): 0.29858367292559856 Validation accuracy: 0.8622028141678797

Epoch 9
Avg Train loss (loss/batch): 0.2788880489034225 Train accuracy: 0.8707691374499575
Avg Valid loss (loss/batch): 0.29487424127418055 Validation accuracy: 0.8561377971858322

Epoch 10
Avg Train loss (loss/batch): 0.2732096202882505 Train accuracy: 0.8725282057503336
Avg Valid loss (loss/batch): 0.29629185195575264 Validation accuracy: 0.8537117903930131
```

Listing A.9: *ND1-BFP_{single}-Multiclass* Training report

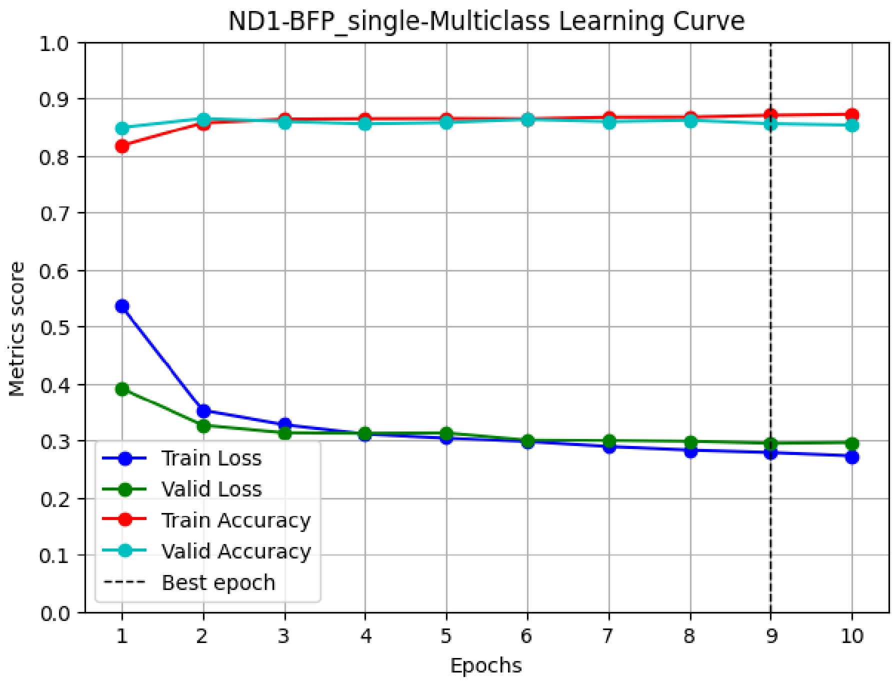


Figure A.8: *ND1-BFP_{single}-Multiclass* Learning curve

A.1.2.9 *ND1-$BFP_{combined}$-Multiclass*

```
Epoch 1
Avg Train loss (loss/batch): 0.5862617118949979 Train accuracy: 0.7971915564721582
Avg Valid loss (loss/batch): 0.38478542829567275 Validation accuracy: 0.844007763221737
```

```

Epoch 2
Avg Train loss (loss/batch): 0.36241588276204234 Train accuracy: 0.8545432488171782
Avg Valid loss (loss/batch): 0.34506995125445616 Validation accuracy: 0.8495875788452207

Epoch 3
Avg Train loss (loss/batch): 0.3447440083311298 Train accuracy: 0.856757248574548
Avg Valid loss (loss/batch): 0.327173315972771 Validation accuracy: 0.8568655992236778

Epoch 4
Avg Train loss (loss/batch): 0.32068374769746955 Train accuracy: 0.8621254397670751
Avg Valid loss (loss/batch): 0.3217948670212113 Validation accuracy: 0.861232411450752

Epoch 5
Avg Train loss (loss/batch): 0.30980105082191184 Train accuracy: 0.8644607545796433
Avg Valid loss (loss/batch): 0.30835478658066273 Validation accuracy: 0.859049005337215

Epoch 6
Avg Train loss (loss/batch): 0.29728647891357407 Train accuracy: 0.8663714666990173
Avg Valid loss (loss/batch): 0.3121096250565431 Validation accuracy: 0.8558951965065502

Epoch 7
Avg Train loss (loss/batch): 0.29410185023359686 Train accuracy: 0.8680395487079947
Avg Valid loss (loss/batch): 0.3042866687514698 Validation accuracy: 0.8634158175642892

Epoch 8
Avg Train loss (loss/batch): 0.28594717045299445 Train accuracy: 0.869192041732379
Avg Valid loss (loss/batch): 0.30943681369472964 Validation accuracy: 0.8578360019408054

Epoch 9
Avg Train loss (loss/batch): 0.28101176520155385 Train accuracy: 0.8681911925269926
Avg Valid loss (loss/batch): 0.29631546168804274 Validation accuracy: 0.861475012130034

Epoch 10
Avg Train loss (loss/batch): 0.27462038178403786 Train accuracy: 0.87352905495572
Avg Valid loss (loss/batch): 0.29573262786543847 Validation accuracy: 0.8556525958272683
```

Listing A.10: *ND1-BFP$_{combined}$-Multiclass* Training report

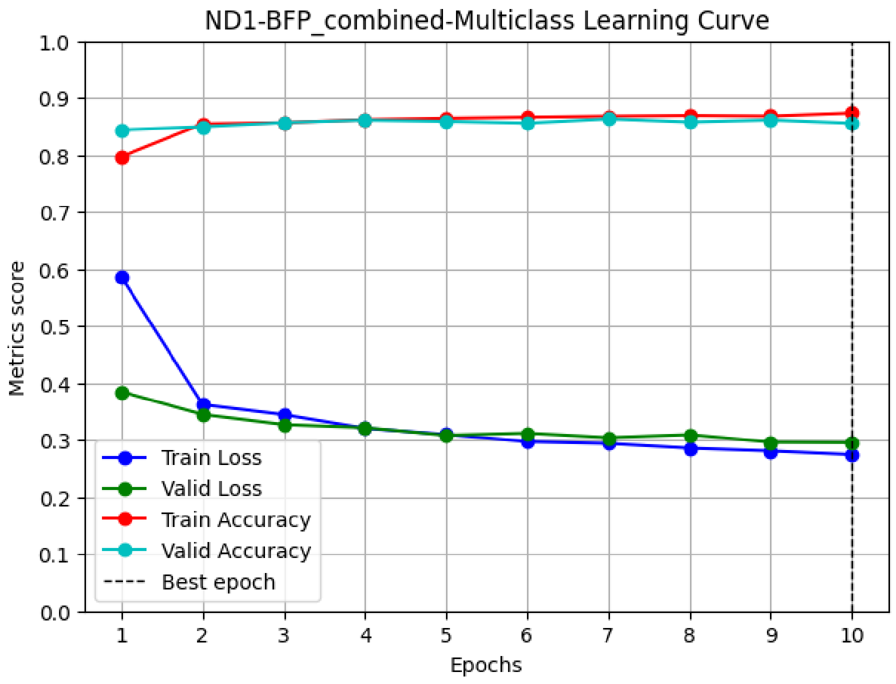


Figure A.9: *ND1-BFP$_{combined}$-Multiclass* Learning curve

A.1.2.10 *ND2-JavaBERT-Multiclass*

```
Epoch 1
Avg Train loss (loss/batch): 0.7946329905918267 Train accuracy: 0.7489306879670188
Avg Valid loss (loss/batch): 0.5782086035682857 Validation accuracy: 0.7913186535054416

Epoch 2
Avg Train loss (loss/batch): 0.5475441269055332 Train accuracy: 0.8042428927252426
Avg Valid loss (loss/batch): 0.49983635830321266 Validation accuracy: 0.8163756011136422

Epoch 3
Avg Train loss (loss/batch): 0.44208388309511876 Train accuracy: 0.8363136648630078
Avg Valid loss (loss/batch): 0.43406256447274666 Validation accuracy: 0.8262465198683877

Epoch 4
Avg Train loss (loss/batch): 0.3978763055464155 Train accuracy: 0.8470153740444902
Avg Valid loss (loss/batch): 0.39406650067202253 Validation accuracy: 0.8385218931915971

Epoch 5
Avg Train loss (loss/batch): 0.3581396752650794 Train accuracy: 0.8587477454264365
Avg Valid loss (loss/batch): 0.3859381464481569 Validation accuracy: 0.8394077448747153

Epoch 6
Avg Train loss (loss/batch): 0.3207529559343015 Train accuracy: 0.8687623464742764
Avg Valid loss (loss/batch): 0.3628896326122541 Validation accuracy: 0.8523158694001519
```

```

Epoch 7
Avg Train loss (loss/batch): 0.27860357825908616 Train accuracy: 0.8810444043631367
Avg Valid loss (loss/batch): 0.34773788829336466 Validation accuracy: 0.8642115920020248

Epoch 8
Avg Train loss (loss/batch): 0.2370314433051666 Train accuracy: 0.8948896332560337
Avg Valid loss (loss/batch): 0.35009393240135417 Validation accuracy: 0.8705391040242977

Epoch 9
Avg Train loss (loss/batch): 0.19370144660178326 Train accuracy: 0.9108992527699047
Avg Valid loss (loss/batch): 0.36745709379854125 Validation accuracy: 0.8762338648443432

Epoch 10
Avg Train loss (loss/batch): 0.15875654877131073 Train accuracy: 0.9254144120931032
Avg Valid loss (loss/batch): 0.38236236577840593 Validation accuracy: 0.8807896735003796
```

Listing A.11: *ND2-JavaBERT-Multiclass* Training report

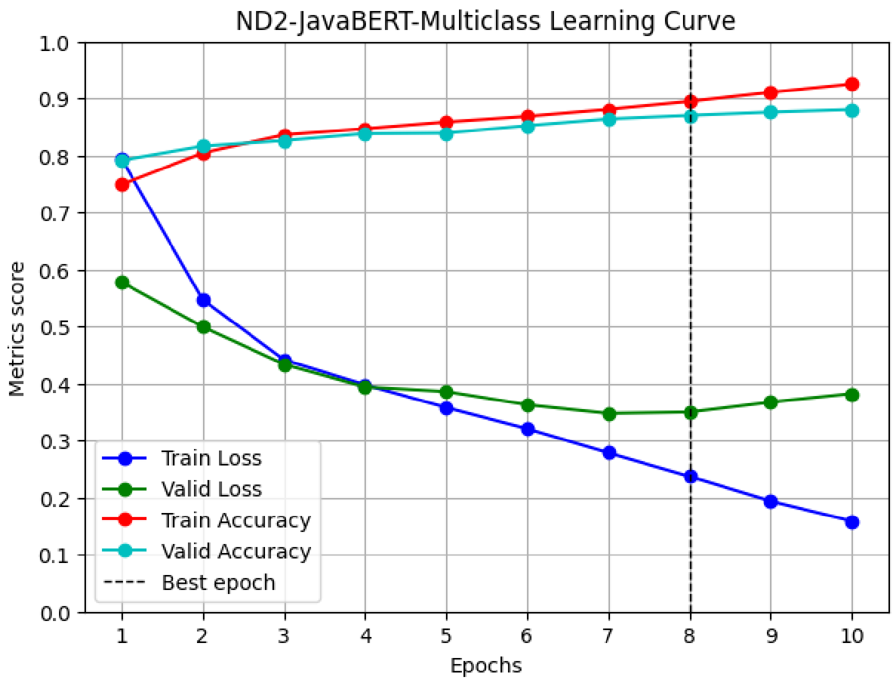


Figure A.10: *ND2-JavaBERT-Multiclass* Learning curve

A.1.2.11 *ND2-BFP_{single}-Multiclass*

```
Epoch 1
Avg Train loss (loss/batch): 0.6561974398825591 Train accuracy: 0.7839216696727648
Avg Valid loss (loss/batch): 0.5699631231973882 Validation accuracy: 0.7980258162490509
```

```

Epoch 2
Avg Train loss (loss/batch): 0.4882194047822764 Train accuracy: 0.8213003521429185
Avg Valid loss (loss/batch): 0.47460269201637 Validation accuracy: 0.8061250316375601

Epoch 3
Avg Train loss (loss/batch): 0.4345469607315833 Train accuracy: 0.8338400755818947
Avg Valid loss (loss/batch): 0.4281318059213884 Validation accuracy: 0.8277651227537333

Epoch 4
Avg Train loss (loss/batch): 0.3943938320213236 Train accuracy: 0.8433908786395259
Avg Valid loss (loss/batch): 0.41197584305908913 Validation accuracy: 0.837129840546697

Epoch 5
Avg Train loss (loss/batch): 0.36744172136027886 Train accuracy: 0.8513441552864383
Avg Valid loss (loss/batch): 0.3907608342906073 Validation accuracy: 0.8351050366995697

Epoch 6
Avg Train loss (loss/batch): 0.33276953621076444 Train accuracy: 0.8622863523146955
Avg Valid loss (loss/batch): 0.35037708438077725 Validation accuracy: 0.8554796254112883

Epoch 7
Avg Train loss (loss/batch): 0.2894577452339995 Train accuracy: 0.8762346474276389
Avg Valid loss (loss/batch): 0.33374046358749804 Validation accuracy: 0.865097443685143

Epoch 8
Avg Train loss (loss/batch): 0.24685923886449854 Train accuracy: 0.8915399811045264
Avg Valid loss (loss/batch): 0.3241609387683778 Validation accuracy: 0.8747152619589977

Epoch 9
Avg Train loss (loss/batch): 0.20613594923006756 Train accuracy: 0.9057287640642446
Avg Valid loss (loss/batch): 0.32704324322795764 Validation accuracy: 0.8773728170083523

Epoch 10
Avg Train loss (loss/batch): 0.1744251397371543 Train accuracy: 0.9192647942970025
Avg Valid loss (loss/batch): 0.33689859685686246 Validation accuracy: 0.880030372057707
```

Listing A.12: *ND2-BFP$_{single}$-Multiclass* Training report

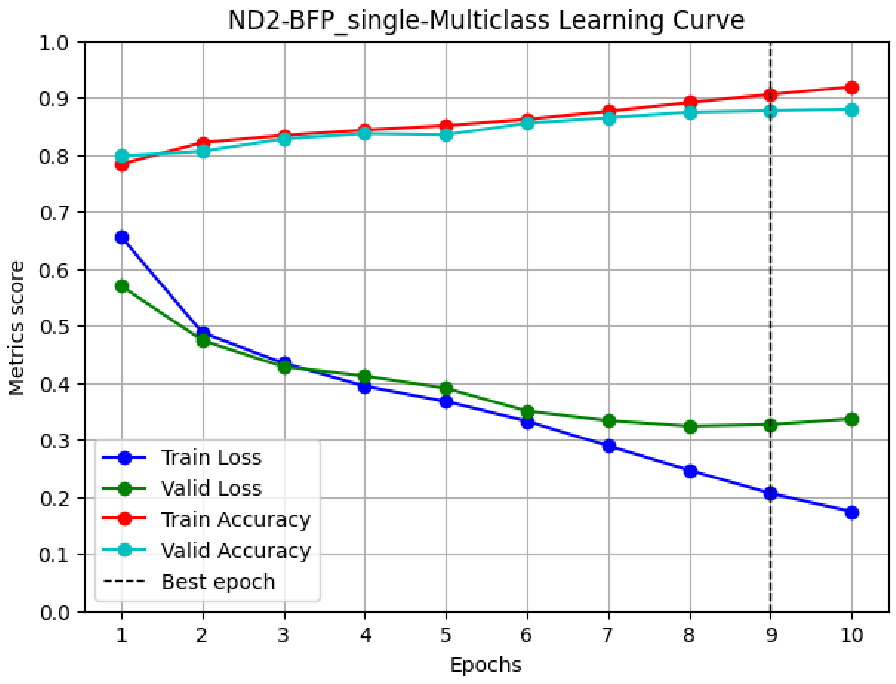


Figure A.11: *ND2-BFP$_{single}$-Multiclass* Learning curve

A.1.2.12 *ND2-BFP$_{combined}$-Multiclass*

```
Epoch 1
Avg Train loss (loss/batch): 0.6507730985571468 Train accuracy: 0.7827535858455724
Avg Valid loss (loss/batch): 0.4795037516346086 Validation accuracy: 0.8204252088078967

Epoch 2
Avg Train loss (loss/batch): 0.42777940802093745 Train accuracy: 0.8413810873486215
Avg Valid loss (loss/batch): 0.4179994826135146 Validation accuracy: 0.8399139458364971

Epoch 3
Avg Train loss (loss/batch): 0.3732959639928352 Train accuracy: 0.85589624667182
Avg Valid loss (loss/batch): 0.3699460963001254 Validation accuracy: 0.8539610225259427

Epoch 4
Avg Train loss (loss/batch): 0.32854976079662673 Train accuracy: 0.8675427295370609
Avg Valid loss (loss/batch): 0.3384247875091278 Validation accuracy: 0.8630726398380157

Epoch 5
Avg Train loss (loss/batch): 0.28372806848169435 Train accuracy: 0.8821953104869878
Avg Valid loss (loss/batch): 0.3393397659871833 Validation accuracy: 0.8716780561883067

Epoch 6
Avg Train loss (loss/batch): 0.23536991325803994 Train accuracy: 0.897157090097054
Avg Valid loss (loss/batch): 0.31447401119275364 Validation accuracy: 0.882814477347507
```

```

Epoch 7
Avg Train loss (loss/batch): 0.18211373201687517 Train accuracy: 0.9174267800395087
Avg Valid loss (loss/batch): 0.3382656138365341 Validation accuracy: 0.8867375348013161

Epoch 8
Avg Train loss (loss/batch): 0.14432577886858998 Train accuracy: 0.9311861204157004
Avg Valid loss (loss/batch): 0.3539722056606649 Validation accuracy: 0.8902809415337889

Epoch 9
Avg Train loss (loss/batch): 0.11260964741315593 Train accuracy: 0.9456325689255347
Avg Valid loss (loss/batch): 0.40437043996064187 Validation accuracy: 0.8968615540369527

Epoch 10
Avg Train loss (loss/batch): 0.09041992768840758 Train accuracy: 0.956849609207249
Avg Valid loss (loss/batch): 0.42275356996688773 Validation accuracy: 0.8986332574031891
```

Listing A.13: *ND2-BFP*$_{combined}$*-Multiclass* Training report

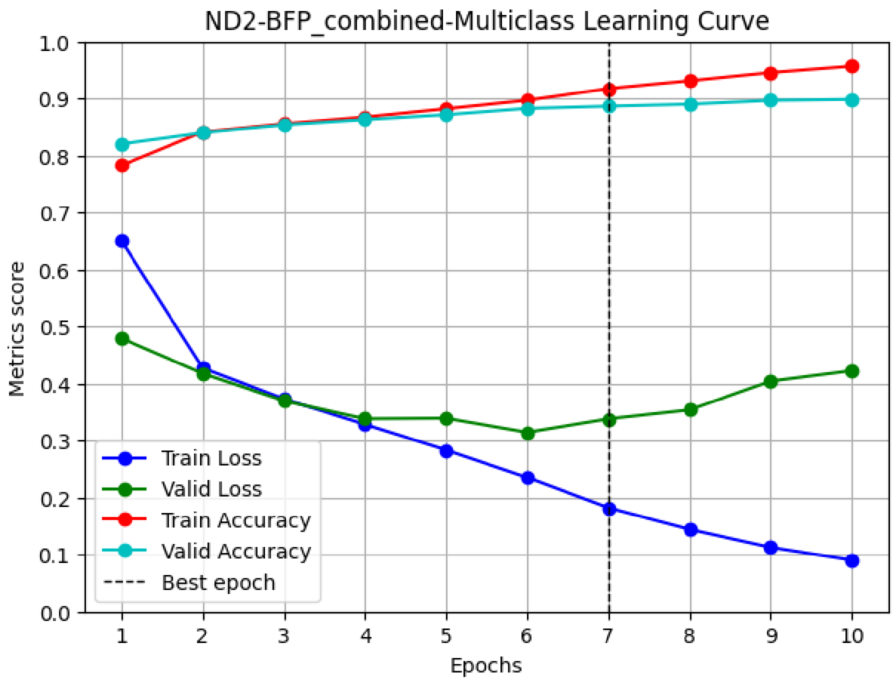


Figure A.12: *ND2-BFP*$_{combined}$*-Multiclass* Learning curve

A.2 Masked Language Models

A.2.1 Implementation code

```
# -*- coding: utf-8 -*-
"""mlm_template.ipynb

Automatically generated by Colaboratory.

# Masked Language Modeling

**Instructions:** Update the paths (datasets, any existing checkpoints, etc)

**Notebook References:**
*   [Bug-Fix Pairs Datasets](https://sites.google.com/view/learning-fixes/data)
*   [HF Fine-tuning a masked language model](https://huggingface.co/course/chapter7
    ↪ /3)
*   [HF Masked language modeling](https://huggingface.co/docs/transformers/main/
    ↪ tasks/masked_language_modeling)
"""

!pip install datasets
!pip install transformers
!pip install torch

from google.colab import drive
drive.mount('/content/drive')

DATA_PATH = '/content/drive/My Drive/MLM/data/'
MODEL_PATH = '/content/drive/My Drive/MLM/models/'

"""# Define functions

## Model checkpoint
"""

model_checkpoint = 'bert-base-cased'

"""## Load Dataset"""

from datasets import load_dataset

def my_load_dataset(folder):
  dataset = load_dataset('csv', data_files={'train': DATA_PATH + folder + '/train.
    ↪ csv', 'test': DATA_PATH + folder + '/test.csv'})
  print(dataset)

  return dataset

"""## Create model"""

from transformers import AutoModelForMaskedLM

```

```python
def get_model(model_checkpoint_):
  model = AutoModelForMaskedLM.from_pretrained(model_checkpoint_)
  model.cuda()

  return model

"""## Preprocessing the data"""

from transformers import AutoTokenizer

def get_tokenizer(model_checkpoint_):
  return AutoTokenizer.from_pretrained(model_checkpoint_)

def tokenize_function(sample):
  result = tokenizer(sample["code"])
  if tokenizer.is_fast:
      result["word_ids"] = [result.word_ids(i) for i in range(len(result["input_ids
    ↪ "]))]
  return result

def get_tokenized_dataset(dataset_, remove_cols=['id', 'code', 'buggy', 'filename'
    ↪ ]):
  tokenized_dataset = dataset_.map(
    tokenize_function, batched=True, num_proc=4, remove_columns=remove_cols
  )

  return tokenized_dataset

"""Concatenate all the code samples and split the concatenated samples into shorter
    ↪  chunks defined by chunk_size, which should be both shorter than the maximum
    ↪  input length and short enough for your GPU RAM."""

chunk_size = 256

def group_texts(examples):
  # Concatenate all texts
  concatenated_examples = {k: sum(examples[k], []) for k in examples.keys()}

  # Compute length of concatenated texts
  total_length = len(concatenated_examples[list(examples.keys())[0]])

  # We drop the last chunk if it's smaller than chunk_size
  total_length = (total_length // chunk_size) * chunk_size

  # Split by chunks of max_len
  result = {
    k: [t[i : i + chunk_size] for i in range(0, total_length, chunk_size)]
    for k, t in concatenated_examples.items()
  }

  # Create a new labels column
  result["labels"] = result["input_ids"].copy()
  return result

"""Note that in the last step of group_texts() we create a new labels column which
    ↪ is a copy of the input_ids one. That's because in masked language modeling
    ↪ the objective is to predict randomly masked tokens in the input batch, and
    ↪ by creating a labels column we provide the ground truth for our language
    ↪ model to learn from.

Now all we need to do is insert randomly [MASK] tokens and train the model!

## Fine-tuning with the Trainer API

Define training hyperparameters and Trainer class
"""

from transformers import DataCollatorForLanguageModeling
from transformers import TrainingArguments, Trainer
from IPython.display import display

def get_args():
  return TrainingArguments(
    output_dir=MODEL_PATH,
    overwrite_output_dir=True,
    evaluation_strategy='epoch',
    save_strategy='epoch',
    per_device_train_batch_size=32,
    per_device_eval_batch_size=32,
    fp16=True,
    optim="adamw_torch",
    learning_rate=2e-5,
    num_train_epochs=5,
    weight_decay=0.01,
    seed=42
  )

def get_trainer(model_, tokenizer_, args_, dataset_, data_collator_):
  return Trainer(
    model=model_,
    tokenizer=tokenizer_,
    args=args_,
    train_dataset=dataset_["train"],
    eval_dataset=dataset_["test"],
    data_collator=data_collator_,
  )

"""Evaluate on eval set"""
```

```

import math

def evaluate_model(trainer_):
  eval_results = trainer_.evaluate()
  print(f"Perplexity: {math.exp(eval_results['eval_loss']):.2f}")

"""# Train Models"""

model_name = 'combined'
# model_name = 'single'

dataset = my_load_dataset(model_name)
dataset

tokenizer = get_tokenizer(model_checkpoint)
tokenized_dataset = get_tokenized_dataset(dataset)
del dataset

tokenized_dataset

lm_dataset = tokenized_dataset.map(group_texts, batched=True, num_proc=4)
del tokenized_dataset

lm_dataset

trainer = get_trainer(
    get_model(model_checkpoint),
    tokenizer,
    get_args(),
    lm_dataset,
    DataCollatorForLanguageModeling(tokenizer=tokenizer, mlm_probability=0.15)
)

display(trainer.train())

evaluate_model(trainer)

trainer.save_model(MODEL_PATH + model_name)
```

Listing A.14: Baseline model architecture.

A.2.2 Training reports

```
Epoch 1
Training loss: 0.803400
Validation loss: 0.697756
```

```

Epoch 2
Training loss: 0.674300
Validation loss: 0.610899

Epoch 3
Training loss: 0.618300
Validation loss: 0.577367

Epoch 4
Training loss: 0.595700
Validation loss: 0.553462

Epoch 5
Training loss: 0.581900
Validation loss: 0.546494
Evaluation perplexity: 1.73
```

Listing A.15: $BFP_{combined}$ MLM training report

```
Epoch 1
Training loss: 0.558400
Validation loss: 0.494941

Epoch 2
Training loss: 0.476100
Validation loss: 0.433146

Epoch 3
Training loss: 0.441100
Validation loss: 0.403015

Epoch 4
Training loss: 0.422200
Validation loss: 0.393980

Epoch 5
Training loss: 0.421100
Validation loss: 0.388629
Evaluation perplexity: 1.48
```

Listing A.16: BFP_{single} MLM training report

Appendix B

J-TAS Action instructions

```
This action analysis Java files and generates a report in [Static Analysis Results Interchange Format (SARIF) format](https://www.oasis−open.org/standard/sarif−v2−1−0/). The results can be seen in the ∗Security tab∗ of your repository.

We recommend using the [actions/checkout](https://github.com/marketplace/actions/checkout) action to check out your repository, and [github/codeql−action/upload−sarif](https://github.com/github/codeql−action/tree/main/upload−sarif) to upload the SARIF file. For more information on their usage, check the respective READMEs.

## Usage

<!−− start usage −−>
‘‘‘yaml
− uses: andrenasx/J−TAS@main
  with:
    # Paths to the directories containing the Java source files to analyze.
    # These paths are relative to the root of the repository, and separated by spaces when multiple paths are provided.
    # Example: ’src/main/java src/test/java’
    # Default: ’’
    paths: ’’

    # Paths of the the Java source files to analyze.
    # These paths are relative to the root of the repository, and separated by spaces when multiple paths are provided.
    # Example: ’src/main/java/example/HelloWorld.java src/test/java/example/HelloWorldTest.java’
    # Default: ’’
    files: ’’
‘‘‘
<!−− end usage −−>

When no ‘paths‘ nor ‘files‘ are provided, the action will analyze all Java files in the repository.

## Workflow examples

```

~~~
- [Analyse the whole repository](#Analyse-the-repository-on-every-push)
- [Analyse specific files](#Analyse-specific-files)
- [Analyse specific directories](#Analyse-specific-directories)
- [Analyse only the changed files in current commit](#Analyse-only-the-changed-files)

### Analyse the repository on every push

```yaml
on: [push]
name: J-TAS analysis

jobs:
  jtas-analysis:
    runs-on: ubuntu-latest

    steps:
      - name: Checkout this repo code
        uses: actions/checkout@v3

      - name: Run J-TAS
        uses: andrenasx/J-TAS@main

      - name: Upload J-TAS report
        uses: github/codeql-action/upload-sarif@v2
        with:
          sarif_file: results.sarif
          category: my-analysis-tool
~~~

Analyse specific files

```yaml
on: [push]
name: J-TAS analysis

jobs:
  jtas-analysis:
    runs-on: ubuntu-latest

    steps:
      - name: Checkout this repo code
        uses: actions/checkout@v3

      - name: Run J-TAS
        uses: andrenasx/J-TAS@main
        with:
          files: 'src/main/java/com/example/HelloWorld.java src/test/java/com/example/HelloWorldTest.java'

      - name: Upload J-TAS report
```

~~~
        uses: github/codeql-action/upload-sarif@v2
        with:
          sarif_file: results.sarif
          category: my-analysis-tool
```

### Analyse specific directories

```yaml
on: [push]
name: J-TAS analysis

jobs:
  jtas-analysis:
    runs-on: ubuntu-latest

    steps:
      - name: Checkout this repo code
        uses: actions/checkout@v3

      - name: Run J-TAS
        uses: andrenasx/J-TAS@main
        with:
          paths: 'src/main/java/com/controller src/main/java/com/service'

      - name: Upload J-TAS report
        uses: github/codeql-action/upload-sarif@v2
        with:
          sarif_file: results.sarif
          category: my-analysis-tool
```

### Analyse only the changed files

```yaml
on: [push]
name: J-TAS analysis

jobs:
  jtas-analysis:
    runs-on: ubuntu-latest

    steps:
      - name: Checkout repository code
        uses: actions/checkout@v3
        with:
          fetch-depth: 2

      - name: Process files changed in the current commit
~~~

```
        id: diff
        run: |
          changedFiles=$(git diff --name-only HEAD^)
          echo "files=${changedFiles//$'\n'/ }" >> "$GITHUB_OUTPUT"

      - name: Run J-TAS
        uses: andrenasx/J-TAS@main
        with:
          files: ${{ steps.diff.outputs.files }}

      - name: Upload J-TAS report
        uses: github/codeql-action/upload-sarif@v2
        with:
          sarif_file: results.sarif
          category: my-analysis-tool
```
‘‘‘

Listing B.1: *J-TAS README.md file.*

Appendix C

Results details

C.1 *VDET-JavaBERT-Multilabel*

Table C.1: *VDET-JavaBERT-Multilabel* test set's class metrics.

Class	**TP**	**TN**	**FP**	**FN**	**FPR**	**FNR**
113	693	10804	5	25	0.000463	0.034819
129	1162	10271	0	94	0.000000	0.074841
134	364	11138	0	25	0.000000	0.064267
15	157	11357	0	13	0.000000	0.076471
190	2117	9269	43	98	0.004618	0.044244
191	1594	9807	0	126	0.000000	0.073256
197	388	11110	0	29	0.000000	0.069544
23	53	11459	0	15	0.000000	0.220588
319	174	11353	0	0	0.000000	0.000000
36	92	11423	0	12	0.000000	0.115385
369	878	10609	0	40	0.000000	0.043573
400	700	10827	0	0	0.000000	0.000000
470	158	11367	0	2	0.000000	0.012500
606	207	11306	0	14	0.000000	0.063348
643	218	11294	0	15	0.000000	0.064378
690	153	11373	0	1	0.000000	0.006494
78	141	11377	0	9	0.000000	0.060000
789	472	11022	6	27	0.000544	0.054108
80	227	11284	1	15	0.000089	0.061983
89	919	10543	10	55	0.000948	0.056468
90	37	11482	0	8	0.000000	0.177778
False	7573	2956	459	539	0.134407	0.066445
True	2960	7575	537	455	0.066198	0.133236

Table C.2: *VDET-JavaBERT-Multilabel* test set's classification report.

	Precision	**Recall**	**F1-score**	**Support**
113	0.99	0.97	0.98	718
129	1.00	0.93	0.96	1256
134	1.00	0.94	0.97	389
15	1.00	0.92	0.96	170
190	0.98	0.96	0.97	2215
191	1.00	0.93	0.96	1720
197	1.00	0.93	0.96	417
23	1.00	0.78	0.88	68
319	1.00	1.00	1.00	174
36	1.00	0.88	0.94	104
369	1.00	0.96	0.98	918
400	1.00	1.00	1.00	700
470	1.00	0.99	0.99	160
606	1.00	0.94	0.97	221
643	1.00	0.94	0.97	233
690	1.00	0.99	1.00	154
78	1.00	0.94	0.97	150
789	0.99	0.95	0.97	499
80	1.00	0.94	0.97	242
89	0.99	0.94	0.97	974
90	1.00	0.82	0.90	45
False	0.94	0.93	0.94	8112
True	0.85	0.87	0.86	3415
micro avg	0.95	0.93	0.94	23054
macro avg	0.99	0.93	0.96	23054
weighted avg	0.95	0.93	0.94	23054
samples avg	0.95	0.93	0.94	23054

Table C.3: *VDET-JavaBERT-Multilabel* real-world test set's class metrics.

Class	TP	TN	FP	FN	FPR	FNR
113	0	585	18	6	0.029851	1.000000
129	0	606	3	0	0.004926	0.000000
134	0	588	21	0	0.034483	0.000000
15	0	601	8	0	0.013136	0.000000
190	0	592	17	0	0.027915	0.000000
191	0	584	25	0	0.041051	0.000000
197	0	606	3	0	0.004926	0.000000
23	0	603	6	0	0.009852	0.000000
319	0	550	59	0	0.096880	0.000000
36	0	609	0	0	0.000000	0.000000
369	0	596	13	0	0.021346	0.000000
400	2	482	63	62	0.115596	0.968750
470	0	608	1	0	0.001642	0.000000
606	0	598	11	0	0.018062	0.000000
643	0	604	5	0	0.008210	0.000000
690	0	593	16	0	0.026273	0.000000
78	0	540	49	20	0.083192	1.000000
789	0	511	98	0	0.160920	0.000000
80	0	601	8	0	0.013136	0.000000
89	31	93	1	484	0.010638	0.939806
90	0	591	14	4	0.023140	1.000000
False	193	127	130	159	0.505837	0.451705
True	124	201	151	133	0.428977	0.517510

Table C.4: *VDET-JavaBERT-Multilabel* real-world test set's classification report.

	Precision	**Recall**	**F1-score**	**Support**
113	0.00	0.00	0.00	6
129	0.00	0.00	0.00	0
134	0.00	0.00	0.00	0
15	0.00	0.00	0.00	0
190	0.00	0.00	0.00	0
191	0.00	0.00	0.00	0
197	0.00	0.00	0.00	0
23	0.00	0.00	0.00	0
319	0.00	0.00	0.00	0
36	0.00	0.00	0.00	0
369	0.00	0.00	0.00	0
400	0.03	0.03	0.03	64
470	0.00	0.00	0.00	0
606	0.00	0.00	0.00	0
643	0.00	0.00	0.00	0
690	0.00	0.00	0.00	0
78	0.00	0.00	0.00	20
789	0.00	0.00	0.00	0
80	0.00	0.00	0.00	0
89	0.97	0.06	0.11	515
90	0.00	0.00	0.00	4
False	0.60	0.55	0.57	352
True	0.45	0.48	0.47	257
micro avg	0.33	0.29	0.31	1218
macro avg	0.09	0.05	0.05	1218
weighted avg	0.68	0.29	0.31	1218
samples avg	0.33	0.29	0.30	1218

C.2 *ND1-JavaBERT-Multilabel*

Table C.5: *ND1-JavaBERT-Multilabel* test set's class metrics.

Class	TP	TN	FP	FN	FPR	FNR
113	219	3878	0	29	0.000000	0.116935
129	268	3789	0	69	0.000000	0.204748
134	109	3981	0	36	0.000000	0.248276
15	32	4073	0	21	0.000000	0.396226
190	710	3295	51	70	0.015242	0.089744
191	558	3455	18	95	0.005183	0.145482
197	94	4017	0	15	0.000000	0.137615
319	63	4060	0	3	0.000000	0.045455
36	20	4082	0	24	0.000000	0.545455
369	262	3803	0	61	0.000000	0.188854
400	210	3916	0	0	0.000000	0.000000
470	42	4064	0	20	0.000000	0.322581
476	86	4039	0	1	0.000000	0.011494
563	68	4056	0	2	0.000000	0.028571
606	62	4018	0	46	0.000000	0.425926
643	46	4035	0	45	0.000000	0.494505
690	96	4013	0	17	0.000000	0.150442
78	36	4067	0	23	0.000000	0.389831
789	118	3979	3	26	0.000753	0.180556
80	50	4055	4	17	0.000985	0.253731
81	21	4084	0	21	0.000000	0.500000
83	18	4084	0	24	0.000000	0.571429
89	166	3894	0	66	0.000000	0.284483
90	18	4085	0	23	0.000000	0.560976
False	2259	1647	104	116	0.059395	0.048842
True	1651	2258	117	100	0.049263	0.057110

Table C.6: *ND1-JavaBERT-Multilabel* test set's classification report.

	Precision	**Recall**	**F1-score**	**Support**
113	1.00	0.88	0.94	248
129	1.00	0.80	0.89	337
134	1.00	0.75	0.86	145
15	1.00	0.60	0.75	53
190	0.93	0.91	0.92	780
191	0.97	0.85	0.91	653
197	1.00	0.86	0.93	109
319	1.00	0.95	0.98	66
36	1.00	0.45	0.62	44
369	1.00	0.81	0.90	323
400	1.00	1.00	1.00	210
470	1.00	0.68	0.81	62
476	1.00	0.99	0.99	87
563	1.00	0.97	0.99	70
606	1.00	0.57	0.73	108
643	1.00	0.51	0.67	91
690	1.00	0.85	0.92	113
78	1.00	0.61	0.76	59
789	0.98	0.82	0.89	144
80	0.93	0.75	0.83	67
81	1.00	0.50	0.67	42
83	1.00	0.43	0.60	42
89	1.00	0.72	0.83	232
90	1.00	0.44	0.61	41
False	0.96	0.95	0.95	2375
True	0.93	0.94	0.94	1751
micro avg	0.96	0.88	0.92	8252
macro avg	0.99	0.75	0.84	8252
weighted avg	0.96	0.88	0.92	8252
samples avg	0.96	0.88	0.91	8252

Table C.7: *ND1-JavaBERT-Multilabel* real-world test set's class metrics.

Class	**TP**	**TN**	**FP**	**FN**	**FPR**	**FNR**
113	2	601	2	4	0.003317	0.666667
129	0	606	3	0	0.004926	0.000000
134	0	576	33	0	0.054187	0.000000
15	0	609	0	0	0.000000	0.000000
190	0	603	6	0	0.009852	0.000000
191	0	581	28	0	0.045977	0.000000
197	0	609	0	0	0.000000	0.000000
319	0	559	50	0	0.082102	0.000000
36	0	609	0	0	0.000000	0.000000
369	0	601	8	0	0.013136	0.000000
400	0	485	60	64	0.110092	1.000000
470	0	599	10	0	0.016420	0.000000
476	0	580	29	0	0.047619	0.000000
563	0	605	4	0	0.006568	0.000000
606	0	609	0	0	0.000000	0.000000
643	0	602	7	0	0.011494	0.000000
690	0	547	62	0	0.101806	0.000000
78	3	589	0	17	0.000000	0.850000
789	0	591	18	0	0.029557	0.000000
80	0	607	2	0	0.003284	0.000000
81	0	597	12	0	0.019704	0.000000
83	0	609	0	0	0.000000	0.000000
89	35	94	0	480	0.000000	0.932039
90	0	605	0	4	0.000000	1.000000
False	259	50	207	93	0.805447	0.264205
True	58	259	93	199	0.264205	0.774319

Table C.8: *ND1-JavaBERT-Multilabel* real-world test set's classification report.

	Precision	**Recall**	**F1-score**	**Support**
113	0.50	0.33	0.40	6
129	0.00	0.00	0.00	0
134	0.00	0.00	0.00	0
15	0.00	0.00	0.00	0
190	0.00	0.00	0.00	0
191	0.00	0.00	0.00	0
197	0.00	0.00	0.00	0
319	0.00	0.00	0.00	0
36	0.00	0.00	0.00	0
369	0.00	0.00	0.00	0
400	0.00	0.00	0.00	64
470	0.00	0.00	0.00	0
476	0.00	0.00	0.00	0
563	0.00	0.00	0.00	0
606	0.00	0.00	0.00	0
643	0.00	0.00	0.00	0
690	0.00	0.00	0.00	0
78	1.00	0.15	0.26	20
789	0.00	0.00	0.00	0
80	0.00	0.00	0.00	0
81	0.00	0.00	0.00	0
83	0.00	0.00	0.00	0
89	1.00	0.07	0.13	515
90	0.00	0.00	0.00	4
False	0.56	0.74	0.63	352
True	0.38	0.23	0.28	257
micro avg	0.36	0.29	0.32	1218
macro avg	0.13	0.06	0.07	1218
weighted avg	0.68	0.29	0.30	1218
samples avg	0.39	0.29	0.32	1218

C.3 *ND2-JavaBERT-Multilabel*

Table C.9: *ND2-JavaBERT-Multilabel* test set's class metrics.

Class	**TP**	**TN**	**FP**	**FN**	**FPR**	**FNR**
113	394	7569	28	73	0.003686	0.156317
129	608	7182	114	160	0.015625	0.208333
134	236	7736	28	64	0.003606	0.213333
15	72	7968	5	19	0.000627	0.208791
190	1394	6256	180	234	0.027968	0.143735
191	1127	6577	149	211	0.022153	0.157698
197	201	7795	33	35	0.004216	0.148305
319	102	7961	0	1	0.000000	0.009709
36	95	7934	9	26	0.001133	0.214876
369	527	7394	49	94	0.006583	0.151369
400	369	7695	0	0	0.000000	0.000000
470	73	7951	10	30	0.001256	0.291262
476	111	7951	0	2	0.000000	0.017699
606	99	7893	20	52	0.002527	0.344371
643	125	7877	16	46	0.002027	0.269006
690	156	7879	11	18	0.001394	0.103448
78	65	7949	17	33	0.002134	0.336735
789	251	7666	59	88	0.007638	0.259587
80	112	7909	23	20	0.002900	0.151515
89	561	7307	86	110	0.011633	0.163934
90	48	7980	14	22	0.001751	0.314286
False	4813	2714	264	273	0.088650	0.053677
True	2723	4802	284	255	0.055840	0.085628

Table C.10: *ND2-JavaBERT-Multilabel* test set's classification report.

	Precision	**Recall**	**F1-score**	**Support**
113	0.93	0.84	0.89	467
129	0.84	0.79	0.82	768
134	0.89	0.79	0.84	300
15	0.94	0.79	0.86	91
190	0.89	0.86	0.87	1628
191	0.88	0.84	0.86	1338
197	0.86	0.85	0.86	236
319	1.00	0.99	1.00	103
36	0.91	0.79	0.84	121
369	0.91	0.85	0.88	621
400	1.00	1.00	1.00	369
470	0.88	0.71	0.78	103
476	1.00	0.98	0.99	113
606	0.83	0.66	0.73	151
643	0.89	0.73	0.80	171
690	0.93	0.90	0.91	174
78	0.79	0.66	0.72	98
789	0.81	0.74	0.77	339
80	0.83	0.85	0.84	132
89	0.87	0.84	0.85	671
90	0.77	0.69	0.73	70
False	0.95	0.95	0.95	5086
True	0.91	0.91	0.91	2978
micro avg	0.91	0.88	0.90	16128
macro avg	0.89	0.83	0.86	16128
weighted avg	0.91	0.88	0.90	16128
samples avg	0.91	0.88	0.89	16128

Table C.11: *ND2-JavaBERT-Multilabel* real-world test set's class metrics.

Class	TP	TN	FP	FN	FPR	FNR
113	2	573	30	4	0.049751	0.666667
129	0	609	0	0	0.000000	0.000000
134	0	562	47	0	0.077176	0.000000
15	0	592	17	0	0.027915	0.000000
190	0	601	8	0	0.013136	0.000000
191	0	598	11	0	0.018062	0.000000
197	0	608	1	0	0.001642	0.000000
319	0	566	43	0	0.070608	0.000000
36	0	608	1	0	0.001642	0.000000
369	0	599	10	0	0.016420	0.000000
400	4	474	71	60	0.130275	0.937500
470	0	592	17	0	0.027915	0.000000
476	0	585	24	0	0.039409	0.000000
606	0	599	10	0	0.016420	0.000000
643	0	596	13	0	0.021346	0.000000
690	0	584	25	0	0.041051	0.000000
78	2	588	1	18	0.001698	0.900000
789	0	604	5	0	0.008210	0.000000
80	0	598	11	0	0.018062	0.000000
89	26	88	6	489	0.063830	0.949515
90	0	600	5	4	0.008264	1.000000
False	218	101	156	134	0.607004	0.380682
True	102	221	131	155	0.372159	0.603113

Table C.12: *ND2-JavaBERT-Multilabel* real-world test set's classification report.

	Precision	**Recall**	**F1-score**	**Support**
113	0.06	0.33	0.11	6
129	0.00	0.00	0.00	0
134	0.00	0.00	0.00	0
15	0.00	0.00	0.00	0
190	0.00	0.00	0.00	0
191	0.00	0.00	0.00	0
197	0.00	0.00	0.00	0
319	0.00	0.00	0.00	0
36	0.00	0.00	0.00	0
369	0.00	0.00	0.00	0
400	0.05	0.06	0.06	64
470	0.00	0.00	0.00	0
476	0.00	0.00	0.00	0
606	0.00	0.00	0.00	0
643	0.00	0.00	0.00	0
690	0.00	0.00	0.00	0
78	0.67	0.10	0.17	20
789	0.00	0.00	0.00	0
80	0.00	0.00	0.00	0
89	0.81	0.05	0.10	515
90	0.00	0.00	0.00	4
False	0.58	0.62	0.60	352
True	0.44	0.40	0.42	257
micro avg	0.36	0.29	0.32	1218
macro avg	0.11	0.07	0.06	1218
weighted avg	0.62	0.29	0.31	1218
samples avg	0.38	0.29	0.32	1218

C.4 *VDET-JavaBERT-Multiclass*

Table C.13: *VDET-JavaBERT-Multiclass* test set's class metrics.

Class	TP	TN	FP	FN	FPR	FNR
Not Vuln.	7631	2896	519	481	0.151977	0.059295
15	67	11439	17	4	0.001484	0.056338
23	27	11497	0	3	0.000000	0.100000
36	22	11491	0	14	0.000000	0.388889
78	42	11471	0	14	0.000000	0.250000
80	67	11426	0	34	0.000000	0.336634
89	269	11145	107	6	0.009509	0.021818
90	14	11511	0	2	0.000000	0.125000
113	146	11306	6	69	0.000530	0.320930
129	320	11071	107	29	0.009572	0.083095
134	101	11397	21	8	0.001839	0.073394
190	428	10890	10	199	0.000917	0.317384
191	437	10928	136	26	0.012292	0.056156
197	101	11384	0	42	0.000000	0.293706
319	49	11460	18	0	0.001568	0.000000
369	236	11204	76	11	0.006738	0.044534
400	163	11293	49	22	0.004320	0.118919
470	54	11460	0	13	0.000000	0.194030
606	63	11443	13	8	0.001135	0.112676
643	45	11464	3	15	0.000262	0.250000
690	33	11479	0	15	0.000000	0.312500
789	130	11330	0	67	0.000000	0.340102

Table C.14: *VDET-JavaBERT-Multiclass* test set's classification report.

	Precision	**Recall**	**F1-score**	**Support**
Not Vuln.	0.94	0.94	0.94	8112
15	0.80	0.94	0.86	71
23	1.00	0.90	0.95	30
36	1.00	0.61	0.76	36
78	1.00	0.75	0.86	56
80	1.00	0.66	0.80	101
89	0.72	0.98	0.83	275
90	1.00	0.88	0.93	16
113	0.96	0.68	0.80	215
129	0.75	0.92	0.82	349
134	0.83	0.93	0.87	109
190	0.98	0.68	0.80	627
191	0.76	0.94	0.84	463
197	1.00	0.71	0.83	143
319	0.73	1.00	0.84	49
369	0.76	0.96	0.84	247
400	0.77	0.88	0.82	185
470	1.00	0.81	0.89	67
606	0.83	0.89	0.86	71
643	0.94	0.75	0.83	60
690	1.00	0.69	0.81	48
789	1.00	0.66	0.80	197
accuracy			0.91	11527
macro avg	0.90	0.82	0.85	11527
weighted avg	0.91	0.91	0.91	11527

Table C.15: *VDET-JavaBERT-Multiclass* real-world test set's class metrics.

Class	TP	TN	FP	FN	FPR	FNR
Not Vuln.	326	25	232	26	0.902724	0.073864
15	0	609	0	0	0.000000	0.000000
23	0	609	0	0	0.000000	0.000000
36	0	609	0	0	0.000000	0.000000
78	0	600	0	9	0.000000	1.000000
80	0	608	1	0	0.001642	0.000000
89	11	371	16	211	0.041344	0.950450
90	0	608	0	1	0.000000	1.000000
113	0	607	0	2	0.000000	1.000000
129	0	609	0	0	0.000000	0.000000
134	0	608	1	0	0.001642	0.000000
190	0	607	2	0	0.003284	0.000000
191	0	609	0	0	0.000000	0.000000
197	0	609	0	0	0.000000	0.000000
319	0	604	5	0	0.008210	0.000000
369	0	595	14	0	0.022989	0.000000
400	0	586	0	23	0.000000	1.000000
470	0	608	1	0	0.001642	0.000000
606	0	609	0	0	0.000000	0.000000
643	0	609	0	0	0.000000	0.000000
690	0	609	0	0	0.000000	0.000000
789	0	609	0	0	0.000000	0.000000

Table C.16: *VDET-JavaBERT-Multiclass* real-world test set's classification report.

	Precision	**Recall**	**F1-score**	**Support**
Not Vuln.	0.58	0.93	0.72	352
15	0.00	0.00	0.00	0
23	0.00	0.00	0.00	0
36	0.00	0.00	0.00	0
78	0.00	0.00	0.00	9
80	0.00	0.00	0.00	0
89	0.41	0.05	0.09	222
90	0.00	0.00	0.00	1
113	0.00	0.00	0.00	2
129	0.00	0.00	0.00	0
134	0.00	0.00	0.00	0
190	0.00	0.00	0.00	0
191	0.00	0.00	0.00	0
197	0.00	0.00	0.00	0
319	0.00	0.00	0.00	0
369	0.00	0.00	0.00	0
400	0.00	0.00	0.00	23
470	0.00	0.00	0.00	0
606	0.00	0.00	0.00	0
643	0.00	0.00	0.00	0
690	0.00	0.00	0.00	0
789	0.00	0.00	0.00	0
micro avg	0.55	0.55	0.55	609
macro avg	0.05	0.04	0.04	609
weighted avg	0.49	0.55	0.45	609

C.5 *ND1-JavaBERT-Multiclass*

Table C.17: *ND1-JavaBERT-Multiclass* test set's class metrics.

Class	TP	TN	FP	FN	FPR	FNR
Not Vuln.	2262	1627	128	105	0.072934	0.044360
15	17	4072	12	21	0.002938	0.552632
36	8	4090	4	20	0.000977	0.714286
78	21	4073	7	21	0.001716	0.500000
80	28	4059	9	26	0.002212	0.481481
81	17	4088	1	16	0.000245	0.484848
83	18	4088	1	15	0.000245	0.454545
89	79	3896	141	6	0.034927	0.070588
90	6	4093	2	21	0.000488	0.777778
113	86	3989	37	10	0.009190	0.104167
129	110	3969	8	35	0.002012	0.241379
134	35	4049	18	20	0.004426	0.363636
190	199	3836	13	74	0.003378	0.271062
191	192	3776	117	37	0.030054	0.161572
197	70	4031	3	18	0.000744	0.204545
319	14	4098	0	10	0.000000	0.416667
369	105	3980	15	22	0.003755	0.173228
400	74	4035	10	3	0.002472	0.038961
470	29	4076	2	15	0.000490	0.340909
476	20	4092	5	5	0.001220	0.200000
563	20	4101	0	1	0.000000	0.047619
606	22	4078	4	18	0.000980	0.450000
643	13	4086	3	20	0.000734	0.606061
690	29	4070	19	4	0.004647	0.121212
789	78	4006	11	27	0.002738	0.257143

Table C.18: *ND1-JavaBERT-Multiclass* test set's classification report.

	Precision	**Recall**	**F1-score**	**Support**
Not Vuln.	0.95	0.96	0.95	2367
15	0.59	0.45	0.51	38
36	0.67	0.29	0.40	28
78	0.75	0.50	0.60	42
80	0.76	0.52	0.62	54
81	0.94	0.52	0.67	33
83	0.95	0.55	0.69	33
89	0.36	0.93	0.52	85
90	0.75	0.22	0.34	27
113	0.70	0.90	0.79	96
129	0.93	0.76	0.84	145
134	0.66	0.64	0.65	55
190	0.94	0.73	0.82	273
191	0.62	0.84	0.71	229
197	0.96	0.80	0.87	88
319	1.00	0.58	0.74	24
369	0.88	0.83	0.85	127
400	0.88	0.96	0.92	77
470	0.94	0.66	0.77	44
476	0.80	0.80	0.80	25
563	1.00	0.95	0.98	21
606	0.85	0.55	0.67	40
643	0.81	0.39	0.53	33
690	0.60	0.88	0.72	33
789	0.88	0.74	0.80	105
accuracy			0.86	4122
macro avg	0.81	0.68	0.71	4122
weighted avg	0.88	0.86	0.86	4122

Table C.19: *ND1-JavaBERT-Multiclass* real-world test set's class metrics.

Class	**TP**	**TN**	**FP**	**FN**	**FPR**	**FNR**
Not Vuln.	338	13	244	14	0.949416	0.039773
15	0	606	3	0	0.004926	0.000000
36	0	609	0	0	0.000000	0.000000
78	0	600	0	9	0.000000	1.000000
80	0	609	0	0	0.000000	0.000000
81	0	609	0	0	0.000000	0.000000
83	0	609	0	0	0.000000	0.000000
89	1	385	2	221	0.005168	0.995495
90	0	608	0	1	0.000000	1.000000
113	0	604	3	2	0.004942	1.000000
129	0	609	0	0	0.000000	0.000000
134	0	609	0	0	0.000000	0.000000
190	0	609	0	0	0.000000	0.000000
191	0	609	0	0	0.000000	0.000000
197	0	609	0	0	0.000000	0.000000
319	0	609	0	0	0.000000	0.000000
369	0	605	4	0	0.006568	0.000000
400	0	586	0	23	0.000000	1.000000
470	0	609	0	0	0.000000	0.000000
476	0	608	1	0	0.001642	0.000000
563	0	609	0	0	0.000000	0.000000
606	0	609	0	0	0.000000	0.000000
643	0	607	2	0	0.003284	0.000000
690	0	603	6	0	0.009852	0.000000
789	0	604	5	0	0.008210	0.000000

Table C.20: *ND1-JavaBERT-Multiclass* real-world test set's classification report.

	Precision	**Recall**	**F1-score**	**Support**
Not Vuln.	0.58	0.96	0.72	352
15	0.00	0.00	0.00	0
36	0.00	0.00	0.00	0
78	0.00	0.00	0.00	9
80	0.00	0.00	0.00	0
81	0.00	0.00	0.00	0
83	0.00	0.00	0.00	0
89	0.33	0.00	0.01	222
90	0.00	0.00	0.00	1
113	0.00	0.00	0.00	2
129	0.00	0.00	0.00	0
134	0.00	0.00	0.00	0
190	0.00	0.00	0.00	0
191	0.00	0.00	0.00	0
197	0.00	0.00	0.00	0
319	0.00	0.00	0.00	0
369	0.00	0.00	0.00	0
400	0.00	0.00	0.00	23
470	0.00	0.00	0.00	0
476	0.00	0.00	0.00	0
563	0.00	0.00	0.00	0
606	0.00	0.00	0.00	0
643	0.00	0.00	0.00	0
690	0.00	0.00	0.00	0
789	0.00	0.00	0.00	0
micro avg	0.56	0.56	0.56	609
macro avg	0.04	0.04	0.03	609
weighted avg	0.46	0.56	0.42	609

C.6 *ND2-JavaBERT-Multiclass*

Table C.21: *ND2-JavaBERT-Multiclass* test set's class metrics.

Class	TP	TN	FP	FN	FPR	FNR
Not Vuln.	4774	2695	283	312	0.095030	0.061345
15	31	8008	6	19	0.000749	0.380000
36	41	7976	26	21	0.003249	0.338710
78	37	8002	7	18	0.000874	0.327273
80	61	7971	12	20	0.001503	0.246914
89	153	7762	81	68	0.010328	0.307692
90	25	8018	7	14	0.000872	0.358974
113	123	7868	35	38	0.004429	0.236025
129	209	7695	88	72	0.011307	0.256228
134	63	7958	5	38	0.000628	0.376238
190	437	7368	162	97	0.021514	0.181648
191	336	7473	151	104	0.019806	0.236364
197	117	7913	17	17	0.002144	0.126866
319	36	8019	9	0	0.001121	0.000000
369	147	7850	8	59	0.001018	0.286408
400	126	7918	20	0	0.002520	0.000000
470	35	7997	8	24	0.000999	0.406780
476	29	8027	3	5	0.000374	0.147059
606	34	7997	14	19	0.001748	0.358491
643	41	7943	63	17	0.007869	0.293103
690	41	7997	11	15	0.001374	0.267857
789	131	7852	21	60	0.002667	0.314136

Table C.22: *ND2-JavaBERT-Multiclass* test set's classification report.

	Precision	**Recall**	**F1-score**	**Support**
Not Vuln.	0.94	0.94	0.94	5086
15	0.84	0.62	0.71	50
36	0.61	0.66	0.64	62
78	0.84	0.67	0.75	55
80	0.84	0.75	0.79	81
89	0.65	0.69	0.67	221
90	0.78	0.64	0.70	39
113	0.78	0.76	0.77	161
129	0.70	0.74	0.72	281
134	0.93	0.62	0.75	101
190	0.73	0.82	0.77	534
191	0.69	0.76	0.72	440
197	0.87	0.87	0.87	134
319	0.80	1.00	0.89	36
369	0.95	0.71	0.81	206
400	0.86	1.00	0.93	126
470	0.81	0.59	0.69	59
476	0.91	0.85	0.88	34
606	0.71	0.64	0.67	53
643	0.39	0.71	0.51	58
690	0.79	0.73	0.76	56
789	0.86	0.69	0.76	191
accuracy			0.87	8064
macro avg	0.79	0.75	0.76	8064
weighted avg	0.88	0.87	0.87	8064

Table C.23: *ND2-JavaBERT-Multiclass* real-world test set's class metrics.

Class	**TP**	**TN**	**FP**	**FN**	**FPR**	**FNR**
Not Vuln.	330	30	227	22	0.883268	0.0625
15	0	609	0	0	0.000000	0.000000
36	0	609	0	0	0.000000	0.000000
78	0	600	0	9	0.000000	1.0000
80	0	609	0	0	0.000000	0.000000
89	0	385	2	222	0.005168	1.0000
90	0	608	0	1	0.000000	1.0000
113	0	607	0	2	0.000000	1.0000
129	0	606	3	0	0.004926	0.000000
134	0	607	2	0	0.003284	0.000000
190	0	600	9	0	0.014778	0.000000
191	0	599	10	0	0.016420	0.000000
197	0	609	0	0	0.000000	0.000000
319	0	603	6	0	0.009852	0.000000
369	0	602	7	0	0.011494	0.000000
400	0	582	4	23	0.006826	1.0000
470	0	606	3	0	0.004926	0.000000
476	0	608	1	0	0.001642	0.000000
606	0	609	0	0	0.000000	0.000000
643	0	606	3	0	0.004926	0.000000
690	0	607	2	0	0.003284	0.000000
789	0	609	0	0	0.000000	0.000000

Table C.24: *ND2-JavaBERT-Multiclass* real-world test set's classification report.

	Precision	**Recall**	**F1-score**	**Support**
Not Vuln.	0.59	0.94	0.73	352
15	0.00	0.00	0.00	0
36	0.00	0.00	0.00	0
78	0.00	0.00	0.00	9
80	0.00	0.00	0.00	0
89	0.00	0.00	0.00	222
90	0.00	0.00	0.00	1
113	0.00	0.00	0.00	2
129	0.00	0.00	0.00	0
134	0.00	0.00	0.00	0
190	0.00	0.00	0.00	0
191	0.00	0.00	0.00	0
197	0.00	0.00	0.00	0
319	0.00	0.00	0.00	0
369	0.00	0.00	0.00	0
400	0.00	0.00	0.00	23
470	0.00	0.00	0.00	0
476	0.00	0.00	0.00	0
606	0.00	0.00	0.00	0
643	0.00	0.00	0.00	0
690	0.00	0.00	0.00	0
789	0.00	0.00	0.00	0
micro avg	0.54	0.54	0.54	609
macro avg	0.03	0.04	0.03	609
weighted avg	0.34	0.54	0.42	609

C.7 *VDET-BFP_combined-Multiclass*

Table C.25: *VDET-BFP$_{combined}$-Multiclass* test set's class metrics.

Class	TP	TN	FP	FN	FPR	FNR
Not Vuln.	7277	3251	164	835	0.048023	0.102934
15	67	11439	17	4	0.001484	0.056338
23	27	11497	0	3	0.000000	0.100000
36	32	11483	8	4	0.000696	0.111111
78	55	11459	12	1	0.001046	0.017857
80	93	11403	23	8	0.002013	0.079208
89	251	11156	96	24	0.008532	0.087273
90	14	11511	0	2	0.000000	0.125000
113	209	11263	49	6	0.004332	0.027907
129	326	11067	111	23	0.009930	0.065903
134	101	11397	21	8	0.001839	0.073394
190	564	10758	142	63	0.013028	0.100478
191	435	10930	134	28	0.012111	0.060475
197	136	11334	50	7	0.004392	0.048951
319	49	11460	18	0	0.001568	0.000000
369	236	11204	76	11	0.006738	0.044534
400	185	11272	70	0	0.006172	0.000000
470	66	11446	14	1	0.001222	0.014925
606	53	11450	6	18	0.000524	0.253521
643	53	11451	16	7	0.001395	0.116667
690	35	11476	3	13	0.000261	0.270833
789	183	11280	50	14	0.004413	0.071066

Table C.26: *VDET-BFP$_{combined}$-Multiclass* test set's classification report.

	Precision	**Recall**	**F1-score**	**Support**
Not Vuln.	0.98	0.90	0.94	8112
15	0.80	0.94	0.86	71
23	1.00	0.90	0.95	30
36	0.80	0.89	0.84	36
78	0.82	0.98	0.89	56
80	0.80	0.92	0.86	101
89	0.72	0.91	0.81	275
90	1.00	0.88	0.93	16
113	0.81	0.97	0.88	215
129	0.75	0.93	0.83	349
134	0.83	0.93	0.87	109
190	0.80	0.90	0.85	627
191	0.76	0.94	0.84	463
197	0.73	0.95	0.83	143
319	0.73	1.00	0.84	49
369	0.76	0.96	0.84	247
400	0.73	1.00	0.84	185
470	0.82	0.99	0.90	67
606	0.90	0.75	0.82	71
643	0.77	0.88	0.82	60
690	0.92	0.73	0.81	48
789	0.79	0.93	0.85	197
accuracy			0.91	11527
macro avg	0.82	0.92	0.86	11527
weighted avg	0.92	0.91	0.91	11527

Table C.27: *VDET-BFP$_{combined}$-Multiclass* real-world test set's class metrics.

Class	**TP**	**TN**	**FP**	**FN**	**FPR**	**FNR**
Not Vuln.	275	64	193	77	0.750973	0.218750
15	0	606	3	0	0.004926	0.000000
23	0	609	0	0	0.000000	0.000000
36	0	608	1	0	0.001642	0.000000
78	2	598	2	7	0.003333	0.777778
80	0	607	2	0	0.003284	0.000000
89	4	385	2	218	0.005168	0.981982
90	0	608	0	1	0.000000	1.000000
113	0	607	0	2	0.000000	1.000000
129	0	609	0	0	0.000000	0.000000
134	0	583	26	0	0.042693	0.000000
190	0	608	1	0	0.001642	0.000000
191	0	606	3	0	0.004926	0.000000
197	0	608	1	0	0.001642	0.000000
319	0	590	19	0	0.031199	0.000000
369	0	598	11	0	0.018062	0.000000
400	0	572	14	23	0.023891	1.000000
470	0	600	9	0	0.014778	0.000000
606	0	609	0	0	0.000000	0.000000
643	0	609	0	0	0.000000	0.000000
690	0	609	0	0	0.000000	0.000000
789	0	568	41	0	0.067323	0.000000

Table C.28: *VDET-BFP$_{combined}$-Multiclass* real-world test set's classification report.

	Precision	**Recall**	**F1-score**	**Support**
Not Vuln.	0.59	0.78	0.67	352
15	0.00	0.00	0.00	0
23	0.00	0.00	0.00	0
36	0.00	0.00	0.00	0
78	0.50	0.22	0.31	9
80	0.00	0.00	0.00	0
89	0.67	0.02	0.04	222
90	0.00	0.00	0.00	1
113	0.00	0.00	0.00	2
129	0.00	0.00	0.00	0
134	0.00	0.00	0.00	0
190	0.00	0.00	0.00	0
191	0.00	0.00	0.00	0
197	0.00	0.00	0.00	0
319	0.00	0.00	0.00	0
369	0.00	0.00	0.00	0
400	0.00	0.00	0.00	23
470	0.00	0.00	0.00	0
606	0.00	0.00	0.00	0
643	0.00	0.00	0.00	0
690	0.00	0.00	0.00	0
789	0.00	0.00	0.00	0
micro avg	0.46	0.46	0.46	609
macro avg	0.08	0.05	0.05	609
weighted avg	0.59	0.46	0.41	609

C.8 *VDET-BFP_single-Multiclass*

Table C.29: *VDET-BFP$_{single}$-Multiclass* test set's class metrics.

Class	TP	TN	FP	FN	FPR	FNR
Not Vuln.	7311	3210	205	801	0.060029	0.098743
15	67	11439	17	4	0.001484	0.056338
23	27	11497	0	3	0.000000	0.100000
36	34	11481	10	2	0.000870	0.055556
78	55	11459	12	1	0.001046	0.017857
80	93	11403	23	8	0.002013	0.079208
89	269	11145	107	6	0.009509	0.021818
90	14	11511	0	2	0.000000	0.125000
113	153	11302	10	62	0.000884	0.288372
129	292	11103	75	57	0.006710	0.163324
134	101	11397	21	8	0.001839	0.073394
190	585	10734	166	42	0.015229	0.066986
191	437	10929	135	26	0.012202	0.056156
197	136	11334	50	7	0.004392	0.048951
319	46	11461	17	3	0.001481	0.061224
369	236	11204	76	11	0.006738	0.044534
400	185	11272	70	0	0.006172	0.000000
470	66	11446	14	1	0.001222	0.014925
606	63	11443	13	8	0.001135	0.112676
643	53	11451	16	7	0.001395	0.116667
690	33	11478	1	15	0.000087	0.312500
789	183	11280	50	14	0.004413	0.071066

Table C.30: *VDET-BFP$_{single}$-Multiclass* test set's classification report.

	Precision	**Recall**	**F1-score**	**Support**
Not Vuln.	0.97	0.90	0.94	8112
15	0.80	0.94	0.86	71
23	1.00	0.90	0.95	30
36	0.77	0.94	0.85	36
78	0.82	0.98	0.89	56
80	0.80	0.92	0.86	101
89	0.72	0.98	0.83	275
90	1.00	0.88	0.93	16
113	0.94	0.71	0.81	215
129	0.80	0.84	0.82	349
134	0.83	0.93	0.87	109
190	0.78	0.93	0.85	627
191	0.76	0.94	0.84	463
197	0.73	0.95	0.83	143
319	0.73	0.94	0.82	49
369	0.76	0.96	0.84	247
400	0.73	1.00	0.84	185
470	0.82	0.99	0.90	67
606	0.83	0.89	0.86	71
643	0.77	0.88	0.82	60
690	0.97	0.69	0.80	48
789	0.79	0.93	0.85	197
accuracy			0.91	11527
macro avg	0.82	0.91	0.86	11527
weighted avg	0.92	0.91	0.91	11527

Table C.31: *VDET-BFP$_{single}$-Multiclass* real-world test set's class metrics.

Class	**TP**	**TN**	**FP**	**FN**	**FPR**	**FNR**
Not Vuln.	292	62	195	60	0.758755	0.170455
15	0	588	21	0	0.034483	0.000000
23	0	609	0	0	0.000000	0.000000
36	0	609	0	0	0.000000	0.000000
78	2	599	1	7	0.001667	0.777778
80	0	607	2	0	0.003284	0.000000
89	8	379	8	214	0.020672	0.963964
90	0	608	0	1	0.000000	1.000000
113	0	602	5	2	0.008237	1.000000
129	0	607	2	0	0.003284	0.000000
134	0	571	38	0	0.062397	0.000000
190	0	607	2	0	0.003284	0.000000
191	0	608	1	0	0.001642	0.000000
197	0	609	0	0	0.000000	0.000000
319	0	603	6	0	0.009852	0.000000
369	0	607	2	0	0.003284	0.000000
400	0	578	8	23	0.013652	1.000000
470	0	608	1	0	0.001642	0.000000
606	0	609	0	0	0.000000	0.000000
643	0	609	0	0	0.000000	0.000000
690	0	609	0	0	0.000000	0.000000
789	0	594	15	0	0.024631	0.000000

Table C.32: *VDET-BFP$_{single}$-Multiclass* real-world test set's classification report.

	Precision	**Recall**	**F1-score**	**Support**
Not Vuln.	0.60	0.83	0.70	352
15	0.00	0.00	0.00	0
23	0.00	0.00	0.00	0
36	0.00	0.00	0.00	0
78	0.67	0.22	0.33	9
80	0.00	0.00	0.00	0
89	0.50	0.04	0.07	222
90	0.00	0.00	0.00	1
113	0.00	0.00	0.00	2
129	0.00	0.00	0.00	0
134	0.00	0.00	0.00	0
190	0.00	0.00	0.00	0
191	0.00	0.00	0.00	0
197	0.00	0.00	0.00	0
319	0.00	0.00	0.00	0
369	0.00	0.00	0.00	0
400	0.00	0.00	0.00	23
470	0.00	0.00	0.00	0
606	0.00	0.00	0.00	0
643	0.00	0.00	0.00	0
690	0.00	0.00	0.00	0
789	0.00	0.00	0.00	0
micro avg	0.50	0.50	0.50	609
macro avg	0.08	0.05	0.05	609
weighted avg	0.54	0.50	0.43	609

C.9 *ND1-BFP_combined-Multiclass*

Table C.33: *ND1-BFP$_{combined}$-Multiclass* test set's class metrics.

Class	TP	TN	FP	FN	FPR	FNR
Not Vuln.	2201	1673	82	166	0.046724	0.070131
15	17	4078	6	21	0.001469	0.552632
36	8	4091	3	20	0.000733	0.714286
78	21	4069	11	21	0.002696	0.500000
80	22	4054	14	32	0.003441	0.592593
81	17	4084	5	16	0.001223	0.484848
83	18	4081	8	15	0.001956	0.454545
89	71	3900	137	14	0.033936	0.164706
90	5	4093	2	22	0.000488	0.814815
113	86	3976	50	10	0.012419	0.104167
129	131	3871	106	14	0.026653	0.096552
134	34	4055	12	21	0.002951	0.381818
190	217	3808	41	56	0.010652	0.205128
191	188	3850	43	41	0.011045	0.179039
197	70	4026	8	18	0.001983	0.204545
319	21	4089	9	3	0.002196	0.125000
369	105	3981	14	22	0.003504	0.173228
400	77	4035	10	0	0.002472	0.000000
470	27	4076	2	17	0.000490	0.386364
476	22	4090	7	3	0.001709	0.120000
563	20	4101	0	1	0.000000	0.047619
606	21	4078	4	19	0.000980	0.475000
643	11	4089	0	22	0.000000	0.666667
690	28	4076	13	5	0.003179	0.151515
789	84	4004	13	21	0.003236	0.200000

Table C.34: *ND1-BFP$_{combined}$-Multiclass* test set's classification report.

	Precision	**Recall**	**F1-score**	**Support**
Not Vuln	0.96	0.93	0.95	2367
15	0.74	0.45	0.56	38
36	0.73	0.29	0.41	28
78	0.66	0.50	0.57	42
80	0.61	0.41	0.49	54
81	0.77	0.52	0.62	33
83	0.69	0.55	0.61	33
89	0.34	0.84	0.48	85
90	0.71	0.19	0.29	27
113	0.63	0.90	0.74	96
129	0.55	0.90	0.69	145
134	0.74	0.62	0.67	55
190	0.84	0.79	0.82	273
191	0.81	0.82	0.82	229
197	0.90	0.80	0.84	88
319	0.70	0.88	0.78	24
369	0.88	0.83	0.85	127
400	0.89	1.00	0.94	77
470	0.93	0.61	0.74	44
476	0.76	0.88	0.81	25
563	1.00	0.95	0.98	21
606	0.84	0.53	0.65	40
643	1.00	0.33	0.50	33
690	0.68	0.85	0.76	33
789	0.87	0.80	0.83	105
accuracy			0.85	4122
macro avg	0.77	0.69	0.70	4122
weighted avg	0.88	0.85	0.86	4122

Table C.35: *ND1-BFP$_{combined}$-Multiclass* real-world test set's class metrics.

Class	TP	TN	FP	FN	FPR	FNR
Not Vuln.	270	42	215	82	0.836576	0.232955
15	0	603	6	0	0.009852	0.000000
36	0	609	0	0	0.000000	0.000000
78	1	564	36	8	0.060000	0.888889
80	0	607	2	0	0.003284	0.000000
81	0	609	0	0	0.000000	0.000000
83	0	609	0	0	0.000000	0.000000
89	2	382	5	220	0.012920	0.990991
90	0	607	1	1	0.001645	1.000000
113	0	605	2	2	0.003295	1.000000
129	0	608	1	0	0.001642	0.000000
134	0	608	1	0	0.001642	0.000000
190	0	607	2	0	0.003284	0.000000
191	0	607	2	0	0.003284	0.000000
197	0	602	7	0	0.011494	0.000000
319	0	609	0	0	0.000000	0.000000
369	0	603	6	0	0.009852	0.000000
400	0	586	0	23	0.000000	1.000000
470	0	595	14	0	0.022989	0.000000
476	0	608	1	0	0.001642	0.000000
563	0	607	2	0	0.003284	0.000000
606	0	609	0	0	0.000000	0.000000
643	0	595	14	0	0.022989	0.000000
690	0	602	7	0	0.011494	0.000000
789	0	597	12	0	0.019704	0.000000

Table C.36: *ND1-BFP$_{combined}$-Multiclass* real-world test set's classification report.

	Precision	**Recall**	**F1-score**	**Support**
Not Vuln.	0.56	0.77	0.65	352
15	0.00	0.00	0.00	0
36	0.00	0.00	0.00	0
78	0.03	0.11	0.04	9
80	0.00	0.00	0.00	0
81	0.00	0.00	0.00	0
83	0.00	0.00	0.00	0
89	0.29	0.01	0.02	222
90	0.00	0.00	0.00	1
113	0.00	0.00	0.00	2
129	0.00	0.00	0.00	0
134	0.00	0.00	0.00	0
190	0.00	0.00	0.00	0
191	0.00	0.00	0.00	0
197	0.00	0.00	0.00	0
319	0.00	0.00	0.00	0
369	0.00	0.00	0.00	0
400	0.00	0.00	0.00	23
470	0.00	0.00	0.00	0
476	0.00	0.00	0.00	0
563	0.00	0.00	0.00	0
606	0.00	0.00	0.00	0
643	0.00	0.00	0.00	0
690	0.00	0.00	0.00	0
789	0.00	0.00	0.00	0
micro avg	0.45	0.45	0.45	609
macro avg	0.03	0.04	0.03	609
weighted avg	0.43	0.45	0.38	609

C.10 *ND1-BFP_single-Multiclass*

Table C.37: *ND1-BFP$_{single}$-Multiclass* test set's class metrics.

Class	TP	TN	FP	FN	FPR	FNR
Not Vuln.	2229	1655	100	138	0.056980	0.058302
15	18	4074	10	20	0.002449	0.526316
36	9	4091	3	19	0.000733	0.678571
78	23	4072	8	19	0.001961	0.452381
80	35	4034	34	19	0.008358	0.351852
81	19	4085	4	14	0.000978	0.424242
83	18	4080	9	15	0.002201	0.454545
89	54	4017	20	31	0.004954	0.364706
90	5	4095	0	22	0.000000	0.814815
113	86	3980	46	10	0.011426	0.104167
129	126	3908	69	19	0.017350	0.131034
134	52	3971	96	3	0.023605	0.054545
190	230	3800	49	43	0.012731	0.157509
191	169	3876	17	60	0.004367	0.262009
197	72	4031	3	16	0.000744	0.181818
319	20	4090	8	4	0.001952	0.166667
369	105	3981	14	22	0.003504	0.173228
400	68	4041	4	9	0.000989	0.116883
470	27	4077	1	17	0.000245	0.386364
476	21	4091	6	4	0.001464	0.160000
563	21	4101	0	0	0.000000	0.000000
606	21	4078	4	19	0.000980	0.475000
643	13	4086	3	20	0.000734	0.606061
690	28	4079	10	5	0.002446	0.151515
789	83	3965	52	22	0.012945	0.209524

Table C.38: *ND1-BFP$_{single}$-Multiclass* test set's classification report.

	Precision	**Recall**	**F1-score**	**Support**
Not Vuln.	0.96	0.94	0.95	2367
15	0.64	0.47	0.55	38
36	0.75	0.32	0.45	28
78	0.74	0.55	0.63	42
80	0.51	0.65	0.57	54
81	0.83	0.58	0.68	33
83	0.67	0.55	0.60	33
89	0.73	0.64	0.68	85
90	1.00	0.19	0.31	27
113	0.65	0.90	0.75	96
129	0.65	0.87	0.74	145
134	0.35	0.95	0.51	55
190	0.82	0.84	0.83	273
191	0.91	0.74	0.81	229
197	0.96	0.82	0.88	88
319	0.71	0.83	0.77	24
369	0.88	0.83	0.85	127
400	0.94	0.88	0.91	77
470	0.96	0.61	0.75	44
476	0.78	0.84	0.81	25
563	1.00	1.00	1.00	21
606	0.84	0.53	0.65	40
643	0.81	0.39	0.53	33
690	0.74	0.85	0.79	33
789	0.61	0.79	0.69	105
accuracy			0.86	4122
macro avg	0.78	0.70	0.71	4122
weighted avg	0.88	0.86	0.86	4122

Table C.39: *ND1-BFP$_{single}$-Multiclass* real-world test set's class metrics.

Class	TP	TN	FP	FN	FPR	FNR
Not Vuln.	253	74	183	99	0.712062	0.281250
15	0	560	49	0	0.080460	0.000000
36	0	606	3	0	0.004926	0.000000
78	3	592	8	6	0.013333	0.666667
80	0	602	7	0	0.011494	0.000000
81	0	609	0	0	0.000000	0.000000
83	0	608	1	0	0.001642	0.000000
89	1	377	10	221	0.025840	0.995495
90	1	606	2	0	0.003289	0.000000
113	0	599	8	2	0.013180	1.000000
129	0	609	0	0	0.000000	0.000000
134	0	608	1	0	0.001642	0.000000
190	0	608	1	0	0.001642	0.000000
191	0	602	7	0	0.011494	0.000000
197	0	603	6	0	0.009852	0.000000
319	0	608	1	0	0.001642	0.000000
369	0	607	2	0	0.003284	0.000000
400	0	586	0	23	0.000000	1.000000
470	0	607	2	0	0.003284	0.000000
476	0	609	0	0	0.000000	0.000000
563	0	609	0	0	0.000000	0.000000
606	0	609	0	0	0.000000	0.000000
643	0	605	4	0	0.006568	0.000000
690	0	609	0	0	0.000000	0.000000
789	0	553	56	0	0.091954	0.000000

Table C.40: *ND1-BFP$_{single}$-Multiclass* real-world test set's classification report.

	Precision	**Recall**	**F1-score**	**Support**
Not Vuln.	0.58	0.72	0.64	352
15	0.00	0.00	0.00	0
36	0.00	0.00	0.00	0
78	0.27	0.33	0.30	9
80	0.00	0.00	0.00	0
81	0.00	0.00	0.00	0
83	0.00	0.00	0.00	0
89	0.09	0.00	0.01	222
90	0.33	1.00	0.50	1
113	0.00	0.00	0.00	2
129	0.00	0.00	0.00	0
134	0.00	0.00	0.00	0
190	0.00	0.00	0.00	0
191	0.00	0.00	0.00	0
197	0.00	0.00	0.00	0
319	0.00	0.00	0.00	0
369	0.00	0.00	0.00	0
400	0.00	0.00	0.00	23
470	0.00	0.00	0.00	0
476	0.00	0.00	0.00	0
563	0.00	0.00	0.00	0
606	0.00	0.00	0.00	0
643	0.00	0.00	0.00	0
690	0.00	0.00	0.00	0
789	0.00	0.00	0.00	0
micro avg	0.42	0.42	0.42	609
macro avg	0.05	0.08	0.06	609
weighted avg	0.37	0.42	0.38	609

C.11 *ND2-BFP_combined-Multiclass*

Table C.41: *ND2-$BFP_{combined}$-Multiclass* test set's class metrics.

Class	TP	TN	FP	FN	FPR	FNR
Not Vuln.	4843	2683	295	243	0.099060	0.047778
15	38	8004	10	12	0.001248	0.240000
36	53	7981	21	9	0.002624	0.145161
78	35	8003	6	20	0.000749	0.363636
80	71	7940	43	10	0.005386	0.123457
89	185	7764	79	36	0.010073	0.162896
90	22	8014	11	17	0.001371	0.435897
113	127	7881	22	34	0.002784	0.211180
129	201	7726	57	80	0.007324	0.284698
134	80	7929	34	21	0.004270	0.207921
190	409	7447	83	125	0.011023	0.234082
191	352	7519	105	88	0.013772	0.200000
197	116	7910	20	18	0.002522	0.134328
319	35	8019	9	1	0.001121	0.027778
369	152	7844	14	54	0.001782	0.262136
400	123	7921	17	3	0.002142	0.023810
470	41	7999	6	18	0.000750	0.305085
476	33	8024	6	1	0.000747	0.029412
606	36	7997	14	17	0.001748	0.320755
643	38	8004	2	20	0.000250	0.344828
690	46	7990	18	10	0.002248	0.178571
789	139	7856	17	52	0.002159	0.272251

Table C.42: *ND2-BFP$_{combined}$-Multiclass* test set's classification report.

	Precision	**Recall**	**F1-score**	**Support**
Not Vuln.	0.94	0.95	0.95	5086
15	0.79	0.76	0.78	50
36	0.72	0.85	0.78	62
78	0.85	0.64	0.73	55
80	0.62	0.88	0.73	81
89	0.70	0.84	0.76	221
90	0.67	0.56	0.61	39
113	0.85	0.79	0.82	161
129	0.78	0.72	0.75	281
134	0.70	0.79	0.74	101
190	0.83	0.77	0.80	534
191	0.77	0.80	0.78	440
197	0.85	0.87	0.86	134
319	0.80	0.97	0.88	36
369	0.92	0.74	0.82	206
400	0.88	0.98	0.92	126
470	0.87	0.69	0.77	59
476	0.85	0.97	0.90	34
606	0.72	0.68	0.70	53
643	0.95	0.66	0.78	58
690	0.72	0.82	0.77	56
789	0.89	0.73	0.80	191
accuracy			0.89	8064
macro avg	0.80	0.79	0.79	8064
weighted avg	0.89	0.89	0.89	8064

Table C.43: *ND2-BFP$_{combined}$-Multiclass* real-world test set's class metrics.

Class	**TP**	**TN**	**FP**	**FN**	**FPR**	**FNR**
Not Vuln.	296	47	210	56	0.817121	0.159091
15	0	607	2	0	0.003284	0.000000
36	0	606	3	0	0.004926	0.000000
78	1	597	3	8	0.005000	0.888889
80	0	602	7	0	0.011494	0.000000
89	2	382	5	220	0.012920	0.990991
90	0	608	0	1	0.000000	1.000000
113	0	604	3	2	0.004942	1.000000
129	0	606	3	0	0.004926	0.000000
134	0	565	44	0	0.072250	0.000000
190	0	606	3	0	0.004926	0.000000
191	0	607	2	0	0.003284	0.000000
197	0	609	0	0	0.000000	0.000000
319	0	608	1	0	0.001642	0.000000
369	0	603	6	0	0.009852	0.000000
400	0	574	12	23	0.020478	1.000000
470	0	603	6	0	0.009852	0.000000
476	0	609	0	0	0.000000	0.000000
606	0	609	0	0	0.000000	0.000000
643	0	609	0	0	0.000000	0.000000
690	0	609	0	0	0.000000	0.000000
789	0	609	0	0	0.000000	0.000000

Table C.44: *ND2-BFP$_{combined}$-Multiclass* real-world test set's classification report.

	Precision	**Recall**	**F1-score**	**Support**
Not Vuln.	0.58	0.84	0.69	352
15	0.00	0.00	0.00	0
36	0.00	0.00	0.00	0
78	0.25	0.11	0.15	9
80	0.00	0.00	0.00	0
89	0.29	0.01	0.02	222
90	0.00	0.00	0.00	1
113	0.00	0.00	0.00	2
129	0.00	0.00	0.00	0
134	0.00	0.00	0.00	0
190	0.00	0.00	0.00	0
191	0.00	0.00	0.00	0
197	0.00	0.00	0.00	0
319	0.00	0.00	0.00	0
369	0.00	0.00	0.00	0
400	0.00	0.00	0.00	23
470	0.00	0.00	0.00	0
476	0.00	0.00	0.00	0
606	0.00	0.00	0.00	0
643	0.00	0.00	0.00	0
690	0.00	0.00	0.00	0
789	0.00	0.00	0.00	0
micro avg	0.49	0.49	0.49	609
macro avg	0.05	0.04	0.04	609
weighted avg	0.45	0.49	0.41	609

C.12 *ND2-BFP_single-Multiclass*

Table C.45: *ND2-BFP$_{single}$-Multiclass* test set's class metrics.

Class	TP	TN	FP	FN	FPR	FNR
Not Vuln.	4768	2719	259	318	0.086971	0.062525
15	36	8004	10	14	0.001248	0.280000
36	44	7986	16	18	0.002000	0.290323
78	32	8007	2	23	0.000250	0.418182
80	67	7969	14	14	0.001754	0.172840
89	182	7705	138	39	0.017595	0.176471
90	23	8018	7	16	0.000872	0.410256
113	123	7888	15	38	0.001898	0.236025
129	200	7664	119	81	0.015290	0.288256
134	68	7956	7	33	0.000879	0.326733
190	414	7380	150	120	0.019920	0.224719
191	346	7483	141	94	0.018494	0.213636
197	121	7909	21	13	0.002648	0.097015
319	35	8020	8	1	0.000997	0.027778
369	162	7833	25	44	0.003181	0.213592
400	119	7926	12	7	0.001512	0.055556
470	40	7994	11	19	0.001374	0.322034
476	28	8024	6	6	0.000747	0.176471
606	33	7991	20	20	0.002497	0.377358
643	39	7991	15	19	0.001874	0.327586
690	41	8000	8	15	0.000999	0.267857
789	126	7860	13	65	0.001651	0.340314

Table C.46: *ND2-BFP_{single}-Multiclass* test set's classification report.

	Precision	**Recall**	**F1-score**	**Support**
Not Vuln.	0.95	0.94	0.94	5086
15	0.78	0.72	0.75	50
36	0.73	0.71	0.72	62
78	0.94	0.58	0.72	55
80	0.83	0.83	0.83	81
89	0.57	0.82	0.67	221
90	0.77	0.59	0.67	39
113	0.89	0.76	0.82	161
129	0.63	0.71	0.67	281
134	0.91	0.67	0.77	101
190	0.73	0.78	0.75	534
191	0.71	0.79	0.75	440
197	0.85	0.90	0.88	134
319	0.81	0.97	0.89	36
369	0.87	0.79	0.82	206
400	0.91	0.94	0.93	126
470	0.78	0.68	0.73	59
476	0.82	0.82	0.82	34
606	0.62	0.62	0.62	53
643	0.72	0.67	0.70	58
690	0.84	0.73	0.78	56
789	0.91	0.66	0.76	191
accuracy			0.87	8064
macro avg	0.80	0.76	0.77	8064
weighted avg	0.88	0.87	0.88	8064

Table C.47: *ND2-BFP$_{single}$-Multiclass* real-world test set's class metrics.

Class	**TP**	**TN**	**FP**	**FN**	**FPR**	**FNR**
Not Vuln.	295	43	214	57	0.832685	0.161932
15	0	608	1	0	0.001642	0.000000
36	0	605	4	0	0.006568	0.000000
78	1	599	1	8	0.001667	0.888889
80	0	601	8	0	0.013136	0.000000
89	7	378	9	215	0.023256	0.968468
90	0	608	0	1	0.000000	1.000000
113	0	603	4	2	0.006590	1.000000
129	0	607	2	0	0.003284	0.000000
134	0	599	10	0	0.016420	0.000000
190	0	605	4	0	0.006568	0.000000
191	0	609	0	0	0.000000	0.000000
197	0	606	3	0	0.004926	0.000000
319	0	608	1	0	0.001642	0.000000
369	0	600	9	0	0.014778	0.000000
400	0	582	4	23	0.006826	1.000000
470	0	588	21	0	0.034483	0.000000
476	0	609	0	0	0.000000	0.000000
606	0	609	0	0	0.000000	0.000000
643	0	609	0	0	0.000000	0.000000
690	0	602	7	0	0.011494	0.000000
789	0	605	4	0	0.006568	0.000000

Table C.48: *ND2-BFP_{single}-Multiclass* real-world test set's classification report.

	Precision	**Recall**	**F1-score**	**Support**
Not Vuln	0.58	0.84	0.69	352
15	0.00	0.00	0.00	0
36	0.00	0.00	0.00	0
78	0.50	0.11	0.18	9
80	0.00	0.00	0.00	0
89	0.44	0.03	0.06	222
90	0.00	0.00	0.00	1
113	0.00	0.00	0.00	2
129	0.00	0.00	0.00	0
134	0.00	0.00	0.00	0
190	0.00	0.00	0.00	0
191	0.00	0.00	0.00	0
197	0.00	0.00	0.00	0
319	0.00	0.00	0.00	0
369	0.00	0.00	0.00	0
400	0.00	0.00	0.00	23
470	0.00	0.00	0.00	0
476	0.00	0.00	0.00	0
606	0.00	0.00	0.00	0
643	0.00	0.00	0.00	0
690	0.00	0.00	0.00	0
789	0.00	0.00	0.00	0
micro avg	0.50	0.50	0.50	609
macro avg	0.07	0.04	0.04	609
weighted avg	0.50	0.50	0.42	609