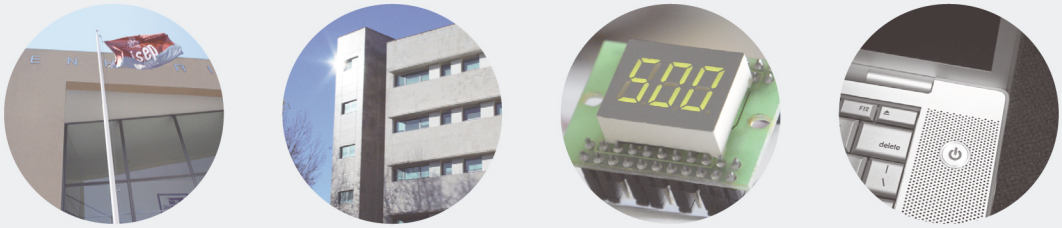
INSTITUTO SUPERIOR DE ENGENHARIA DO PORTO
MESTRADO EM ENGENHARIA ELECTROTÉCNICA E DE COMPUTADORES



Automated Vulnerability Detection in Source Code

ARTHUR QUINTINO BATISTA
Setembro de 2023

POLITÉCNICO
DO PORTO

Politécnico do Porto
Instituto Superior de Engenharia do Porto

Automated Vulnerability Detection in Source Code

Arthur Quintino Batista

Master in Electrical and Computer Engineering
Specialization Area of Autonomous Systems

Departamento de Engenharia Eletrotécnica
Instituto Superior de Engenharia do Porto

September, 2023

This dissertation partially satisfies the requirements of the Thesis/Dissertation course of the program Master in Electrical and Computer Engineering, Specialization Area of Autonomous Systems.

Candidate: Arthur Quintino Batista, No. 1171006, 1171006@isep.ipp.pt

Scientific Guidance: Professora Doutora Isabel Praça, icp@isep.ipp.pt

Scientific Co-Guidance: Professora Doutora Eva Maia, egm@isep.ipp.pt

Departamento de Engenharia Eletrotécnica
Instituto Superior de Engenharia do Porto
Rua Dr. António Bernardino de Almeida, 431, 4200–072 Porto

September, 2023

Acknowledgements

Em primeiro lugar, gostaria de expressar a minha profunda gratidão à Professora Isabel Praça e à Professora Eva Maia pela oportunidade, confiança e, acima de tudo, pela paciência demonstrada ao longo deste ano. Sem dúvida, ambas desempenharam um papel crucial na conclusão deste trabalho, e este marco não teria sido possível sem o apoio constante e a disponibilidade que sempre me ofereceram. Quero estender o meu agradecimento a todos os membros do GECAD com quem tive a oportunidade de interagir, pelo valioso auxílio e enriquecedores debates de ideias.

Ao Instituto Superior de Engenharia do Porto e aos seus docentes, desejo expressar a minha sincera gratidão pelo ensino de excelência que recebi, preparando-me de forma sólida para o meu futuro.

Gostaria também de agradecer a todas as pessoas que hoje posso chamar de amigos, com quem partilhei diversas experiências ao longo do meu percurso académico e que, de diversas maneiras, contribuíram para este momento e para o meu sucesso. Um agradecimento especial à minha namorada, pelo apoio constante, pela paciência e por estar presente nos momentos mais cruciais da minha vida académica, sem dúvidas, foi um pilar fundamental e uma das razões para ter chegado onde estou hoje.

Acima de tudo, quero dedicar um agradecimento especial à minha mãe, que ao longo destes 24 anos dedicou-se, esforçou-se e fez sacrifícios em prol da minha felicidade e bem-estar. Alcançar este ponto na minha vida não teria sido possível sem o seu apoio e orientação. Devo-lhe tudo o que conquistei até hoje e tudo o que ainda espero conquistar no futuro. Esta conquista não é apenas minha, mas nossa, por isso agradeço do fundo do coração e espero poder retribuir de alguma forma todo o amor e dedicação que sempre me proporcionou.

Abstract

Technological advances have facilitated instant global connectivity, transforming the way we interact with the world. Software, propelled by this evolution, plays a pivotal role in our daily lives, being present in virtually every facet of our existence. Programmers, who form the bedrock of the business structure, create source code comprising hundreds or even thousands of lines, encompassing essential functionalities for software to operate seamlessly. However, owing to the inherent complexity of these functionalities and their interdependencies, it is common for errors to escape notice in the code, inadvertently reaching the software production phase and resulting in code vulnerabilities

Each year, the number of identified software vulnerabilities, either publicly disclosed or discovered internally, increases. These vulnerabilities pose a significant risk of exploitation, potentially leading to data breaches or service interruptions. Therefore, the goal of this project is to develop a tool capable of analyzing code written in C and C++ to detect vulnerabilities before the code is deployed to end users.

To achieve this goal, we leveraged existing work in this area by using a dataset of open-source functions written in C and C++. This dataset contains approximately 1.27 million functions categorized into five different Common Weakness Enumerations (CWEs). Preprocessing was performed to optimize the performance of the models used. The models were trained on function snippets only, without considering any external context of the code, thus simplifying the problem and increasing processing efficiency. The results obtained are promising, with the trained models showing high performance in identifying and classifying vulnerabilities. In addition, these results can serve as a benchmark for direct comparisons between different approaches.

Keywords: Vulnerability Detection, Machine Learning, Binary Classification, Multi-Label Classification, Neural Network, Source-Code, Data Processing, Computer Security, CWE.

Resumo

O avanço tecnológico permitiu uma conexão global instantânea, transformando a maneira como interagimos com o mundo. Os softwares, impulsionados por essa evolução, desempenham um papel crucial em nosso cotidiano, estando presentes em praticamente todos os aspectos de nossas vidas. Os programadores, fundamentais na estrutura empresarial, desenvolvem o código-fonte composto por centenas ou até milhares de linhas, incorporando as funcionalidades essenciais para o pleno funcionamento dos softwares. No entanto, devido à complexidade intrínseca dessas funcionalidades e suas interdependências, é comum que erros passem despercebidos no código, chegando inadvertidamente à fase de produção do software e resultando em vulnerabilidades de código.

Anualmente, observa-se um aumento no número de vulnerabilidades de software que são identificadas e divulgadas publicamente ou descobertas internamente. Essas vulnerabilidades representam um sério risco e podem resultar em fuga de informações ou interrupção de serviços. Assim, este projeto visa desenvolver uma ferramenta capaz de analisar o código escrito em C e C++ para identificar vulnerabilidades antes que esse código chegue ao consumidor final.

Para alcançar esse objetivo, utilizamos como ponto de partida diversos trabalhos já realizados nessa área, fazendo uso de um conjunto de dados contendo funções de código aberto escritas em C e C++. Esse conjunto de dados engloba cerca de 1.27 milhões de funções categorizadas por cinco diferentes Common Weakness Enumerations (CWEs). Realizamos um pré-processamento para otimizar o desempenho dos modelos utilizados. Os modelos foram treinados apenas em trechos de funções, sem considerar qualquer contexto externo sobre o código, simplificando assim o problema e melhorando a eficiência do processamento. Os resultados obtidos são promissores, pois os modelos treinados foram capazes de identificar e classificar as vulnerabilidades com alto desempenho, estes resultados podem também servir como base para comparação direta entre diferentes abordagens.

Palavras-Chave: Deteção de Vulnerabilidades, *Machine Learning*, Classificação Binária, Classificação Multi-Etiqueta, Rede Neural, Código-Fonte, Processamento de Dados, Segurança Informática, CWE.

Contents

List of Figures

List of Tables

Listings

List of Acronyms

ADAM	Adaptive Moment Estimation
AI	Artificial Intelligence
ANN	Artificial Neural Network
AST	Abstract Syntax Tree
AUC	Area Under the Curve
BLSTM	Bidirectional Long Short Term Memory
BoW	Bag-of-Words
CNN	Convolutional Neural Network
CVE	Common Vulnerabilities and Exposure
CWE	Common Weakness Enumeration
DL	Deep Learning
FN	False Negative
FNN	Feedfoward Neural Network
FP	False Positive
GBT	Gradient Boosted Tree
GloVe	Global Vectors for Word Representation
GRU	Gated Recurrent Unit
LSTM	Long Short-Term Memory
MCC	Matthew's Correlation Coefficient
ML	Machine Learning
MLP	Multilayer Perceptron
MSE	Mean Square Error

NLP	Natural Language Process
NN	Neural Network
PLOVER	Preliminary List Of Vulnerability Examples for Researchers
RBF	Radial Basis Funcion
RBFN	Radial Basis Funcion Network
RF	Random Forest
RNN	Recurrent Neural Network
ROC	Receiver Operating Characteristic
SGD	Stochastic Gradient Descent
SVM	Support Vector Machine
TN	True Negative
TP	True Positive

Chapter 1

Introduction

A programmer is the most valuable asset of a software company since these professionals create, develop, and maintain all the products that customers will have access to. During software creation, developers write hundreds or thousands of lines of code that form the core functions of the software. Given the extensive number of functions and their dependencies, it is normal for the code to contain errors that go unnoticed by the programmer, and many of these errors end up being shipped to production.

A bug in software can be defined as an underlying cause of a failure or an unwanted output. These bugs can be visible, as they are detected directly by the programmers or compilers, or hidden, as they can only be detected after code reviews, testing, or, in many cases, by the end user. The presence of bugs is not only associated with small-scale programs, larger companies like Google, Microsoft and Amazon give rewards to programmers who find bugs in their software. In Google's case, the Vulnerability Reward Program was created, which in 2022 offered $12 million to security researchers who found about 3000 security vulnerabilities in their software [1]. This monetary prize may seem high, however, more serious bugs can render a software useless. In worst case, these erros can create an exploitable bug, which is also known as a vulnerability. In such situations, these vulnerabilities could be exploited to run malicious software that could, for example, grant attackers unauthorized privileges, such as remote access. Exploits represent a serious security risk and should be addressed as soon as possible. Repairing these vulnerabilities is often more cost-effective than repairing the systems after an attack.

The concern for software security is not a recent concept. Common Weakness Enumeration (CWE) is a list created by the developer community to categorize various common problems in software or hardware that could cause security vulnerabilities. The project began in 1999 when MITRE launched the Common Vulnerabilities and Exposure (CVE) list, with the goal of classifying vulnerabilities, attacks, and flaws. In 2005, the MITRE research team revised these classifications for commercial use, resulting in a document called Preliminary List Of Vulnerability Examples for Researchers (PLOVER). These efforts then culminated in the creation of the Common Weakness Enumeration in 2006, which is aimed at developers, security professionals, and other stakeholders [2]. The goal is to educate software and hardware architects, designers and developers on how to eliminate the most common bugs before products are delivered. The CWE list and associated classification taxonomy serve as a common language that can be used to identify and describe these weaknesses in terms of CWE [3]. This project is updated to the present day and is sponsored by the National Cybersecurity FFRDC, which is operated by MITRE with support from US-CERT and the US National Cyber Security Division [4].

In 2019, *Mend* [5] company analyzed a database containing information about vulnerabilities reported on GitHub, the National Vulnerability Database, security advisories, and open-source project trackers over a decade of programming languages, including C and C++. The study concluded that the most common CWEs in both C and C++ are CWE-119 (Buffer Errors) and CWE-20 (Improper Input Validation) [6]. Figure 1.1 presents a graph showing the most common CWEs and their percentages in the studied database of code written in C++.

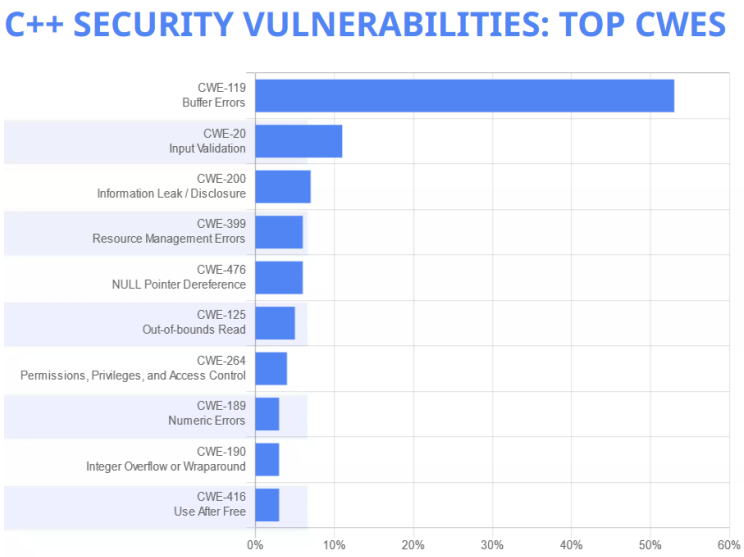


Figure 1.1: Most Common CWEs in C++ [7].

Another study, this time by The Stack, found a record 26448 software vulnerabilities reported in 2022, a 59% increase from the previous year, as shown in the Figure 1.2. This study covered all programming languages that have one or more entries on the CVE list [8].

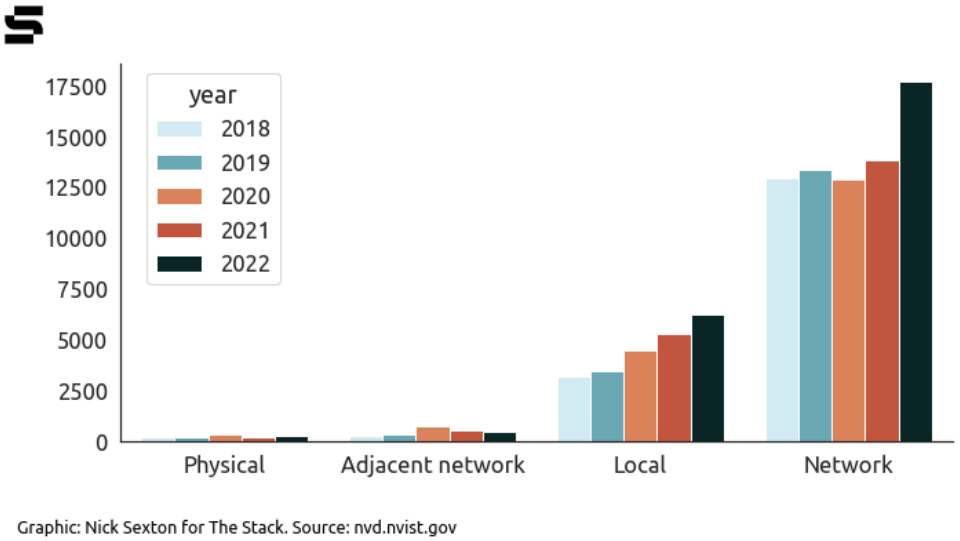


Figure 1.2: How vulnerabilities have been exploited over the years [8].

1.1 Background and Problem Definition

Each year, an increasing number of vulnerabilities are discovered within systems. This rise necessitates companies to invest significant resources, both in terms of money and time, to identify and address these vulnerabilities. Additionally, hidden flaws in software can result in security vulnerabilities that may allow attackers to compromise the system. Despite increased investment in cybersecurity over the years, the number of vulnerabilities in code continues to grow due to the following factors [9]:

- **New Attack Vectors and Increasing Sophistication** - The rise of well-funded cybercriminal organizations has led to an increase in vulnerabilities, making sophisticated cyber controls and regulations obsolete. With a focus on Ransomware attacks, the use of ubiquitous open-source components in codebases and expanding supply chain and third-party risks, organizations need to prioritize their security measures to prevent potential vulnerabilities from being exploited.

- **Adoption of New Technologies** - The increase in vulnerabilities is closely linked to the adoption of new technologies. While the adoption of new technologies brings many benefits to organizations, it also exposes them to unfamiliar risks. For example, cloud technologies offer vast opportunities, but they also present a diverse attack surface that the security industry doesn't fully understand. While technology drives innovation in organizations, the expansion of IT networks to support business growth exposes potential vulnerabilities. At the same time, the number of applications being used by organizations is increasing significantly.

- **Infrastructure and Development Practices** - Smaller code snippets, legacy systems, and system integrations can all create vulnerabilities that threaten an organization's security. Additionally, digital transformation and lack of collaboration between development and security teams can further increase the risk. It is important for organizations to stay aware of these risks and implement effective security measures to protect their systems and data.

- **Organizational Change** - Organizations experience significant changes in their business operations, which may potentially introduce novel vulnerabilities, such as mergers, acquisitions, divestments, or legacy system eradication. Historically, there has never been a high level of concern for systems security. However, this mindset has been changing, and budgets may need to be realigned to prioritize this area and effectively mitigate risks. Access changes like work-from-home, mobile platforms, and remote work have further increased the likelihood of a breach. These changes create new attack surfaces such as cloud migrations and shadow IT, and new systems and applications.

- **Talent Shortage** - The shortage of cybersecurity and software development talent is a major concern for organizations. The growing number of unfilled positions in cybersecurity and software development has led to a lack of expertise and knowledge in these fields. This shortage amplifies existing factors that contribute to vulnerabilities such as new technologies mentioned above.. In addition, outsourcing development versus in-house development can also influence the number of vulnerabilities found in software. With a talent shortage, comes burnout, insufficient training, employee turnover, and keeping pace with workloads, all of which can contribute to recurring vulnerabilities, unpatched issues, and a higher likelihood for errors.

- **Evolving Testing Approaches and Technologies** - It is important to embrace technology as a force multiplier and recognize the limitations of pentesting

due to its time constraints. Shifting left in the software development life cycle (SDLC) is recommended to identify vulnerabilities earlier, although the number of vulnerabilities detected can vary based on testing depth and code quality. The advancement in testing technology is also revealing additional ways to manually uncover critical vulnerabilities.

- **AI-Assisted Code Tools** - AI-powered coding tools are limited by their inability to think creatively outside the box and consider different points of view. These tools rely heavily on understanding problem contexts, which hinders their ability to think innovatively. This limitation stems from their strict adherence to predefined rules and procedures. If these tools are trained on third-party code without proper validation, they may produce code with inherent vulnerabilities and risks. Legal liability for AI-enhanced coding is complex and evolving. As Artificial Intelligence (AI) algorithms become more integral to software development, there's uncertainty about how to assign blame for bugs, security vulnerabilities, and other issues that arise from the generated code. Responsibility may be shared among various parties, including developers, AI tool providers, and software companies. Factors such as the level of involvement, reliance on the AI-generated code, and specific circumstances would influence the assessment of liability [10].

Due to all these factors, it is essential to develop new solutions or improve existing ones to minimize and even eliminate the presence of software bugs. Detecting and fixing bugs during the development phase will reduce the time and money spent in the future. It is in this context we aims to create ways to automatically detect vulnerabilities in code through the steps presented in section 1.1.1.

1.1.1 Project Goal

The goal of the project is to implement AI method capable of automatically detecting vulnerabilities in C/C++ source code. Given the complexity and variety of programs, a public dataset with a large number of training examples will be used to train Machine Learning (ML) models that can effectively learn the patterns of security vulnerabilities directly from the code. To achieve the goal, the following steps should be performed:

1. Analysis of the State of the Art:
2. Analysis of a large public dataset of C/C++ source code, both vulnerable and non-vulnerable;
3. Preprocessing the dataset to convert the source code into a format that can be input into a machine learning model;

4. Training machine learning models, such as a neural network, on the preprocessed dataset;

5. Using the trained model to predict whether new pieces of C/C++ source code contain vulnerabilities;

6. Evaluating the performance of the model using metrics.

1.1.2 Expected Results

The objective of this thesis is to investigate the use of machine learning for the precise and effective classification and identification of vulnerabilities in C/C++ functions. The expected outcome is an improvement of existing approaches to detect and classify vulnerabilities. In this way, unlike tradicional static bug detection tools, the ML model should be able to identify vulnerabilities in the semantics of text while ignoring all references and contexts behind the function source code. This approach will be simpler and lighter and can be used by the vast majority of current machines.

Figure 1.3 provides an overview of the expected framework for this work, highlighting the main steps and techniques needed to classify source code written in C and C++ using ML techniques.

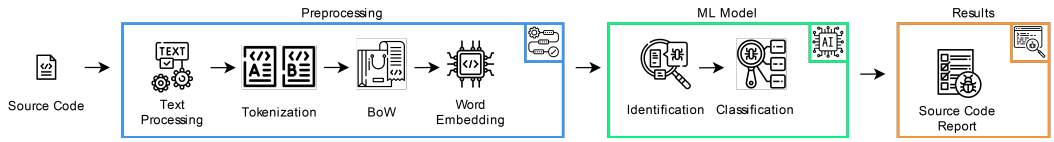


Figure 1.3: Source Code Validation Framework.

1.2 Project Schedule

Table 1.1 shows the various activities planned for the work that will be carried out in this thesis and the respective duration of each in weeks.

Table 1.1: Project Timeline.

S/N	Activity	No. Weeks
1.	Requirement Analysis and Specification	1
2.	Literature Review	5
3.	Data Collection and Analysis	4
4.	Machine Learning Model Design and Implementation	15
5.	Parameters Tuning and Performance Evaluation	10
6.	Validation of Results	4
7.	Conclusions	1

1.3 Report Structure

The present thesis is divided into five chapters, which are described in the following points:

1. **Introduction** - This chapter is dedicated to introducing the thesis, contextualizing and describing the problem it aims to solve, providing an overview of its motivation and outlining the expected results.

2. **State of Art** - This chapter presents the concepts that support the dissertation and the existing literature on the topic of automatic detection and classification of vulnerabilities in C/C++ source code.

3. **Data Analysis & Preprocessing** - This chapter describes the data contained in the dataset and all the preprocessing techniques used.

4. **Machine Learning Applied to Source-Code Verification** - A description is provided of the Machine Learning models that were studied, along with a brief presentation of the results.

5. **Results and Discussion** - The results obtained from the models used will be presented, as well as their discussion and comparison with the literature reviewed.

6. **Conclusion and Future Work** - This chapter will discuss the conclusions and improvements about the results obtained from the developed solution.

Chapter 2

State of the Art

This chapter will present the literature review conducted in the scope of this project. Its purpose is not only to ensure the understanding of the subject matter but also the integrity of this project. This chapter introduces the types of code analysis that currently exist, provides an introduction to C/C++ and its vulnerabilities, and covers some topics related to AI and ML. In this way, other works and approaches related to this thesis will also be presented, and a comparative analysis will be conducted to establish a knowledge base for working on the project.

2.1 Code Analysis

As mentioned in Chapter 1, timely detection of software vulnerabilities is crucial for ensuring the security of the software. With the evolution of Software Engineering, it is now possible to use tools that make this process semi-automatic or fully automatic. These tools are practical applications of the theoretical ideas proposed in the *Halting Problem* presented by Alan Turing [11]. This problem asks whether a program can determine if another program will be interrupted or executed forever. There is no algorithmic solution that can guarantee a correct answer for all possible inputs, making the *Halting Problem* undecidable [12]. While they cannot learn in the exact sense that Turing proposed, they do use algorithms and heuristics to identify common errors and alert developers to potential problems before the program runs. However, like all software tools, they are not perfect and may

not catch all errors. Nonetheless, through continuous development, these tools strive to approach the goal of solving the *Halting Problem* and enabling machines to learn the patterns embedded in source code.

It was in 70s that one of the first code analysis tools, called *Lint*, appeared in response to the popularity of the C language and the ease of commenting on bugs that could later be compiled. The Lint tool addressed these issues by performing a code analysis and flagging potential errors in a way that closely resembles today's compiler warnings [13]. Recently, the evolution of compilers has led to an increased awareness of the importance of code analysis tools. These tools have taken on a pivotal role in uncovering errors that might escape detection during the compiler's verification process, including potential vulnerabilities. These tools employ various methodologies that include:

- **Static Analysis** - A static analysis tool is a process that examines all the syntax of source code for errors and security vulnerabilities. The vulnerability detection in static code analysis is performed while the software is still in the development stage. It can be done manually or through automatic source code scanning tools [14].

- **Dynamic Analysis** - Dynamic Code Analysis is performed after the developer runs the source code by executing the program with random inputs. This analysis utilizes techniques such as fuzzing, which generates randomized inputs to the software system in order to trigger a system crash and expose any vulnerabilities. [15].

- **Symbolic Execution** - Symbolic execution is an analysis technique that tests software for potential violations of specific properties. These properties encompass various scenarios, such as division by zero, null pointer dereferencing, and bypassing authentication backdoors. This technique facilitates the simultaneous exploration of multiple paths with different inputs by employing a symbolic execution engine. The engine utilizes symbolic input values and preserves a Boolean formula for each explored path, describing the conditions satisfied by the taken branches. Additionally, it maintains a symbolic memory store that maps variables to symbolic expressions or values [16].

- **Software Metrics** - A software metric is a mathematical definition that maps the entities of a software system in an attempt to measure or predict some attribute, whether internal or external, of a software or process, in order to obtain objective, reproducible, and quantifiable measurements [17]. Furthermore, we understand a software metrics tool as a program that implements a

set of software metrics definitions that can be directly applied to code analysis tools [18]. These software metrics will then assist both developers and users in evaluating the performance and results of the code analysis tool.

- **Machine Learning** - The use of ML for the detection of vulnerabilities has been gaining more and more space in the scientific community. This technique has the availability of a large amount of open source code, which opens opportunities for the algorithm to learn the patterns of vulnerabilities from mined data [19]. With the continued advancement of ML models, it is now possible to apply this technique to a variety of code analysis approaches. This is due to the ability of these models to assimilate a variety of code analysis methods. In certain situations, they can identify more complex relationships, resulting in superior performance compared to conventional approaches that do not use a ML model.

Currently, the software development industry has numerous commercial and non-commercial tools for performing code analysis such as *SonarQube, Checkmarx SAST, Coverity* and *FindBugs*. However, this dissertation will focus on the use of ML-based techniques to statically analyze vulnerabilities in the code, also using software metrics to evaluate the performance of the developed tool.

2.2 Language C/C++

C programming language was developed by Dennis Ritchie in 1973 and is a direct evolution of the B and BCPL languages [20]. In its early years, C gained popularity as a programming language for academic environments and supported by commercial professionals. With the release of several compilers and the increasing popularity of Unix systems, the language gained a reputation and became widely used in industrial and commercial environments [21]. In Listing 2.1 shows a simple code example written in C.

```
1 #include<stdio.h>
2
3 void main(void)
4 {
5   printf("Hello World");
6 }
```

Listing 2.1: Simple Code for "Hello World" in C.

The popularity of the C programming language was due to its advantages over existing programming languages at the time. It is a procedure-oriented language, which makes it easier to understand the approach. In addition, it has a feature-rich library, fast compile and run times, and a compact size, taking up only 3-5 MB in its configurations. C is also portable, as it can generate an executable (.exe) file when compiled, and has a steep learning curve [21].

In 1983, the C++ programming language was developed. It is an extension of the C language created by B. Stroustrup with the aim of modernizing and adding new features. The vast majority of C syntax is still valid in C++, as they share the same basic data types like integers, characters, and floating-point numbers, as well as similar control structures including loops and conditional statements. However, C++ adds object-oriented programming features such as function overloading and templates to its syntax. It was originally called "C with classes" due to its support for the object-oriented programming paradigm, but it aimed to improve upon C by supporting data abstraction, generic programming, and providing a more enjoyable programming experience for serious programmers [22].

2.2.1 Bad Memory Allocation

During software development, a distinction is commonly made between normal program actions and erroneous actions. However, in many cases, it is only possible to determine if a program has an error after it has been executed. In a secure programming language, errors are trapped as they occur. For instance, the JAVA programming language is considered highly secure due to its exception system that show a representation of the problem that occurred and contains information about the type of error and where it occurred. On the other hand, in languages like C and C++, errors are not typically trapped, allowing the software to continue running even after an incorrect operation, which can adversely impact its overall behavior and potentially lead to severe consequences [23].

The vast majority of C/C++ code that involves memory allocations completely ignores security issues. As a result, one of the most common types of vulnerabilities associated with improper memory usage is the buffer overflow, which accounts 32% of C vulnerabilities and 53% of C++ vulnerabilities in 2019 [7]. In Listing 2.2, an example of a vulnerable code snippet in the Linux Kernel caused by buffer overflow is present, classified as CWE-120. This classification will be discussed in the next section. The vulnerability occurs due to the use of the C function "strcpy", which copies a string from one memory location to another without checking the boundaries of the destination buffer. This vulnerability can be easily fixed by replacing the "strcpy" function with the "strlcpy" function [24]. The "strlcpy" function ensures that the destination string is always properly terminated

with the null character ("\0"), thereby preventing a buffer overflow. If the source string is longer than the destination, the function will create a truncated copy to ensure it fits within the destination.

```
1 n = num_mixer_volumes++;
2
3 strcpy(mixer_vols[n].name, name);
4
5 if (present)
6     mixer_vols[n].num = n;
7 else
8     mixer_vols[n].num = -1;
```

Listing 2.2: Code Snippet of Buffer Overflow in C/C++ Classified as CWE-120 [24].

The most common type of attack that occurs when this vulnerability is present is a "Heap-based Buffer Overflow". In this type of attack, attackers exploit the dynamically allocated heap memory used by the application at run-time to perform an indirect pointer attack. They manipulate the memory pointers by accessing the memory management information stored in or around the managed memory blocks. Another frequently encountered attack is known as "Off-By-One Errors," which is a specific case of buffer overflows. It transpires when adjacent memory locations are overwritten by exactly one byte. Additionally, attacks utilizing only pointers include the "dangling pointer reference." This refers to a pointer that points to a memory location that is no longer allocated, typically leading to program crashes. This type of error often coincides with the "memory heap" issue, where memory is freed twice [25].

There are a number of vulnerabilities in software systems, each of which represents a significant threat that requires careful analysis. However, in this panorama of vulnerabilities, there is one that stands out as the most common, and that is memory misallocation. This particular vulnerability, as described above, plays a key role in creating vulnerabilities in the code.

The impact of these vulnerabilities cannot be underestimated, as they have the potential to cause significant damage. Therefore, it is imperative to take a proactive approach to preventing them. The consequences of such memory manipulation are far-reaching, allowing attackers to gain administrative privileges. This unrestricted access paves the way for a complete takeover of the system, placing it under the control of these unauthorized entities.

2.3 Artificial Intelligence

Artificial Intelligence is a field of computer science that aims to develop intelligent machines capable of performing tasks that typically require human thought, such as detecting errors in source code. The concept of AI was widely discussed in the first half of the 20th century, and in 1950, Alan Turing published a paper called "Computing Machinery and Intelligence", in which he explored the mathematical possibility of AI. In that paper, Turing proposed that machines could use available information to solve problems and make decisions, just like humans do [26].

In 1956, the Dartmouth Summer Conference on Artificial Intelligence Research (DSRPAI) organized by John McCarthy and Marvin Minks showed the first AI program, called Logic Theorist, was presented by Allen Newell, Cliff Shaw, and Herbert Simon as a proof of concept for AI. This program was designed to mimic the problem-solving abilities of a human being and was funded by the Research and Development Corporation (RAND) [27]. Although the conference fell short of expectations because there was no agreement among the scientific community on standard methods for the field but it was remarkable because it demonstrated the possibility of developing AI, driving its research and development in the following years.

The evolution of technology made computers increasingly powerful, enabling them to solve complex mathematical problems quickly, and become more affordable and accessible. This has facilitated the development and improvement of algorithms, including ML algorithms, resulting in the creation of numerous AI models serving different purposes. A noteworthy example is the supercomputer "Deep Blue" (Figure 2.1), which defeated the world chess champion in a match in 1997.

Figure 2.1: One of the two cabinets of Deep Blue [28].

Currently, the advancement of computing has enabled AI to give rise to new subfields, including algorithms based on ML and Deep Learning, as illustrated in Figure 2.2. ML enables machines to learn from data and make decisions based on patterns found in datasets. Deep Learning, as the name suggests, involves deeper layers of computational models, meaning that there are more than one hidden layer to reach more precise conclusions.

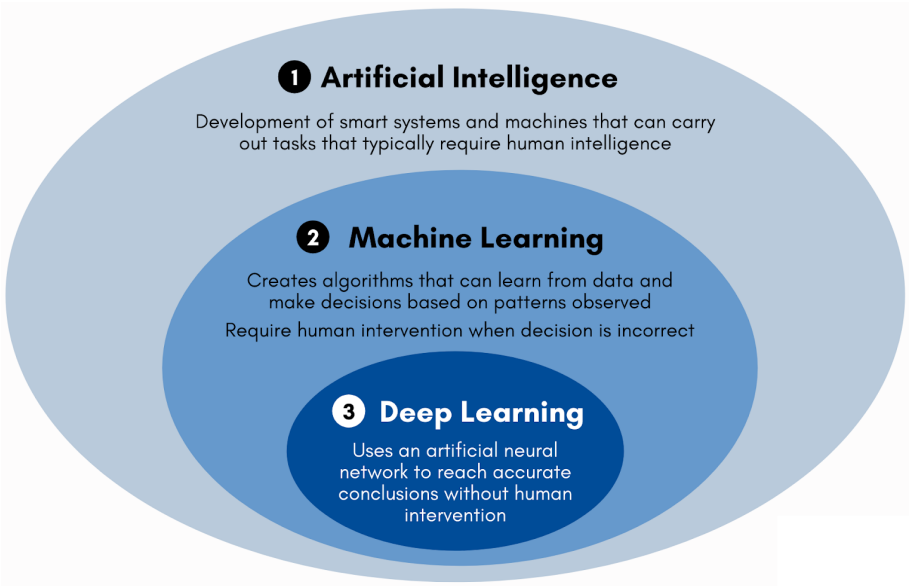


Figure 2.2: Overview Artificial Intelligence, Machine Learning and Deep Learning [29].

2.3.1 Machine Learning

Machine Learning (ML) is a technique used in AI that involves running algorithms, enables a computer program to enhance its performance through experience in specific tasks and performance measures. Its primary objective is to automate the creation of analytical models for cognitive tasks like object recognition and natural language translation. This is achieved by employing algorithms that iteratively learn from training data, allowing the machine to comprehend complex patterns and their relationships without explicit programming [30]. Consequently, ML algorithms have achieved great success in the following areas:

1. **Computer Vision** - This field finds applications in various tasks such as object detection, object recognition, and object processing [31].

2. **Prediction** - ML can be applied to several subdomains within prediction, including classification, analysis and recommendation. Text classification is an example of classification, image analysis falls under analysis, and medical diagnosis can be an example of recommendation [31].

3. **Semantic Analysis and Natural Language Processing** - Semantic analysis establishes relationships between syntax structures in paragraphs, sentences, and words, while natural language processing ensures accurate processing of natural language data [31].

To make these applications from these areas possible, there are several techniques used in ML algorithms, depending on the purpose of AI. These techniques are summarized in the following points:

- **Supervised Learning** - The algorithm creates a function that maps the inputs to generate the desired outputs. This technique uses labeled data to generate the model that will be able to predict the label of a piece of raw data. For this reason, this technique is widely used in applications that require the classification of a given problem [32].
- **Unsupervised Learning** - In this technique, the learning system detects patterns without the presence of a predetermined label or specification. The training data consists only of variables where the goal of the technique is to find structural information of interest, such as a group of elements that share the same property (known as clustering) or data representations [30].
- **Reinforcement Learning** - In this technique, instead of providing input and output pairs, the current states of the system are presented, and a goal is then set, listing the permissible actions and system constraints. This approach enables the ML model to learn how to achieve the goal through trial and error, by rewarding success and providing feedback on its actions [30].

2.3.2 Deep Learning

Deep Learning (DL) is a subfield of AI that is employed when dealing with high-dimensional and large-scale data, such as applications that require processing text, images, videos, and audio [30]. Typically, Deep Learning (DL)-based algorithms utilize deep architectures, characterized by multiple layers of abstraction known as "hidden layers." These layers enable computational models to learn data representations at varying levels of abstraction [33]. Consequently, DL-based algorithms are capable of uncovering intricate relationships within extensive datasets, surpassing the capabilities of shallow ML algorithms.

Therefore, DL finds applications in various fields including science, business, and public administration. With the advancements in computing power, DL algorithms have achieved significant breakthroughs by setting records in speech recognition, image recognition, predicting the activity of potential drug molecules, analyzing particle accelerator data, building brain circuits and predicting the effects of mutations in non-coding DNA. DL has also shown great

potential in natural language understanding. As we look ahead, the utilization of DL-based algorithms is expected to grow, leading to even more promising results. This is due to the fact that DL applications require less manual engineering and can leverage the vast amount of computation power and data available [33].

2.3.3 Artificial Neural Networks

Artificial Neural Network (ANN) or NN are computational models used in AI for ML. Their name are inspired by the biological neural networks that make up the human brain and are based on their structure and functioning. This class of algorithms attempts to find patterns and make predictions about a set of input data.

Artificial Neural Network are composed of a number of connected nodes divided into layers, as shown in Figure 2.3. There is an input layer, one or more hidden layers, and an output layer. Each node, or artificial neuron, as can be seen in Figure 2.3, is connected to others and has an associated weight and threshold. If a node's output is above the estipulated value, the node is activated and sends data to the next layer of the network. Otherwise, no data will be transmitted [34].

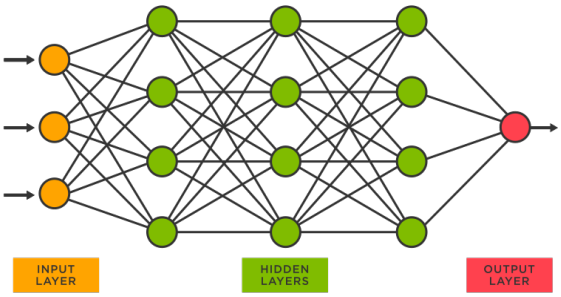


Figure 2.3: Artificial Neural Network and their Layers [35].

The accuracy of NN will depend on several factors associated with their training. A well-trained NN will produce output data with high accuracy. Figure 2.4 shows the composition of an artificial neuron.

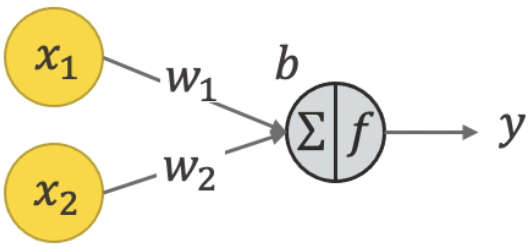


Figure 2.4: Structure of the Artificial Neuron [36].

This neuron takes inputs (x_1 and x_2), wich are multiplied by weights (w_1 and w_2) and produces an output (y) (Equation 2.1). The assignment of weights determines the importance given to each variable, and those with higher weights/values contribute more significantly to the output than those with lower weights/values Then, all the weighted inputs are summed together with a bias (b) (Equation 2.2).

$$\begin{aligned} x_1 &\rightarrow x_1 * w_1 \\ x_2 &\rightarrow x_2 * w_2 \end{aligned} \tag{2.1}$$

$$(x_1 * w_1) + (x_2 * w_2) + b \tag{2.2}$$

Finally, the sum is transformed using an activation function that converts the unbounded input into an output that has a smooth and predictable shape, as described by Equation 2.3 and simplified to Equation 2.4.

$$y = f(x_1 * w_1 + x_2 * w_2 + b) \tag{2.3}$$

$$f(x) = y = \sum_{j=1}^{n} x_j * w_{ij} + b \tag{2.4}$$

This activation function adds non-linearity to the Neural Network and determines the output that will be transmitted to the nodes in the next layer. If the output is equal to or greater than the threshold value, the activation function will have a value of "1" and will be transmitted to the next node and layer. If the output is less than the threshold value, the activation function will have a value of "0" and nothing will be transmitted. There are several types of activation functions, the most popular ones are the binary step function, *sigmoid*, and *ReLU*.

The binary step function, shown in Equation 2.5, depends on the threshold value to determine whether or not the neuron will be activated.

$$f(x) = \begin{cases} 0 & \text{if } x < 0 \\ 1 & \text{if } x \geq 0 \end{cases} \tag{2.5}$$

The *sigmoid* function, shown in Equation 2.6, maps any input value to an output value within the range of [0,1].

$$f(x) = \frac{1}{1 + e^{-x}} \tag{2.6}$$

The *ReLU* function, shown in Equation 2.7, which stands for Rectified Linear Unit, sets all negative input values to zero while leaving positive input values unchanged. This way, neurons will only be deactivated if the output of this function is less than or equal to 0.

$$f(x) = max(0, x) \tag{2.7}$$

If we break down the operation of a single node using binary values, we can apply this concept to a more tangible example that translates into a simple decision making problem (Yes = 1, No = 0) [34]. For instance, let's consider a situation where a person needs to decide whether to enter a university, and there are three factors that influence their decision:

1. Is the university close to their residence? (Yes = 1, No = 0)
2. Is the university public? (Yes = 1, No = 0)
3. Does the university teach engineering? (Yes = 1, No = 0)

Then, let's assume the following inputs:
- $X_1 = 0$ (The university is not close to their residence)
- $X_2 = 1$ (The university is public)
- $X_3 = 1$ (The university teaches engineering)

Now we assign weights to each input based on their relative importance:
- $W_1 = 1$ (Since the person will move to the university dorms)
- $W_2 = 3$ (Since it is uncertain if the person will receive a scholarship)
- $W_3 = 5$ (Since the person is interested in studying engineering)

Finally, a threshold value of 3 will be assumed, which translates into a bias value of -3. The output equation that will solve this decision-making problem will take the following form:

$$\begin{aligned} &y = (X_1 \times W_1) + (X_2 \times W_2) + (X_3 \times W_3) + b \\ &\equiv y = (0 \times 1) + (1 \times 3) + (1 \times 5) - 3 = 5 \end{aligned} \tag{2.8}$$

The equation defined in 2.8 closely resembles the output equation of a neuron, which is defined by equation 2.3. Thus, if any of the activation functions defined above are used, we can determine that the output of the node would be 1, since the output (y) is greater than 0. Therefore, the solution to this decision-making problem is that the person chooses to attend this university.

This idea that the nodes of a NN transmit data to each other is just an easier way to think about it, in fact this is just a theoretical concept. These operations are mathematically translated into a set of vectors and matrices that are multiplied together in order to obtain the desired result for which the network was trained.

To evaluate the accuracy of a NN, a loss function is used. The loss function is a mathematical function that measures the difference between the predicted output of the NN and the actual output, and provides a scalar value that represents the degree of error. A high value of the loss function indicates that the predictions are

far from the expected output, while a low value indicates that the predictions are closer to the expected output. An example of a loss function is Mean Square Error (MSE), where its variables are described in Equation 2.9 [37].

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(y_{true} - y_{pred})^2 \tag{2.9}$$

- n - Number of Samples;
- y - Represent the variable to be predicted;
- y_{true} - Is the true value of the variable, that is the correct answer;
- y_{pred} - Is the predicted value of the variable. This value it is whatever our network outputs.

The main goal when training a model is to minimize the loss function, and therefore most ML or DL algorithms use some kind of loss function during their optimization process. In the optimization process, the weights and the biases are adjusted with each training of the model, so that the value of the loss function gets smaller and the NN produces a correct output depending on the given input. It is important to note that the choice of loss function is directly linked to the type of activation function used in the output of the NN.

2.3.4 Optimizers

To decrease the loss function, random changes to the weight and bias parameters may not be the best approach. Therefore, optimizers have been created to modify the attributes of a NN, such as the weights and learning rate, using functions at each epoch (number of times the algorithm runs). These algorithms aim to overcome the problem of choosing the correct parameters for training, which can become complex, especially in models like deep learning, which have thousands of parameters.

Optimizers aim to enhance the process of model improvement by reducing losses, positively affecting the accuracy and speed of training a model. There are several types of optimizers, with the most popular being Gradient Descent, Stochastic Gradient Descent, Adagrad, AdaDelta, and Adam. Understanding each of these algorithms is essential to their correct application in the model.

Adam

Adaptive Moment Estimation (ADAM) is an optimization algorithm that combines the features of the AdaGrad and RMSProp algorithms, extending the Stochastic Gradient Descent (SGD) method for updating the weights of NN during training.

The ADAM optimizer uses not only the first moment (mean) but also the second moment (uncentered variance) of the gradients to adapt the learning rates. Additionally, this algorithm is simple to implement, runs fast, requires less memory, and has fewer parameters than other optimization algorithms. However, Adam may prioritize faster computation time over the number of data points, while other optimizers, such as SGD, can generalize better at the cost of slower computation speed. The bias correction formula (Equation 2.10), where $\hat{m}_t$ and $\hat{v}_t$ represent the first and second moments of the gradients, respectively. The variables β_1 and β_2 correspond to the decay rates of the gradient means [38].

$$\begin{aligned} \hat{m}_t &= \frac{m_t}{1-\beta_1^t} \\ \hat{v}_t &= \frac{v_t}{1-\beta_2^t} \end{aligned} \tag{2.10}$$

Equation 2.11 presents the formula for updating the parameters in a controlled and unbiased manner at each iteration. Here, w denotes the weight of the model and α is the step size, which may depend on the iteration.

$$w_t = w_{t-1} - \alpha \frac{\hat{m}_t}{\sqrt{\hat{v}_t} - \epsilon} \tag{2.11}$$

2.3.5 Types of Neural Networks

NN are non-linear adaptive information systems that incorporate a hugeness of processing units with self-adaptive characteristics. The performance of a NN is influenced by the number of neurons used, as employing a small number may result in a poor approximation, while an excessive number can lead to overfitting issues [39]. Therefore, it is essential to understand the problem's nature and tailor the NN accordingly to prioritize its effectiveness .

Presently, there exist various NN architectures that handle and train data in distinct ways. The following sections will introduce the different types of architectures that can be found within a NN.

Perceptron

The Perceptron model is one of the simplest and oldest NN models. It was proposed by Frank Rosenblatt in 1958. It is the smallest unit of a NN that performs computations to detect features in the input data. It is a supervised learning algorithm that can be used for binary classification problems. A binary classifier is a function that can decide whether an input represented by a vector belongs to a particular class or not [40]. This model introduces the concept, wich is used in all types of NN, of using weighted inputs and applies the activation function to get the output as the final result.

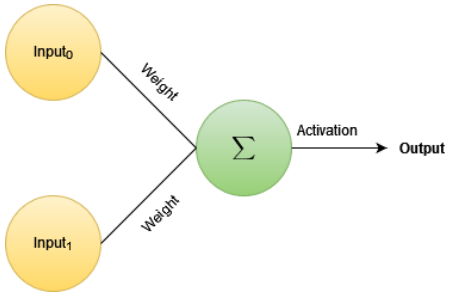


Figure 2.5: Example of Perceptron Network.

Feedfoward Neural Networks

Feedfoward Neural Network (FNN) (Figure 2.6), as the name implies, imposes input data to flow in only one direction, passing through the input neurons, then through the hidden nodes (if present) and exiting through the output node. This NN can be classified as a single-layer or multi-layer FNN depending on whether it has hidden layers or not, but the input and output layers are always present.

In single-layer FNNs, the term "output layer" refers to the layer of nodes (neurons) responsible for generating the final outputs, while the input layer is not considered for computation. Conversely, multi-layer FNNs consist of one or more hidden layers where data is processed before reaching the output layers. In this type of ANN, the weights are fixed, and the values that are fed into the activation function are determined [41]. FNNs are quite simple to maintain and are well-suited for handling noisy data such as faces, patterns, or speech recognition.

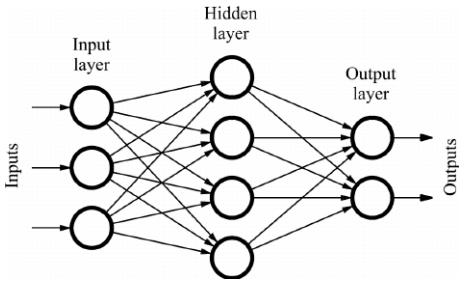


Figure 2.6: Example of Feedfoward Neural Network [42].

Multilayer Perceptron

A Multilayer Perceptron (MLP) is a type of NN in which each neuron is connected to all neurons in the next layer. It consists of multiple hidden layers, with at three layers in total, while the input and output layers are still present. Unlike FNNs, MLPs have bidirectional data propagation, meaning that the data transfer can be both forward and backward. Due to their structure and high complexity, these NN are widely used in DL applications.

MLP consists of a straightforward system comprising interconnected neurons or nodes (Figure 2.7). The nodes are linked by weights, and the output signals are determined by functions that compute the sum of the nodes' inputs, adjusted by a simple non-linear transfer function or activation function. Additionally, a optimizer is used in conjunction with backpropagation to modify the weights and reduce losses.

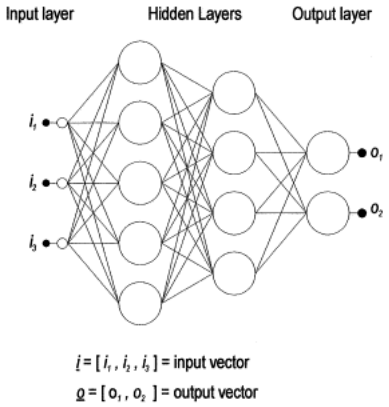


Figure 2.7: Example of Multilayer Perceptron [43].

Convolutional Neural Network

CNN (Figure 2.8) are a type of FNN that transmit information in one direction, but they differ from FNNs in that they use a three-dimensional arrangement instead of a standard two-dimensional array. They are widely used in image classification applications and use the ReLU (Equation 2.7) activation function followed by a softmax [44].

CNNs have multiple convolutional layers, where each neuron in the convolutional layer processes the input data, such as the pixels of an image, and extracts its features through filters and kernels. These filters are small matrices of weights that are applied over the input image to produce a feature map [45]. This process is repeated several times, with the filters being updated through backpropagation, allowing the network to learn and identify different types of features.

After convolution is performed on the first layers, the feature map is transferred to the pooling layer. The purpose of the pooling layer is to reduce the dimensions of the hidden layer by decreasing the size of the feature map. The most common technique for this purpose is max pooling, which keeps the maximum values or largest value of each feature map, discarding the rest [45].

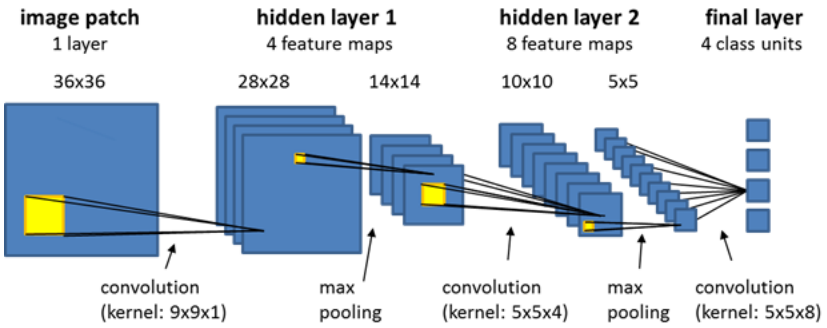


Figure 2.8: Schematic representation of a CNN with two hidden layers [46].

Radial Basis Function Neural Network

The Radial Basis Funcion Network (RBFN) is a NN that consists of an input vector followed by a layer of Radial Basis Funcion (RBF) neurons and an output layer in which each output node determines a category. This classification is performed through the similarity between the input data and the training dataset that each neuron stores, called the prototype. One of the advantages of applications utilizing RBFNs is that they possess a fast, linear learning algorithm within a network capable of representing complex nonlinear mappings [47].

When a vector is inserted into the input layer to be classified, each neuron will calculate the Euclidean distance between the input and its prototype. After this comparison, each RBF neuron (Figure 2.9) will produce a value that defines the similarity measure which will take values from 0 to 1 [48]. For example, if we have two classes, class A and class B, and the new input to be classified is closer to the prototypes of class A than the prototypes of class B, a value close to 1 will be produced, and this input could be classified as Class A. The greater the distance between the prototypes, the response drops exponentially towards 0 [44].

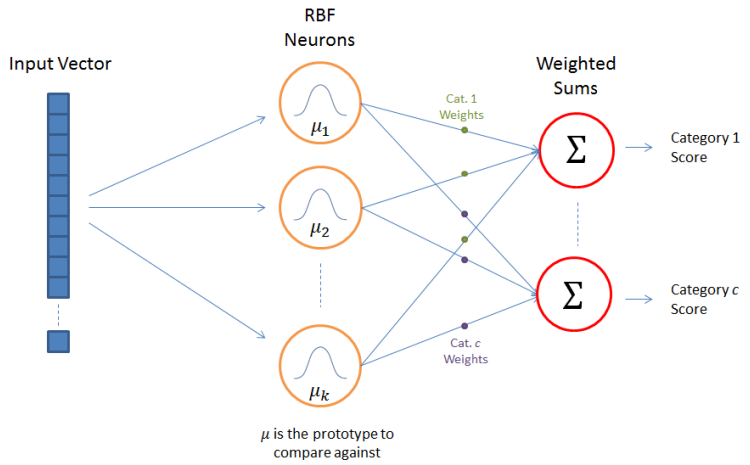


Figure 2.9: Architecture of an RBF Network [49].

Recurrent Neural Network

Recurrent Neural Network (RNN) are specifically designed to learn sequentially or to learn time-varying patterns. A recurrent network is a feedback ANN (Figure 2.10), which can have either fully connected or partially connected architectures. It includes a multilayer FNN with separate inputs and outputs that store essential information for future use. Similar to other NN, backpropagation algorithms are employed to make incremental adjustments to the weights, enhancing the prediction accuracy over time [50].

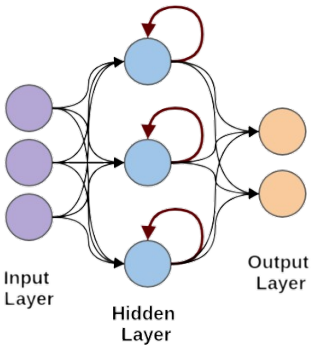


Figure 2.10: Example of Recurrent Neural Network [51].

Long Short-Term Memory Network

Long Short-Term Memory (LSTM) is a type of RNN that processes not only a single data node but a set of data in sequence. LSTM, as their name implies, include a memory cell that can hold information in memory for long periods of time. This type of NN has gates that are used to control when information enters memory

(Figure 2.11), when it is output, and when it is forgotten. The input gate is where it is decided how much information from the last prediction will be stored in memory, the output gate regulates the amount of data passed to the next layer, and the forget gate controls when the data will be forgotten so as not to occupy space in the memory storage [52]. Figure 2.11 shows an example among many of the architecture of an LSTM comparatively to the RNN.

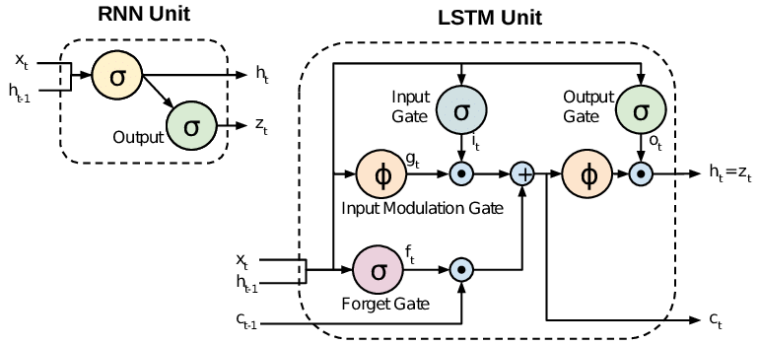


Figure 2.11: Architecture of a Long Short-Term Memory Network versus Recurrent Neural Network [52].

Gated Recurrent Unit (GRU)

GRUs (Figure 2.12), introduced by Junyoung Chung et al. [53] in 2014, were specifically developed to simplify the structure of LSTMs while maintaining comparable or superior performance in natural language processing and temporal sequencing tasks. The objective was to create a neural network that is easier to train and more computationally efficient, achieved by utilizing fewer parameters. GRUs consist of gates, which are processing units responsible for controlling the flow of information within the network. The gates comprise an update gate and a reset gate. These gates play a crucial role in managing how information is updated and maintained across sequences. The update gate determines the amount of information to retain from the previous input and how much new information to consider. On the other hand, the reset gate controls which parts of the past memory should be forgotten. By employing these control mechanisms, the network effectively addresses long-term dependency issues, enabling it to retrieve relevant information even in lengthy sequences.

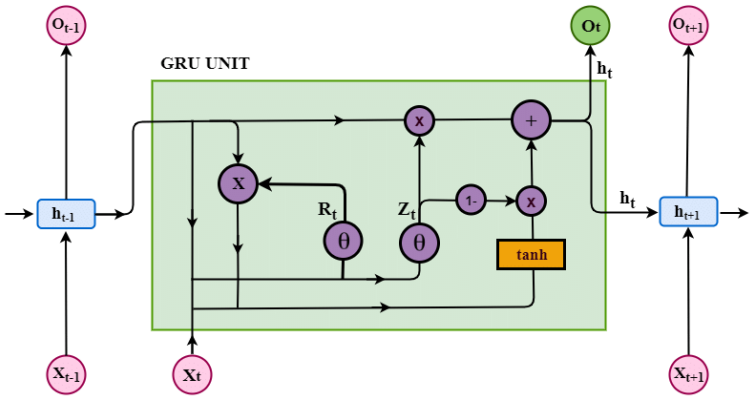


Figure 2.12: Example of a GRU Unit [54].

Sequence-to-Sequence Models

The Sequence-to-Sequence model (Figure 2.13) is a special class of RNN architecture. Uses a encoder-decoder architecture that consists of two LSTM models that work together. The encoder takes in the input sequence and summarizes it into vector of nu that contain all the relevant information from the input elements. The decoder then takes these vectors and generates the output sequence [55].

This model is applicable in cases where the data length is equal to the outputs, and is commonly used in chatbots, machine translation and question and answer systems.

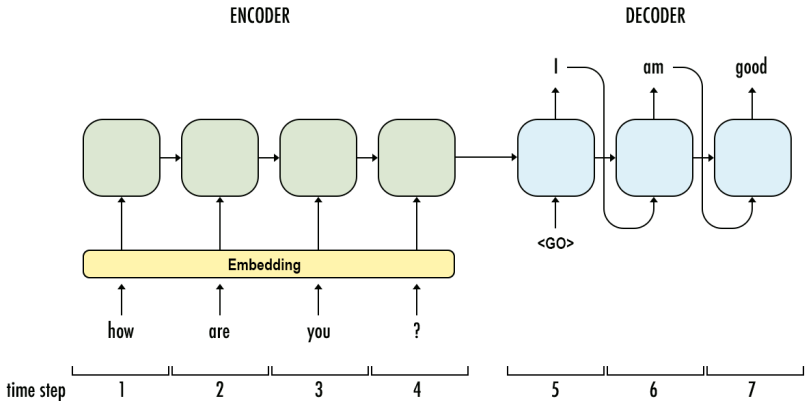


Figure 2.13: Example of Sequence to Sequence Models [44].

Modular Neural Network

Modular NNs (Figure 2.14) consist of a number of different NNs that function independently and perform a subtask. This NN has an input layer where the input data is entered. In the task division layer, the NN subdivides the tasks and data to be sent to the next layers. The sub-network layer is where the data will be treated

independently in a type of NN to obtain, in its output, the result of the sub-task to which it was submitted. Finally, in the output combination layer, the modular NN combines the results of all sub-tasks obtained in the sub-network layer and obtains the prediction.

The greatest advantage of using modular NNs is their processing speed, since their set of NNs does not interact with each other, thus not causing a dependence of results between networks. Additionally, each NN inserted in the modular NN will perform a simpler task.

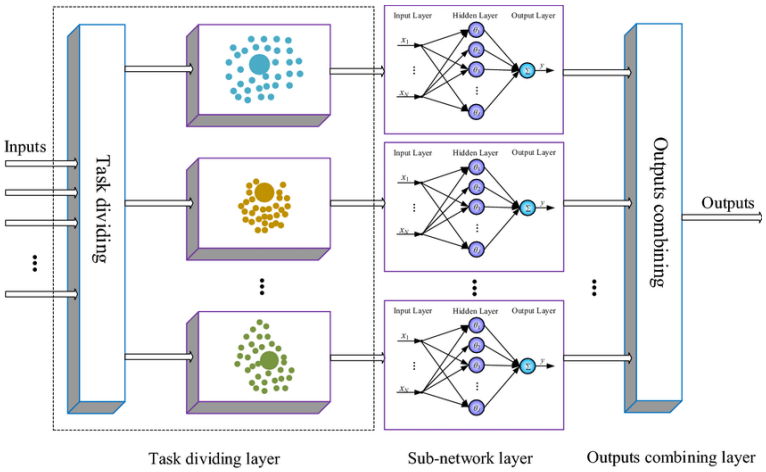


Figure 2.14: Structure of Modular Neural Network [56].

Summary of Neural Network

Based on the information presented, there are various types of NNs that vary in their architecture and training methods. Table A.1 summarizes the significant advantages and disadvantages of using the mentioned types of NNs. Therefore, a particular type of NN may have better precisions compared to another depending on the type of data and its applicability. As mentioned previously, it is crucial to determine the complexity of the problem and select the appropriate technique accordingly. When the data is labeled, the prediction task can be categorized as a "classification problem", and there are several types such as:

- **Binary Classification** - This type of problem involves classifying data into two distinct groups or categories. In binary classification, the data is commonly labeled as 0 or 1, representing false/negative or true/positive, respectively. The model is trained to predict whether an item belongs to one of the two classes.

- **Multi-Class Classification** - This type of problem classifies data into three or more groups or categories. Unlike binary classification, the model is trained

to predict whether an item belongs to one of the three or more classes present. For example, data may be labeled with integers, where each group represents a specific number (0, 1, 2, 3, etc.).

- **Multi-Label Classification** - This type of problem allows an item to belong to multiple groups or categories simultaneously. For instance, a multi-label classifier can be used to classify whether a movie poster belongs to the action, adventure, and fantasy genres simultaneously.

2.3.6 Performance Analysis

To perform the evaluation of a ML model, a range of evaluation metrics can be used. These metrics are the essential part not only to evaluate whether the algorithm is performing well but also to adjust its parameterization in order to improve the ML model. In this thesis, only metrics related to classification problem will be addressed since the model to be developed will need to automatically detect and classify vulnerabilities in code. This way, metrics such as precision, accuracy, recall, confusion matrix, AUC curves, F1-Score, MCC (Matthews Correlation Coefficient) will be addressed.

Confusion Matrix

The Confusion Matrix is a tabular representation that captures the occurrences between two raters: the true/actual rating and the predicted rating [57]. This metric uses the following terminology to evaluate the model's predictions:

- True Positive (TP) - A prediction that correctly indicates that a data set belongs to a group or category.
- True Negative (TN) - A prediction that correctly indicates that a data set is not in a group or category.
- False Positive (FP) - A prediction that incorrectly indicates that a data set is present in a particular group or category.
- False Negative (FN) - A prediction that incorrectly indicates that a data set is not present in a given group or category.

This matrix is constructed by calculating statistical measures such as TP, TN, FP, and FN (Table 2.1) based on correlation terms [58]. It is important to note that a ML model does not produce true or false output, but rather a continuous probability value that is later converted to positive or negative using a threshold of 0.5 in many cases. Understanding these terms is critical to calculating the remaining metrics such as accuracy, precision, recall, and most importantly, the AUC curves.

Table 2.1: Confusion Matrix and its analogies.

		Actual Values	
		Positive (1)	Negative (0)
Predicted Values	Positive (1)	TP	FP
	Negative (0)	FN	TN

Precision and Recall

Precision is a metric that determines the proportion of positive predictions that are actually true positives and can be calculated by the following formula:

Precision (Equation 2.12) is defined as the proportion of true positives to the sum of true positives and false positives. This metric provides insights into the reliability of the model's positive predictions, indicating how much confidence we can have in the model when it classifies a data point as positive [57].

$$Precision = \frac{TP}{TP + FP} \tag{2.12}$$

Recall, also known as sensitivity or TP rate, is a metric that determines the proportion of true positives that are correctly predicted by the model and is given by the Equation 2.13 [57]:

$$Recall = \frac{TP}{TP + FN} \tag{2.13}$$

Both precision and recall should be as high as possible, but they are often trade-offs, meaning that improving one may come at the cost of decreasing the other. Therefore, it is important to find a balance between the two depending on the specific problem and its requirements.

Accuracy

Accuracy refers to the sum of the number of correct predictions at the numerator and the total number of input data [57]. This metric will only work correctly if there is an equal number of input data belonging to different classes. The accuracy of a model can be calculated by the Equation 2.14:

$$Accuracy = \frac{TP + TN}{TP + NP + TN + FN} \tag{2.14}$$

F1-Score

The F1-Score is the harmonic weighted mean of precision and recall, where the best value of F1-Score are 1 and the worst are 0. This metric allows for comparison of two models with high precision and low recall, and vice versa [57]. The F1-Score evaluates how robust and accurate the model is by analyzing the number of instances that were not missed and the number that were correctly classified. Mathematically, this metric can be expressed as:

$$F1 = 2 \times \frac{Recall \times Precision}{Recall + Precision} \tag{2.15}$$

Matthew's Correlation Coefficient

MCC is a metric utilized to evaluate the performance of ML models in predicting multi-class problems. The MCC ranges between -1 and 1, with values close to 1 indicating excellent prediction performance, signifying a strong positive correlation between the predictions and the true labels. Conversely, a value of 0 suggests no correlation between the variables, indicating that the model assigns data to classes randomly without any connection. A negative MCC value signifies an inverse relationship between the true classes and the predicted classes [57]. The formula can be expressed as:

$$MCC_{multi-class} = \frac{c \times s - \sum_k^K p_k \times t_k}{\sqrt{\left(s^2 - \sum_k^K p_k^2\right)\left(s^2 - \sum_k^K t_k^2\right)}} \tag{2.16}$$

- k - Class Number;
- $c = \sum_k^K C_{kk}$ - Number of elements correctly predicted;
- $s = \sum_i^K \sum_j^K C_{ij}$ - Total number of elements;
- $p_k = \sum_i^K C_{ki}$ - Number of times that class k was predicted;
- $t_k = \sum_i^K C_{ik}$ - Number of times that class k was correctly ocurred.

MCC is a straightforward metric used for evaluating the performance of a classifier. This metric places emphasis on accurately predicting both positive and negative cases, irrespective of the dataset's balance. Unlike Accuracy and F1-score, which perform well on balanced datasets but can be misleading when dealing with imbalanced data, MCC provides a reliable measure of classifier performance in various dataset distributions [59].

AUC-ROC

The AUC-ROC represents the Area Under the Curve (AUC) of the Receiver Operating Characteristic (ROC) curve (Figure 2.15). The ROC curve is an evaluation metric for binary classification, which is a probability curve that plots the FP rate against the TP rate on the x and y axes, respectively. In other words, this curve shows the performance of a classification model at all classification thresholds. AUC, on the other hand, is a metric that evaluates the overall performance of the model to distinguish between classes [60].

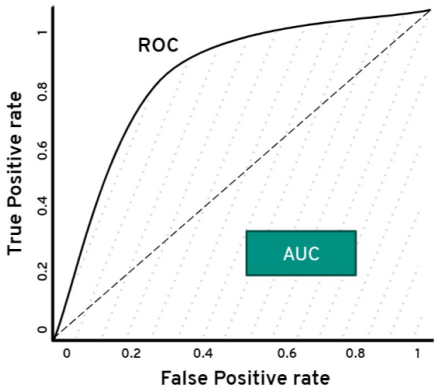


Figure 2.15: Illustration of AUC-ROC Curve [61].

2.4 Related Works

This section presents a collection of works related to the main theme of this thesis. It explores the literature that supports the techniques and methodologies used to detect vulnerabilities in source code using AI, ML or DL, as well as other works related to the detection of vulnerabilities or exploits that can serve as study material to provide a more comprehensive view of the topic. This literature review is essential for understanding the initial steps and the tools available, which will be covered in the following chapters.

The last section provides a brief summary of both the literature and the datasets used. This section serves as a comparison between the different techniques used by the same or different datasets and aims to clarify the choices made in the implementation.

2.4.1 Literature Review

Automated Vulnerability Detection in Source Code

Russell et al. [19] presents the potential of using Machine Learning to detect software vulnerabilities directly from source code. To achieve this, an extensive dataset was created that gathers a set of functions written in C/C++ that constitute the source code from repositories of Debian, Github, and SATA IV Juliet Test Suite. All uncategorized functions were subsequently categorized using a static analysis toolkit.

The method used directly interprets the lexed source code and sends it through a CNN or RNN. The data then runs through a custom lexer that represents the source code in a 156-token vocabulary, filtered by removing comments and matching similar words. The lexed functions were embedded in an l x k matrix, where l is the number of tokens and k is a random vector of fixed length. This matrix is then used as initial embeddings for the CNN and RNN models. However, the use of NN to classify vulnerabilities did not meet expectations and had worse results than if feature engineering was used. In order to evaluate the performance of the model, Random Forest (RF) algorithm was also trained using a Bag-of-Words (BoW) representation.

In terms of results, a variety of model combinations were used to determine the most optimal approach. These combinations included BoW + RF, RNN, CNN, RNN + RF, and CNN + RF. Among these, the model that showed the most promising performance was CNN + RF, using combined data from the Debian and Github repositories. It achieved an F1-Score of 0.566 and a ROC-AUC value of 0.904. In contrast, the BoW + RF combination achieved an F1-Score of 0.498 and a ROC-AUC of 0.883. The CNN model performed best on the SATE IV data with an ROC-AUC of 0.954 and an F1-Score of 0.840. In comparison, the BoW + RF combination had an AUC-ROC of 0.913 and an F1-Score of 0.786 for the same data. Looking at the RNN models alone, the RNN + RF combination showed the best results with a ROC-AUC of 0.899 and an F1-Score of 0.552. The superior results in SATE IV compared to Debian + Github can be attributed to the richer dataset of SATE IV, which has a larger number of examples for each vulnerability, as well as more consistent styles and structures.

The results were satisfactory since they proved that the use of ML techniques in vulnerability detection is more effective than the use of static analyzers, as in the case of the Clang tool [62]. Although the Clang tool performs better on SATE IV data, it still finds very few vulnerabilities compared to the different ML methods.

Vulnerability Prediction From Source Code Using Machine Learning

Z. Bilgin at al. [63] presented an approach to represent source code in an Abstract Syntax Tree (AST) format and then used a CNN to distinguish between vulnerable and non-vulnerable code. The AST representation was designed to create a structured format of source code that is appropriate for use in other ML processes. To create the AST, the source code was transformed into a binary tree where each internal node had two children so that all leaf nodes had the same depth. This was necessary to ensure that all functions had the same size. The resulting binary tree was then converted into a depth-ordered vector of tokens and subsequently transformed into a three-dimensional numeric vector using token embeddings. The depth of the binary tree was set to 8 for performance reasons.

The dataset used in this study was the same as the one used by Russell et al. [19] and consisted of five different types of predetermined vulnerabilities. The results showed promising outcomes, although it was observed that some types of vulnerabilities were more difficult to detect than others.

SySeVR: A Framework for Using Deep Learning to Detect Vulnerabilities

Z. Li et al. [64], [65] investigate whether the inclusion of semantic code information could help detect vulnerabilities related to specific library/API calls in the source code. To do this, the classification using only data dependency features was compared with the simultaneous use of control dependency and data dependency features. The open-source tool Joern [66] was used to analyze programs and generate various features, including the identification of library/API function calls and the generation of code gadgets from program slices. The code gadgets were then vectorized and tested with different NNs to find the best option for detecting vulnerabilities. The authors concluded that using both control and data dependency is more effective than just data dependency. The highest performance was achieved by a BLSTM NN, which achieved an F1-Score of 0.924 on a dataset of 68353 code gadgets, of which 13686 were labeled as vulnerable. Various methods for dealing with imbalanced data were tested, but the conclusion was that not using any sampling yielded the best results.

μVulDeePecker: A Deep Learning-Based System for Multiclass Vulnerability Detection

D. Zou et. al. [67] proposes a deep learning-based system called μVulDeePecker [68] for the detection of multiple classes of software vulnerabilities. This work is inspired by VulDeePecker [65], which is the first binary vulnerability detection system based on deep learning, and it refines the code gadget concept. The system uses a BLSTM networks to automatically learn features from the input source code

and detect vulnerabilities. μVulDeePecker is not merely an extension of VulDeePecker, as it was found that it is not effective to use VulDeePecker for multiclass vulnerability detection. As a result, two variants of VulDeePecker were considered. The first variant, called VulDeePecker+, directly modified the architecture of the NN. The other variant consisted of training a separate VulDeePecker classifier for each vulnerability type, and each model is applied to each sample for vulnerability detection. The latter variant has significant weaknesses, as it is not scalable for various types of vulnerabilities and is not effective when the samples are small.

μVulDeePecker consists of six modules. The first module is the data preprocessing module, which is responsible for cleaning and normalizing the input data to reduce noise and increase the accuracy of vulnerability detection when creating the code gadgets to build the PDGs. The second module is responsible for classifying the data collected by NVD and SARD into vulnerable when the vulnerability i is between the range of $1 < i < 40$, otherwise, i takes a value of 0, and the code gadget is classified as not vulnerable. In the third module, Zou et al. [67] write an automatic program based on lexical analysis to analyze each token of the classified data. Then, vectors are generated to represent normalized code gadgets using a word-to-vector model with hyperparameters $\tau 1$ and $\tau 2$ for gadget and attention length, respectively. Padding or cutting is used to ensure that vectors have a fixed length of $\tau 1$ or $\tau 2$. The BLSTM is used to build the μVulDeePecker NN architecture, which is trained using Keras with hyperparameters optimized via grid search. The global and local feature learning models are trained separately to prevent the destruction of features learned from one network by the others.

The data used in this work was the same as that used by Z. Li et al. [64] and the μVulDeePecker system achieved a maximum F1-Score of 0.9628, indicating its effectiveness and improvement over its predecessor system.

An Automatic Source Code Vulnerability Detection Approach Based on KELM

G. Tang et. al. [69] proposed to use an Extreme Learning Machine (ELM) in conjunction with the kernel method to improve the accuracy of the vulnerability detection model and solve the problem of training effectiveness. Similar to SySeVR [64] and μVulDeePecker [67], the dataset used was in the form of code gadgets and was converted into a symbolic representation through multiple levels. This symbolic representation was then converted into a vector representation using doc2vec. Finally, these vector representations were used to train the ELM NNs to detect vulnerabilities. An ELM NN is a type of FNN that uses a non-iterative training mechanism.

The results of this study showed that combining ELM with the kernel method improved both efficiency and accuracy, with a maximum F1-Score of 0.923 achieved on the RM-ALL dataset. Additionally, the study found that vector

representation using doc2vec worked well for large datasets, and appropriate symbolization levels could effectively improve the accuracy of vulnerability detection.

Devign: Effective Vulnerability Identification by Learning Comprehensive Program Semantics via Graph Neural Networks

Y. Zhou et. al. [70] introduced a model called Devign, which uses source code functions converted into graphical structures to learn vulnerability patterns through graphical NNs. To address the issue of dataset inaccuracies resulting from classifications done by static tools, a security team created a new dataset from scratch and manually labeled C functions collected from open-source projects such as Linux Kernel, QEMU, Wireshark, and FFmpeg, which took approximately 600 man-hours. They then used an open-source graph-based code property tool, Joern [66], to extract ASTs and CFGs for all functions in the dataset.

The Devign architecture consists of a graph layer that represents the semantics of the composite code, which encodes the raw source code of a function into a joint graph structure with the semantics of the overarching program. The Gated Graph Recurrent layer then learns node characteristics by aggregating and transmitting information about neighboring nodes in graphs. Finally, the Conv module extracts significant node representations for graph-level prediction. The results of the work show that Devign with comprehensive graph-encoded semantics outperforms most modern vulnerability identification methods, achieving a maximum F1-Score of 84.97 on Linux function data that follows best practices and coding styles. Regarding the FFmpeg dataset, the model obtained a maximum F1-Score of 0.3492, which is lower than the F1-Score obtained on the Qemu dataset [71], where the model achieved a score of 0.4112.

DeepVulSeeker: A Novel Vulnerability Identification Framework via Code Graph Structure and Pre-training Mechanism

J. Wang et. al. [72] propose a model called DeepVulSeeker, which is based on heterogeneous structural information with MetaPaths, pre-trained semantic models, and a GRSA encoding network. DeepVulSeeker captures the semantic relationship between each programmatic node and its adjacent nodes with the help of a pre-trained model based on a multi-layer bidirectional transformer called UniXcoder. This model then uses semantic information to obtain the structural representations of the code by means of a data flow graph (DFG), control flow graph (CFG), and Abstract Syntax Tree (AST). Finally, DeepVulSeeker combines the semantic and structural information obtained and feeds it to a CNN and a FNN to predict whether a piece of code is vulnerable. The authors used the Qemu and

FFMPEG dataset, which consists of 18519 functions, of which 8125 are vulnerable and classified according to their CWE.

In terms of results, the model achieved a maximum F1-Score and Precision of 0.99 on the dataset of vulnerabilities classified as CWE-754, with CWE-476 having the worst results with 0.8052 and 0.9080, respectively. In the FFMPEG datasets, the model had better and acceptable performance than Qemu dataset, with a acurracy and F1-Scores of 0.6354 and 0.6703, respectively, which were superior to other known models.

Systematic Analysis of Deep Learning Model for Vulnerable Code Detection

B. Nazim et. al. [73] presented a review and analysis of existing models and approaches for detecting vulnerabilities in source code written in C and C++. The paper provides a brief overview of key topics for comparing different models, such as source code representation, NN architecture, and datasets used. The models are then compared based on their parameters, including the type of NN used, vector representation, source code representation, dataset used, and feature representation.

B. Nazim et. al. [73] concludes that while progress has been made in detecting vulnerabilities in source code, the field is still in its early stages and many problems remain unsolved. However, this highlights the need for further research in this area to overcome these problems and develop more accurate and effective solutions.

2.4.2 Datasets

In the context of detecting vulnerabilities in source code, the use of datasets is essential for training and evaluating ML models. In this section, we will present an overview of the datasets used in the literature discussed above. We will highlight the main characteristics of each dataset, such as the size, the types of vulnerabilities present, and the sources from which the code was obtained. Understanding the characteristics of these datasets is crucial for assessing the effectiveness and limitations of the various ML models developed for vulnerability detection.

Draper VDISC

Draper VDISC [74] is public dataset created in 2018 gathering 1.27 million functions written in C/C++. The dataset has been mined through open source software and labeled using static analysis programs. The data is stored in three HDF5 files, one of which corresponds to 80% of the functions and is used for training. The remaining two correspond to 10% of the functions each and are used for validation and testing. This division follows the approach used in the work by Russel et. al. (2018) [19]. The dataset has a total size of about 1 GB.

Each function in the dataset is stored as a UTF-8 string of variable length, starting with the function name. The dataset also includes five binary labels representing detected vulnerabilities, namely CWE-119, CWE-120, CWE-469, CWE-476, and CWE-other. The distribution of functions and their vulnerabilities in the dataset is shown in Table 2.2 [74].

Table 2.2: Number of functions distributed among the labels in the Draper VDISC dataset [74].

Label	**No. of Functions (%)**
CWE-119	1.9
CWE-120	3.7
CWE-469	0.95
CWE-476	0.21
CWE-Other	2.7

The functions included in this dataset were sourced from a variety of open-source projects, including Debian Linux, Github public repositories, and the SATE IV Juliet Test Suite of NIST Samate Project. The first two projects provide real-world data, while the latter is synthetic. Functions from Debian Linux are generally of high quality, demonstrating good coding practices and style. In comparison, functions from Github often exhibit worse quality and a wider range of vulnerabilities. The functions from the SATE IV Juliet Test Suite were specifically designed to contain 118 different examples of CWE vulnerabilities [73].

SARD & NVD

The Software Assurance Reference Dataset (SARD) [75] contains various code excerpts written in synthetic, real, academic security C/C++ and from the National Vulnerability Database (NVD). The dataset comprises a total of 61638 code gadgets, out of which 17725 are considered vulnerable and 43913 are considered not vulnerable. Among the vulnerable code gadgets, 10440 correspond to buffer-related errors, and the remaining 7285 represent vulnerabilities associated with resource mismanagement [76].

In the μVulDeePecker work [67], the dataset was modified by extracting 33409 test cases that generated 181641 code gadgets, covering 40 different types of vulnerabilities. These code gadgets are composed of multiple program statements and may or may not have a direct relationship and dependency with their library/API. Out of all the code gadgets, 138522 represent code that is not vulnerable, while 43119 represent code that is vulnerable [68].

FFMpeg + Qemu

The FFMpeg + QEMU dataset contains C source code functions from open-source functions such as Linux Kernel, FFmpeg, Qemu, and Wireshark. These functions were extracted from a set of commits made over the years from these projects, except for Linux Kernel, which includes only commits between 2016 and 2017.

The dataset is balanced, with almost the same number of vulnerable and non-vulnerable features. It comprises 48687 commits with confirmed labels, with 12811 from Linux Kernel, 13862 from FFmpeg, 11910 from Qemu, and 10004 from Wireshark, as shown in Table 2.3. The labels were assigned manually with the help of a security team, as also mentioned by Y. Zhou et al. (2019) [70].

Table 2.3: Overview of FFMpeg + Qemu Dataset used in Devign Model [70].

Project	Sec. Real Commits	VFCs	Non-VFCs	Graphs	Vul Graphas	Non-Vul Graph
Linux	12811	8647	4164	16583	11198	5385
Qemu	11910	4932	6978	15645	6648	8997
Wireshark	10004	3814	6190	20021	6386	13635
FFmpeg	13962	5962	8000	6716	3420	3296
Total	48687	23355	25332	58965	27652	31313

After manual validation, the dataset was found to contain 23355 vulnerability-fix commits, each representing one vulnerability, with a few rare exceptions of multiple vulnerabilities. Most of the vulnerabilities in this dataset are memory-related, such as buffer overflow or memory leak.

The dataset is publicly available in an unlabeled version, with the functions stored in a JSON file that includes a binary label indicating whether or not it has vulnerabilities. Additionally, the FFmpeg and Qemu functions are available in two CSV files with one binary label indicating whether the function is vulnerable and another indicating the ID [71].

ReVeal

The ReVeal is a public dataset that created in the work developed by S. Chakraborty et. al. (2021) [77] with the goal of addressing the limitations and lack of precision of existing datasets used for detect vulnerabilities in code written in C/C++ languages. This dataset exclusively utilizes real data, tracking past vulnerabilities from two open-source projects: the Linux Debian Kernel and Chromium. Additionally, a portion of this dataset is composed of functions present in the Devign dataset [78].

To curate the data, they used the publicly available data in each patch. In the case of the Linux Debian Kernel, bug information from the Debian security tracker was used, while for Chromium, information present in the Bugzilla bug repository

was used. For functions with vulnerabilities, both their bug and fixed versions were extracted. Vulnerable functions were annotated as "vulnerable", and their fixed versions were annotated as "clean". Functions that did not contain any vulnerabilities were also annotated as "clean". This strategy simulated a real-world scenario in which a deep learning model could inspect vulnerable functions in the context of all other functions in its scope.

2.4.3 Summary

This section presents a comparative analysis of the various techniques used in the literature mentioned in section 2.4.1. These works use various types of ML techniques, as well as pre-processing and representation methods. Therefore, it is essential to understand their strengths and weaknesses in the context of their approach, and it is crucial for new ideas to emerge. Table 2.4 presents a direct comparison between the different approaches used in the annotated literature.

Table 2.4: Literature Comparison

Author	Dataset	Feature Representation	Source Code Representation	Vector Representation	Models	F1-Score (Max)
Russell et. al. [19]	DRAPER VDISC	Token	NLP	word2vec	CNN + RF, RNN + RF	0.566
Z. Bilgin at al. [63]	DRAPER VDISC	Token	Binary AST	Array Representation	MLP, CNN	0.509
Z. Li et al. [64]	SARD & NVD	Token	Code Gadgets	word2vec	BLSTM, BGRU	0.858[a]
D. Zou et. al. [67]	SARD & NVD	Token	Code Gadgets + Code Attetion	word2vec	BLSTM	0.9628[a]
G. Tang et. al. [69]	SARD & NVD	Token	Code Gadgets	doc2vec	KELM	0.923[a]
Y. Zhou et. al. [70]	FFMpeg + Qemu	Graph	AST + CFG + DFG + NCS	word2vec	GGNN	0.4112
J. Wang et. al. [72]	FFMpeg + Qemu	Graph	AST + DFG + CFG + GRSA	word2vec	BLSTM, CNN	0.6703

[a] The F1-Score values represent the results of an experiment aimed at answering a specific question proposed by the authors, and not the overall performance of the model on the dataset.

As the focus of this thesis is on identifying and classifying vulnerabilities, the lack of classification in public datasets presents a drawback. While it is possible to label them, existing tools are not highly accurate, and manual labeling would be impractical due to the time required. Nonetheless, the use of these datasets should not be ruled out, as they can be used to test the model after it has been finalized.

According to the literature, most common features are not very effective when compared to more complex representations, such as lexed or graph-based representations. However, the graph-based approach may not be suitable as it requires the entire context, including all functions that are directly and indirectly linked to the code snippet. On the other hand, using a lexed representation does not have this issue and can yield similar results. Therefore, for achieving the objectives of this thesis, a lexed-based approach may be the best option since it detects vulnerabilities based on the semantics of the text.

Chapter 3

Data Analysis & Preprocessing

This chapter presents the analysis performed and the corresponding preprocessing techniques applied to the Draper VDISC dataset selected for training the ML model. This is one of the most important steps in the entire process as it involves the elimination of irrelevant data for the identification and classification of vulnerabilities in the code. By removing such noise data, the dataset used to train the AI model becomes more refined, thereby enhancing the performance of the developed model.

Figure 3.1 illustrates the initial stage of this work, highlighting the primary preprocessing techniques that will be discussed in the following sections.

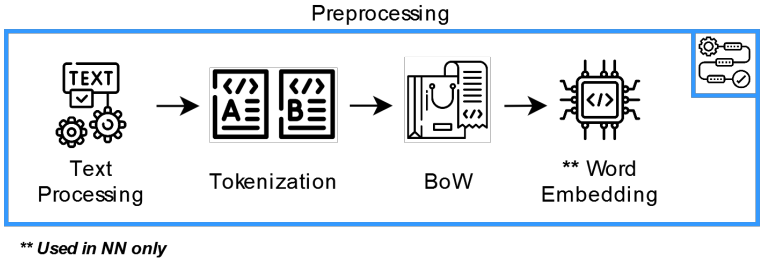


Figure 3.1: Total Vulnerable and Non-Vulnerable Functions per Label.

3.1 Dataset Selection

Since the objective of this study is solely to identify and classify vulnerabilities, there is no requirement to create a dataset comprising generated or collected data. Following the literature review in Section 2.4.2, several datasets were identified for potential utilization in this study, along with their respective advantages and disadvantages. Table 3.1 summarizes the information about the datasets, including the relevant characteristics of each dataset that are important to the development of this work.

Table 3.1: Datasets Comparison.

Dataset	Access	Type of Data	Code Representation	Labels	Data Vulnerable (%)	Number of Data
Draper VDISC	Public	Real and Semi-Synthetic	Source Code (C/C++)	Yes	9.5	1 274 366
SARD & NVD	Public	Semi-Synthetic and Synthetic	Code Gadgets	No	31	61 638
FFMpeg + Qemu	Public	Real	Source Code (C/C++)	No (In Public Version)	45.6	27 358 (In Public Version)
ReVeal	Public	Real	Source Code (C/C++)	No	9.16	18 169

According to the literature, common features used to detect vulnerabilities on code are less effective compared to more complex representations such as lexical or graph-based approaches. Features are attributes extracted from code that ML models use for prediction. Common features may contain all the raw source code, while feature representation is how these attributes are encoded for the model to understand. The graph-based approach relies on a complete understanding of the context surrounding a given snippet of code. This includes all functions that are directly and indirectly connected to the code snippet. In other words, the approach must consider the entire network of relationships between different parts of the code base in order to effectively identify vulnerabilities.In contrast, using a lexical representation doesn't face this particular challenge. It can produce comparable results without the need for such extensive context. Therefore, in the context of achieving the goals set forth in this thesis, opting for a lexed-based approach may be the most appropriate choice. This is because this approach is able to detect vulnerabilities by focusing on the semantics of the text, without having to consider the intricate web of code relationships that the graph-based approach relies on.

Based on this analysis, the Draper VDISC dataset comes closest to meeting the needs of this work. Once the text semantics-based approaches approach is chosen, this dataset remains an appropriate choice because it contains only functions along with their source code, enabling a more comprehensive analysis. The public version of the dataset contains a larger number of functions compared to the others identified, and each function is already classified into five vulnerabilities. Since this dataset contains both the full code of synthetic functions and functions from real projects, it covers different scenarios and code syntaxes, as well as different types of vulnerabilities. This ensures that the features that can be extracted from this dataset are diverse. According on the information presented in Table 3.1,

it is evident that the proportion of data corresponding to vulnerabilities is only 9.5%. However, even though this percentage is relatively small, it exceeds the corresponding percentages of other datasets when considering the total number of vulnerabilities.

3.2 Data

As previously mentioned, the Draper VDISC will serve as the primary dataset for this study. This dataset was used by R. Russell et al [19] and was designed to identify and classify vulnerable functions. This dataset consists of 1.27 million functions and is labeled with their corresponding CWE numbers. The labeling process was performed using static code analysis tools such as Clang, Cppcheck, and Flawfinder. In addition, R. Russell et al. [19] engaged a team of cybersecurity researchers to ensure accurate correlation of each vulnerability with its associated CWE. While this method could potentially introduce incorrect labels into the samples, alternatives such as using pull requests and manual labeling were considered. However, these alternatives proved difficult to automate and time-consuming. Despite this limitation, Russell et al. [19] demonstrated that the use of this dataset remains effective for training models. The dataset has also undergone further refinement, including data curation that eliminates duplicate functions that could lead to model overfitting. In addition, functions that compile to the same set of operation code have been removed to improve the quality of the dataset.

The 1.27 million functions are from both Github repositories and Debian Linux source code. Of these, only four types of CWE were individually tagged, while the rest were grouped under a single tag. This dataset has a significant class imbalance, within the 9.5% of vulnerabilities identified, only 6.5% of the dataset consists of functions that have one or more vulnerabilities. However, this distribution closely reflects real-world scenarios.

In addition, the dataset was pre-split into training, validation, and test sets using an 80:10:10 split. Table 3.2 provides an overview of the vulnerabilities within each of these releases.

Table 3.2: Distribution of Vulnerabilities Among Draper VDISC Versions.

Vulnerability	Train	Validade	Test
CWE-119	1.9%	1.9%	1.9%
CWE-120	3.7%	3.7%	3.8%
CWE-469	0.2%	0.2%	0.2%
CWE-476	1.0%	0.9%	0.9%
CWE-Others	2.7%	2.8%	2.7%

3.2.1 Dataset Labels

This section describes the five categories of CWEs that make up the Draper VDISC labels, which will be the focus of our model classification. A proper understanding of what the syntax of a CWE looks like can help in preprocessing the source code.

CWE-119

CWE-119, example on Listing 3.1, is categorized as a failure to restrict operations within the bounds of a memory buffer. This vulnerability arises when the programmer neglects to validate whether an index request falls within acceptable limits. Consequently, an attacker can exploit this oversight by making an out-of-bounds request, leading to an over-read of the buffer.

```
1 int main (int argc, char **argv) {
2     char *animals[] = {"cat", "dog", "horse",
          "cow"};
3     int index = UntrustedOffset();
4     printf("You have %s\n", animals[index-1])
          ;
5 }
```

Listing 3.1: Example of CWE-119.

CWE-120

CWE-120 is directly associated with buffer overflows and represents the most prevalent vulnerability of this kind in C/C++ programming. This vulnerability occurs when the programmer does not check the size of the input and uses more data than has been allocated in memmory. Listing 3.2 provides an example of a CWE-120 vulnerability.

```
1 void string_change(char * string){
2     char buf[24];
3     strcpy(buf, string);
4 }
```

Listing 3.2: Example of CWE-120.

CWE-469

CWE-469, on Listing 3.3), is a vulnerability that arises when a pointer is subtracted by a certain size. In this type of vulnerability, if the pointers that are being subtracted are not within the same memory space, the calculation can yield incorrect results, potentially enabling the execution of arbitrary code with the privileges of the vulnerable program.

```
1  int size(struct node* head) {
2      struct node* current = head;
3      struct node* tail;
4      while (current != NULL) {
5          tail = current;
6          current = current->next;
7      }
8
9      return tail - head;
10 }
```

Listing 3.3: Example of CWE-469 [79].

CWE-476

CWE-476 is a vulnerability that occurs when using *NULL* pointers, leading to a *NULL* pointer dereference. A example of CWE-476 can be see in Code Snippet 3.4.

```
1 int function(int *pnt, int num){
2     int a = *pnt;
3     return a * num;
4 }
```

Listing 3.4: Example of CWE-476.

CWE-Others

Russell et al. [19] define CWE-Others as any remaining functions that have been identified as vulnerable by static analyzers such as CWE-20, CWE-457, CWE-820, etc. This label also includes functions that have been identified with CWE-563, which is associated with bad code practices but does not cause a security vulnerability. Therefore, based on the information available, there may be other vulnerabilities in addition to those listed above, so we cannot state with certainty all the types of vulnerabilities present in this label.

3.2.2 Data Overview

In this section, an analysis is performed to provide an overview of the labels in the Draper VDISC dataset, considering all of their versions together, which we will refer to as "Combined". Tables 2.2 and 3.1 presented the percentages of vulnerable and non-vulnerable data in the dataset, but this section will present numerical values. To accomplish this, graphs will be presented to depict the distribution and quantity of data in each of the labels, enabling a more comprehensive understanding of the dataset being studied.

Figure 3.2, presents an overview of the dataset composition. The green bar represents the number of functions with vulnerabilities, totaling 121 564 data points. Conversely, the orange bar represents the number of functions without vulnerabilities, corresponding to 1 191 955 data points. This graph provides a visual representation of the ratio between vulnerable and non-vulnerable functions, indicating that there are approximately 9.8 non-vulnerable functions for each vulnerable function.

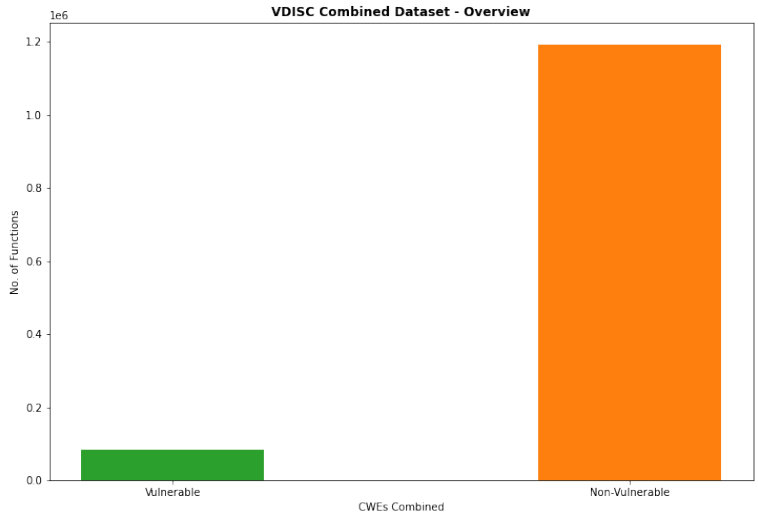


Figure 3.2: Total Vulnerable and Non-Vulnerable Functions.

Figure 3.3 provides an overview of the distribution of data across the five main vulnerability types (CWE-119, CWE-120, CWE-469, CWE-476, CWE-Other) that make up the labels in this dataset. Each vulnerability is represented by two bars, indicating the number of functions classified as vulnerable or not vulnerable. The green bar represents the functions classified as vulnerable, while the orange bar represents the functions classified as not vulnerable. Table 2.2 shows the number of functions as a percentage.

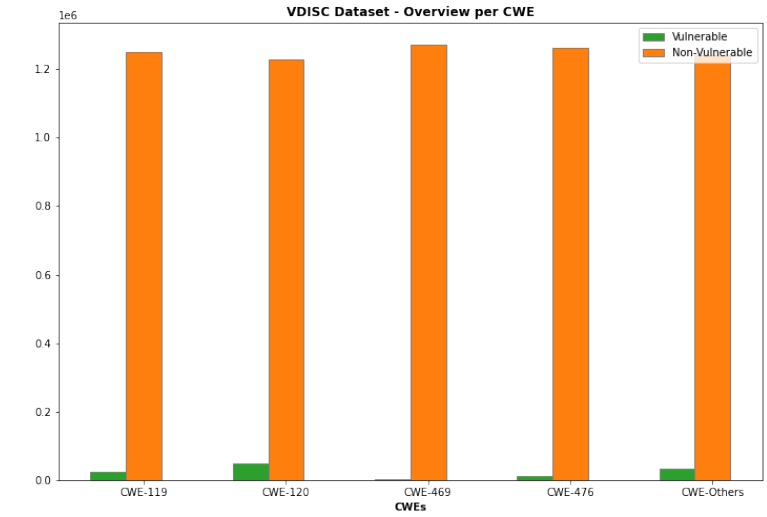


Figure 3.3: Total Vulnerable and Non-Vulnerable Functions per Label.

To gain a better understanding of the number of vulnerable functions for each label, the graph shown in Figure 3.4 was created. This graph highlights the variation in the number of vulnerable functions across different labels, which can have an impact on the final results.

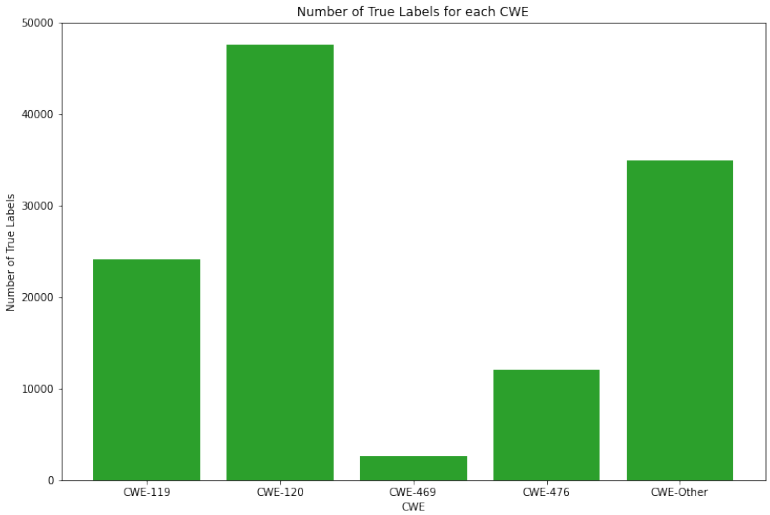


Figure 3.4: Total Vulnerable Functions per Label.

Figure 3.5 shows the number of features and their corresponding number of

vulnerabilities. Each bar represents the number of functions associated with a particular number of vulnerabilities, as detailed in the caption. As observed in the Figure 3.5, there are functions that have been classified with more than one vulnerability, indicating that this work involves a multi-label classification problem.

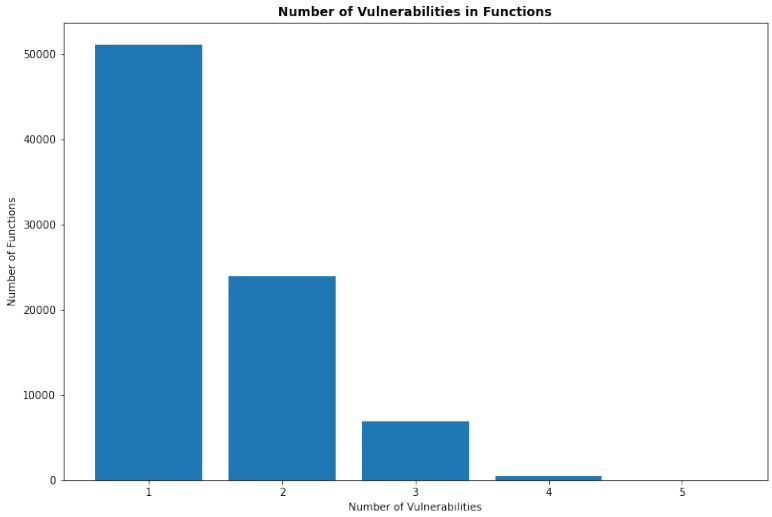


Figure 3.5: Distribution of Functions by Number of Vulnerabilities.

3.3 Preprocessing

In order to obtain accurate representations of the source code, it is necessary to perform source code preprocessing, which involves a series of steps discussed in the following sections. These steps are crucial for simplifying the source code, making it easier for the model to identify patterns that may indicate vulnerabilities.

3.3.1 Data Division

As mentioned previously, the Draper VDISC dataset is organized into five classes representing the five described vulnerabilities. Each class contains functions that may or may not possess the corresponding vulnerabilities. In addition to vulnerability classification, the objective of this work is also to identify the vulnerabilities. To accomplish this, a dataset named "Combined Dataset" was created, which comprises all vulnerabilities and non-vulnerabilities from the Draper VDISC dataset without classifying them. This record will contain a single label indicating the presence or absence of one or more vulnerabilities. In other words, if a feature has multiple vulnerabilities, it will be labeled as "vulnerable" only. Functions

with multiple vulnerabilities may be duplicated within this dataset; however, a check was performed during the dataset creation process to effectively eliminate duplicate functions from the data.

While this preprocessing step is not as critical to the effectiveness of the model as others, these datasets will be used to evaluate our model in binary classification scenarios. Training on these generated datasets allows us to assess which vulnerabilities are more easily detected by the model.

3.3.2 Data Cleaning

The source code provided by the Draper VDISC dataset includes functions written in C/C++, some of which have been sourced from projects on Github, Debian Linux, and other public repositories as already mentioned. In Listing 3.5, an example function from the dataset is shown, retaining the original identification and code structure as developed by the author.

```
1  clear_area(int startx, int starty, int xsize,
       int ysize)
2  {
3    int x;
4
5    TRACE_LOG("Clearing area %d,%d / %d,%d\n",
       startx, starty, xsize, ysize);
6
7    while (ysize > 0)
8    {
9      x = xsize;
10     while (x > 0)
11     {
12       mvaddch(starty + ysize - 2, startx + x
           - 2, ' ');
13       x--;
14     }
15     ysize--;
16   }
17 }
```

Listing 3.5: Example of CWE-469.

To facilitate vulnerability detection, it is necessary to remove any content that does not contribute to the detection process. This includes removing spaces, leaving only one space to separate different types of words, also removed the comments, commas, curly brackets, periods, and ensuring case-insensitivity. This preprocessing step simplifies the functions by eliminating elements that could have

a negative impact on the model and increase processing time. Listing 3.6 demonstrates the application of this preprocessing step to the function shown in Code Snippet 3.5. This process was applied to all functions in the dataset.

```
clear_area(int startx, int starty, int xsize,
     int ysize) int x trace_log("clearing area
     %d,%d / %d,%d\n", startx, starty, xsize,
    ysize) while (ysize > 0) x = xsize while (
    x > 0) mvaddch(starty + ysize - 2, startx
    + x - 2, ' ') x-- ysize--
```

Listing 3.6: Example of CWE-469 after cleaning.

3.3.3 Data Transformation

The process of detecting vulnerabilities in source code involves several stages of data transformation aimed at transforming raw source code into formats suitable for ML analysis. These transformations are critical to enable effective pattern recognition and feature extraction by the models. In this section, we examine the various steps of data transformation, from initial tokenization to the creation of meaningful representations for vulnerability detection.

Lexing: Tokenization

Tokenization is a fundamental step in the process of automatically detecting vulnerabilities in source code. As the first step in our data transformation pipeline, tokenization involves breaking raw source code into smaller units known as tokens. These tokens can include a variety of elements, such as words, characters, subwords, or other meaningful segments, depending on the approach chosen and the specific goals of the analysis. The importance of tokenization stems from its key role in facilitating subsequent ML processes.

Effective integration of tokenization is essential before data is fed into the ML model. This process helps reduce the overall size of the data, making it more manageable and speeding up the subsequent training process. By representing the source code as tokens rather than individual words, tokenization not only speeds up model training, but also plays a critical role in transforming the input data into an appropriate numerical representation. This numeric representation is essential because most of our ML models require numeric input for efficient processing and pattern recognition.

The tangible benefits of tokenization can be seen through an illustrative example. In Figure 3.6, shows an example of the tokenization process applied to the

function. Each token extracted from the source code symbolizes a different element or unit of information from the function. Moreover, this step guarantees uniform sequence lengths, which reduces the risk of data loss and facilitates the subsequent analysis phases. As a result, the tokenized data is a well-prepared input for the subsequent ML models.

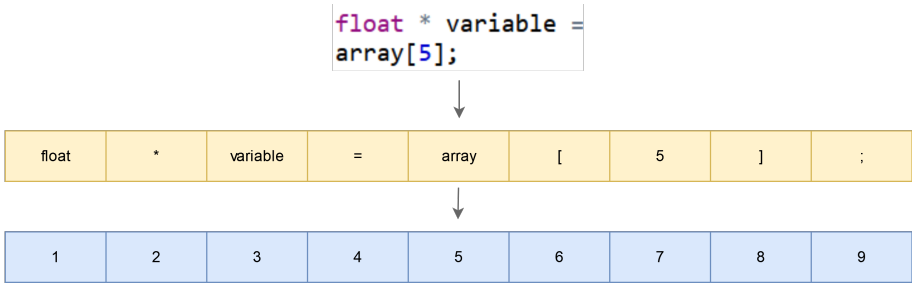


Figure 3.6: Example of the Tokenization Process Applied in this Work.

Bag-of-Words (BoW)

In the field of source code vulnerability detection, it is imperative to use sophisticated techniques, given the limitations of simple statistics and trivial features, as exemplified by the study by Chernis et al. [80]. While basic statistics and trivial features offer computational efficiency, experiments based on these approaches have yielded less promising results. These methods, which focus primarily on basic metrics, lack the depth required to provide the rich contextual information essential for effective source code vulnerability detection. Consequently, this study seeks a more robust methodology to achieve its goals.

One approach widely used in Natural Language Process (NLP), and consequently adopted for this study, is the BoW model. The BoW model serves as a powerful tool for representing textual data within ML algorithms. It simplifies the textual representation of documents by treating each document as a collection or "bag" of words, effectively ignoring nuances of grammatical structure and word order. The core goal of the BoW model is to create a numerical vector that captures the frequency of occurrence of each word within a given set of documents.

The initiation of the BoW process involves the construction of a vocabulary, essentially a compiled list of all unique words found in the documents. Each document is then translated into a vector format, where each vector element corresponds to a word in the vocabulary. The numerical value of a vector element represents the frequency of occurrence of the corresponding word in the document. Importantly, in line with the data preprocessing pipeline, this vocabulary construction and vectorization process occurs concurrently with the function tokenization step, as illustrated in the figure 3.6.

Word Embeddings

Word embedding techniques play a key role in ML and NLP models by providing numerical representations for words or phrases. This transformation enhances the interpretability and processing capacity of ML algorithms by seamlessly transforming numerical data into their preferred input format. Word Embeddings excel at capturing semantic nuances and relationships based on contextual usage, effectively grouping words with similar meanings or contexts into vectors that are tightly clustered in vector space.

In this study, the strategic use of word embedding techniques, particularly the Embedding Layer and GloVe, is of paramount importance, especially within NN-based ML models. This choice facilitates direct performance comparisons between ML models, covering both training and testing speeds.

The Embedding Layer, a type of word embedding, co-trains with the NN model. Unique word encodings are ensured by text preprocessing. During training, the embedding vectors are adapted by the back-propagation algorithm. The embedding layer, located at the beginning of the model, receives BoW input and generates corresponding embedding vectors for subsequent layers, which are set to 300 dimensions.

Conversely, GloVe, an evolution of Word2Vec, integrates global statistical properties with local context-based learning. Unlike the window approach of Word2Vec, GloVe constructs a word occurrence matrix using full text statistics. This study uses a pre-trained GloVe model with 840 billion tokens and a vocabulary of 2.2 million words. These are transformed into 300-dimensional vectors [81]. Embedding techniques enhance the analytical capabilities that are essential for unraveling the intricacies of source code vulnerabilities.

The benefits of using word embeddings are clear. They skillfully transform source code into a format that ML models can efficiently process. These embeddings capture nuanced semantics by recognizing patterns across snippets of code, even when the wording is not identical. They effectively address the challenges of high-dimensional data by compressing the vocabulary into manageable dimensions. In addition, they navigate the complexity of out-of-vocabulary words and easily adapt to new code constructs. In summary, word embeddings enable a nuanced understanding of source code semantics, increasing vulnerability detection accuracy and adaptability to real-world complexities.

3.3.4 Data Reduction

When analyzing the distribution of data in a dataset, it is common to assess whether the dataset is balanced or unbalanced, particularly in classification problems. Unbalanced data refers to an uneven distribution of classes within the

dataset. For instance, in Figure 3.3, we can see that the number of non-vulnerable functions is much higher than the number of vulnerable functions. If we train a classification model without addressing this issue, the model's results may be biased. Additionally, this problem can impact the correlations between features.

Data imbalance is a persistent problem in various domains, especially in the area of data processing for ML models. As a result, numerous resampling techniques have been developed to address this problem, including both undersampling and oversampling methods. In a simple way, oversampling duplicates data from the smaller classes, while in undersampling, data from the larger classes are removed. In contrast to the work of R. Russell et. al. [19], this work uses the undersampling technique to balance the Draper VDISC dataset and improve the data processing time. In this way, some observations from the majority classes were randomly eliminated to match the number of elements in the minority class. Figure 3.7 illustrates the outcome of the undersampling performed on the dataset.

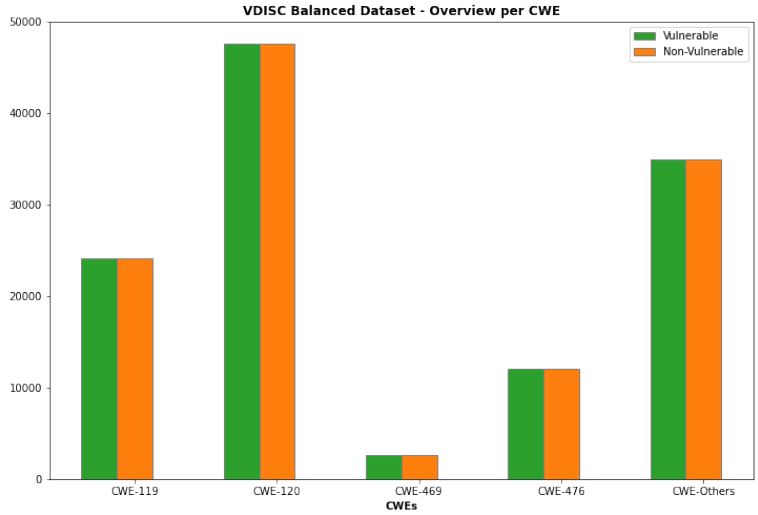


Figure 3.7: Total Vulnerable and Non-Vulnerable Functions per Label after Undersampling.

The same process was applied to the combined dataset, and the result is presented in Figure 3.8. As observed, the number of vulnerable entries decreased from 121 564 to 82 411. In this way, it can be concluded that approximately 32% of the vulnerable entries in the dataset exhibit more than one vulnerability.

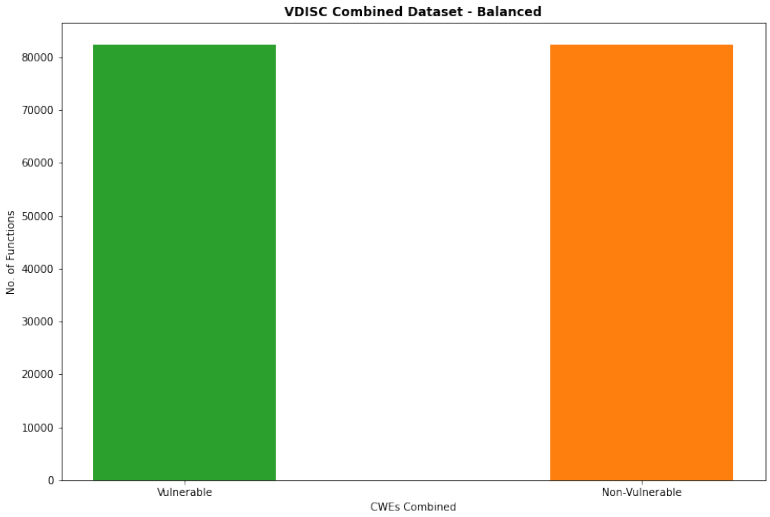


Figure 3.8: Distribution of Combined Dataset Data After Undersamppling.

Chapter 4

ML Applied To Source-Code Verification

This chapter describes the ML techniques used to train the lexed data and provides a comprehensive overview of the selected models. For this purpose, two different phases were implemented. In the first phase, we trained so-called "Generic Algorithms", which are less robust ML algorithms compared to NNs. We then explored three NN: CNN, GRU, and BLSTM. We will present the structures used in each of the models tested, highlighting their strengths in detecting vulnerabilities in C and C++ functions. In addition, we will discuss certain techniques that were attempted but did not provide the desired results, leading to their exclusion.

Throughout the testing process, various metrics were employed to evaluate the performance of the models. The F1-Score and MCC were utilized for the purpose of comparing their effectiveness. Figure 4.1 illustrates the two stages involved in this process and identifies the algorithms used at each stage.

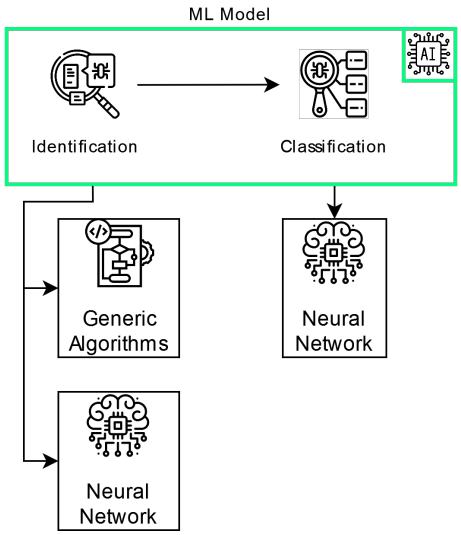


Figure 4.1: Different Stages and the Corresponding Algorithms used in Each.

4.1 Experimental Setup

To effectively accomplish the tasks at hand, a dual machine approach was adopted. Initially, a less robust machine was used for the initial model training phase. However, due to the complex nature of the dataset, this hardware configuration proved inadequate as it reached its limits. Consequently, a transition was made to a more powerful machine better equipped to handle the computational demands of training heavier ML models. This dual-machine strategy played a critical role in optimizing the execution of this work. Each machine brought its own set of specifications and software configurations, allowing a comprehensive response to evolving requirements throughout the development lifecycle. The careful selection of appropriate libraries and software also proved to be a key factor in facilitating the development and execution of the experimental efforts.

Table 4.1: Specifications of Machine 1.

Component	**Specification**
Operating System	Windows 10 Home
CPU	Intel® Core™ i7-7700HQ
GPU	Nvidia GeForce GTX 1050 Ti
RAM	16 GB
CUDA ®	Version 11.5

The first machine (Table 4.1) has a high performance CPU configuration, making it suitable for processing data-intensive tasks. Additionally, the inclusion

of the Nvidia GeForce GTX 1050 Ti graphics card enables hardware acceleration, thereby benefiting ML operations. The 16 GB of RAM proves sufficient for handling certain data processing requirements. However, it's important to note that this machine has limitations when it comes to certain challenges, particularly in scenarios involving training on unbalanced data sets and using more complex models such as NN. As model complexity increases, computational requirements increase accordingly, NN have multiple layers and a large number of interconnected nodes, requiring extensive computing power for training and inference. These models inherently involve complicated mathematical computations, including matrix multiplications and gradient computations, which add to the hardware workload. In cases of unbalanced datasets, where the distribution of classes or instances is skewed, the training process becomes even more complicated. To overcome this challenge, specialized techniques such as weighted loss functions or data augmentation are often used. These techniques result in increased computational requirements, as the model must perform additional computations to appropriately adapt to and learn from the unbalanced data.

Table 4.2: Specifications of Machine 2.

Component	**Specification**
Operating System	Ubuntu 18.04.6 LTS
CPU	Intel® Xeon® Silver 4210
GPU	Nvidia Quadro P4000
RAM	120 GB
CUDA ®	Version 10.1

The second machine (Table 4.2) employed features a more robust configuration. The Intel Xeon Silver 4210 CPU is designed to excel in high-performance applications and possesses advanced virtualization capabilities. Complementing the CPU, the Nvidia Quadro P4000 GPU offers additional processing power for accelerated ML operations. With a substantial 120 GB of RAM, this machine enables effortless manipulation of large datasets and facilitates experiments on a larger scale. Such capabilities prove particularly vital for the employed dataset, which comprises over 1 million of data inputs.

The libraries and softwares employed in this project (Table 4.3) play an important role in the development and implementation of ML techniques for source code vulnerability detection. TensorFlow, in conjunction with Keras, furnishes a high-performance environment for constructing and training ML models based on NN. The Keras-Tuner library is utilized to optimize the hyperparameters of these neural networks. Scikit-learn serves as a widely adopted library for ML and data mining, enabling the training of generic algorithms, hyperparameter tuning,

and providing evaluation metrics for all ML models. Visual Studio Code was selected as the integrated development environment (IDE) for writing, debugging, and executing Python 3.7.5 code.

Table 4.3: Software and Libraries used in the Development Phase.

Software	**Version**
Visual Studio Code	1.79.2

Libraries	**Version**
TensorFlow	2.3.0
Keras	2.9.0
Keras-Tuner	1.3.5
Scikit-Learn	1.0.2
Matplotlib	3.5.3
Pandas	1.3.5
H5py	2.10.0
Numpy	1.18.5

It is worth noting that the specific versions of the aforementioned libraries have been explicitly mentioned to ensure result reproducibility and maintain consistency within the experimental environment. It is recommended to utilize the mentioned versions or later releases to take advantage of potential improvements and bug fixes.

In conclusion, the experimental environment employed in this study provided a conducive setup for detecting vulnerabilities in source code through ML techniques. Machine 2, the associated libraries, and the employed software collaboratively facilitated efficient execution of the experiments, yielding substantial and reliable results.

4.2 Generic Machine Learning Algorithms

After reviewing the available literature, presented in Section 2.4, a set of generic vulnerability detection and classification algorithms were chosen. As the objective of this work includes investigating the optimal approach, additional algorithms were selected to assess their effectiveness. The algorithms chosen for these tests encompassed RF, Lasso, ElasticNET, Naive Bayes, Gradient Boosted Trees, Support Vector Machine (SVM), Logistic Regression, Linear Discriminant Analysis, KNeighborsClassifier, Decision Tree Classifier and Gaussian Naive Bayes.

The initial phase of our workflow involves training a set of algorithms on data transformed using the BoW representation. This BoW representation is

constructed from the text data using the CWE-Combined dataset. To ensure efficiency in terms of resource usage, the algorithms are first trained using default parameter settings. We then select the three best performing algorithms and perform a refinement step using Scikit-Learn's GridSearch library. This fine-tunes the parameters of the algorithms, leading to improved performance results.

To create the BoW representation, we follow these steps, as supported by the accompanying code:

1. **Tokenization:** We use the Tokenizer Module (Listing 4.1) to tokenize the pre-processed C/C++ Source Code, represented by the variable "X" in Listing 4.1. This process involves breaking down the code into individual tokens as shown in Figure 3.6.

2. **Sequence Generation:** Using the `texts_to_sequences` function (Listing 4.1), we convert the tokenized text data into sequences of numerical indices. Each index corresponds to a specific word in our dataset.

3. **Padding:** In the study by R. Russell et al. [19], all sequences were constrained to a maximum size of 500, sequences exceeding this limit were subsequently excluded. In the current work, in order to fully utilize all of the data in the dataset, it was determined that the maximum sequence size would be set equal to the size of the largest sequence defined in the previous step, defined by the "max_len" variable in Listing 4.1. The sequences are then padded using the `pad_sequences` function (Listing 4.1). This step ensures that all sequences have uniform lengths, which is a prerequisite for effective input to ML algorithms.

```
# Tokenizer Module
tokenizer = Tokenizer(oov_token="<OOV>")
tokenizer.fit_on_texts(X)

# Generate Sequences Module
X_seq = tokenizer.texts_to_sequences(X)
max_words = len(tokenizer.word_index) + 1
word_index = tokenizer.word_index

# Pad Sequences Module
max_len= max([len(seq) for seq in X_seq])
X_padded = pad_sequences(X_seq, maxlen=
    max_len)
```

Listing 4.1: BoW Representation Code.

Following the preparation of the BoW sequences, we proceed to train a machine learning model. In this case, the 11 generic algorithms mentioned above were used for later comparison. The training data for the classifier consists of the padded BoW sequences and their corresponding labels. When using the BoW approach, the input data of the algorithms is transformed into a matrix, where the rows correspond to sequences, also known as documents, and the columns represent the number of unique words. These unique words are the total number of different words that appear in a given dataset without considering repetitions or duplicates, which helps to analyze the frequency and presence of each word in different documents. This data representation facilitates the algorithms' ability to recognize patterns and correlations between texts and their respective classifications during training. This is the basis of our vulnerability identification process.

The results of ROC-AUC and F1-Score of these tests are available in the Table 4.4 and 4.5. Based on this metrics results, the algorithms with the best performance were Gradient Boosted Trees, RF and Decision Tree Classifier. In contrast, SVM, Gaussian Naive Bayes, and Logistic Regression showed comparatively weaker results. In Appendix E, specifically in tables E.1, E.3, E.5, E.7, E.9, and E.11, contains the values of the other metrics,

Table 4.4: Results of the Generic Algorithms using Default Parameters for CWE-119, CWE-120 and CWE-469 Vulnerabilities

	CWE-119		CWE-120		CWE-469	
Model	ROC-AUC	F1-Score	ROC-AUC	F1-Score	ROC-AUC	F1-Score
Random Forest	0.6806	0.7089	0.6718	0.6982	0.7183	0.7578
Lasso	0.5640	0.6578	0.5650	0.6590	0.5592	0.6607
ElasticNET	0.5644	0.6581	0.5650	0.6590	0.5600	0.6622
Naives Bayes (NB)	0.6193	0.5464	0.6038	0.522	0.6499	0.5939
Gradient Boosted Tree	**0.7565**	**0.7601**	0.7091	0.7234	0.7025	0.7488
SVM	0.5367	0.5403	0.5342	0.4037	0.4886	0.5436
Logistic Regression	0.6038	0.5385	0.5853	0.4818	0.6189	0.5495
Linear Discriminant Analysis	0.5635	0.6574	0.5649	0.6587	0.5565	0.6572
KNeighborsClassifier	0.5738	0.5831	0.5679	0.5741	0.5751	0.6119
Decision Tree Classifier	0.6097	0.6190	0.5911	0.6020	0.5740	0.6127
Gaussian NB	0.5087	0.0596	0.5003	0.0074	0.4982	0.6771

Table 4.5: Results of the Generic Algorithms using Default Parameters for CWE-476, CWE-Other and CWE-Combined Vulnerabilities

	CWE-476		CWE-Other		CWE-Combined	
Model	ROC-AUC	F1-Score	ROC-AUC	F1-Score	ROC-AUC	F1-Score
Random Forest	0.6225	0.6295	0.6615	0.6896	0.6727	0.6516
Lasso	0.5007	0.6065	0.5564	0.6503	0.6483	0.5610
ElasticNET	0.5601	0.6622	0.5563	0.6501	0.6483	0.5610
Naives Bayes (NB)	0.5948	0.5093	0.5976	0.5107	0.5231	0.6028
Gradient Boosted Tree	0.6289	0.6402	0.6811	0.6975	0.6857	0.6714
SVM	0.5074	0.5454	0.5311	0.5174	0.5553	0.4974
Logistic Regression	0.5674	0.4412	0.5811	0.4589	0.4915	0.5043
Linear Discriminant Analysis	0.5024	0.6077	0.5556	0.6494	0.6480	0.5608
KNeighborsClassifier	0.5616	0.5965	0.5580	0.5738	0.5564	0.5574
Decision Tree Classifier	0.5625	0.5689	0.5882	0.6059	0.5788	0.5723
Gaussian NB	0.5033	0.0427	0.4997	0.0017	0.6627	0.4998

After obtaining the results presented in the Tables 4.4 and 4.5, the bests generics algorithms were tuned to extract the best performance from them. This tuning was done through several iterations, using a reduced range of values to reduce the time spent in the process. Thus, if the result was at the limit of the range, another iteration was performed until the result fit within the range of values determined in the tuning. This was done until all hyperparameters were within the values defined before starting the tuning. Table 4.6 and 4.7 summary the results of the tuning and the hyperparameters found during the process. Tables E.2, E.4, E.6, E.8, E.10, E.12 show the values of the other metrics and the hyperparameters found.

Table 4.6: Results of the Generic Algorithms after Parameter Tuning for CWE-119, CWE-120 and CWE-469 Vulnerabilities

	CWE-119			CWE-120			CWE-469		
Model	ROC-AUC	F1-Score	MCC	ROC-AUC	F1-Score	MCC	ROC-AUC	F1-Score	MCC
Random Forest	0.7225	0.7352	0.4468	0.7102	0.7235	0.4223	0.7505	0.7616	0.5021
Decision Tree Classifier	0.6156	0.6156	0.2313	0.6044	0.6002	0.2089	0.6416	0.6519	0.2833

Table 4.7: Results of the Generic Algorithms after Parameter Tuning for CWE-476, CWE-Other and CWE-Combined Vulnerabilities.

	CWE-476			CWE-Other			CWE-Combined		
Model	ROC-AUC	F1-Score	MCC	ROC-AUC	F1-Score	MCC	ROC-AUC	F1-Score	MCC
Random Forest	**0.7537**	**0.7682**	**0.5120**	0.6930	0.7040	0.3871	0.6910	0.7044	0.3837
Decision Tree Classifier	0.6884	0.7545	0.4535	0.5910	0.5832	0.1822	0.6152	0.6356	0.2320

Subsequently, after obtaining the initial results, these algorithms underwent a second round of training. This time, the objective was to recognize and classify

vulnerabilities where the complete dataset was used. Following an in-depth investigation, the algorithms presented in Tables 4.4 and 4.5 were employed in combination with two techniques: Binary Relevance and Classifier Chain. These techniques are well-known for their effectiveness in handling situations where multiple labels need to be assigned to data points. The goal was to accurately classify vulnerabilities using these techniques.

In the Binary Relevance approach, each label is treated as an independent binary classification problem. This means that for each label, a separate classifier is trained to predict whether that particular label should be assigned to a given input data point. Essentially, the problem is decomposed into multiple binary classification tasks, one for each label. This approach doesn't consider potential relationships between different labels and treats them in isolation. In Table 4.8, shows the results of using this technique for multi-label classification.

Table 4.8: Results of the Generic Algorithms using Binary Relevance

Model	Accuracy	Precision	Recall	F1-Score (Micro)	F1-Score (Macro)	F1-Score (Weighted)
Random Forest	**0.5021**	**0.7516**	0.0302	0.0575	0.0326	0.0551
Lasso	0.4978	0.3545	0.0062	0.0128	0.0069	0.0121
ElasticNET	0.4978	0.3294	0.0060	0.0117	0.0067	0.0117
Naive Bayes	0.3806	0.3100	0.4157	0.3117	0.2383	0.3311
Gradient Boosted Trees	0.5012	0.5624	0.1139	0.1892	0.1225	0.1790
SVM	0.1561	0.2340	0.4472	0.2793	0.2229	0.3001
Logistic Regression	0.4843	0.3237	0.0478	0.0844	0.0553	0.0820
Linear Discriminant Analysis	0.4948	0.2627	0.0094	0.0182	0.0147	0.0181
KNeighborsClassifier	0.4311	0.2978	0.1280	0.1815	0.1063	0.1688
Decision Tree Classifier	0.2677	0.2699	0.3348	**0.2903**	0.2239	0.2980
Gaussian NB	0.0010	0.2114	**0.9965**	0.2576	**0.2445**	**0.3420**

The Classifier Chain approach builds a sequence or chain of classifiers. The first classifier is trained on the input data, as in traditional classification. However, the subsequent classifiers are trained sequentially. For each successive classifier, it takes into account both the input data and the outputs (predictions) of the previous classifiers in the chain. This chaining of classifiers allows to capture potential label dependencies. For example, if a classifier earlier in the chain predicts certain labels, these predictions can be used as features for the classifiers later in the chain. In Table 4.9, shows the results of using Classifier Chain for multi-label classification.

Table 4.9: Results of the Generic Algorithms using Classifier Chain

Model	Accuracy	Precision	Recall	F1-Score (Micro)	F1-Score (Macro)	F1-Score (Weighted)
Random Forest	**0.5024**	**0.7463**	0.0170	0.0331	0.0213	0.0327
Lasso	0.4978	0.3545	0.0062	0.0123	0.0069	0.0121
ElasticNET	0.4977	0.3284	0.0060	0.0117	0.0067	0.0117
Naive Bayes	0.3782	0.2892	0.4254	0.2823	0.2488	0.3322
Gradient Boosted Trees	0.4991	0.5649	0.1189	0.1942	0.1241	0.1815
SVM	0.1729	0.2379	0.4554	0.2833	0.2235	0.3076
Logistic Regression	0.4817	0.3250	0.0503	0.0821	0.0578	0.0812
Linear Discriminant Analysis	0.4968	0.2585	0.0105	0.0193	0.0131	0.0189
KNeighborsClassifier	0.4296	0.2935	0.1255	0.1819	0.1036	0.1658
Decision Tree Classifier	0.2703	0.2681	0.3298	**0.2852**	0.2214	0.2972
Gaussian NB	0.0007	0.2136	**0.9940**	0.2564	**0.2460**	**0.3387**

However, the results were unsatisfactory. The best-performing model achieved only around 38% accuracy and a maximum F1-Score of approximately 30%. Regrettably, these values indicate that the algorithms struggled to consistently pinpoint vulnerabilities within the code. Instead of providing reliable predictions, their outputs seemed almost random. In the Tables 4.8 and 4.9 show the results of the multi-label classification. These results, which are lower than expected, could potentially be attributed to the complexity of the problem addressed in this work. Some vulnerabilities can be intricate and challenging to identify, requiring a deep understanding of the intricacies of the language and secure programming principles. Another contributing factor may be the lack of context, because that vulnerability detection often relies on the code's context, less robust models, like the ones utilized in this study, could face challenges in accurately identifying vulnerabilities.

4.3 Proposed Neural Networks Models

The complexity and size of the dataset make it difficult to uncover the hidden relationships that lead to specific vulnerabilities, which in turn hampers the effectiveness of traditional algorithms. To address this problem, we conducted experiments using a variety of NN models known for their ability to detect complex relationships. NNs have a remarkable ability to capture intricate patterns and dependencies within the data, thereby enhancing our ability to uncover complex vulnerabilities. We have focused on three specific NN architectures: CNN, GRU, and BLSTM. These architectures have been strategically chosen for their application in identifying vulnerabilities in C/C++ source code. Each architecture has unique strengths and suitability for vulnerability detection. CNNs excel at capturing local dependencies and extracting relevant features from fixed-dimension input. Conversely, GRUs and BLSTMs excel at modeling sequential dependencies inherent in source code. By harnessing the potential of these NNs, our goal is to redefine the

limits of vulnerability detection and thereby make a significant contribution to the field of software security.

For our experiments, we chose to use Second Machine because of its superior specifications compared to First Machine. This ensured that we could use more complicated models without worrying about memory leaks. The experimental process for testing NNs closely mirrored the methodology used to evaluate Generic Algorithms, albeit with certain nuances in the initial steps.

Following the analysis of the State of the Art presented in Chapter 2, a set of configurations for the NN were selected and evaluated based on their performance using two metrics: F1-Score and MCC. After determining the better NN configuration, we used the Keras-Tuner library for hyperparameter optimization. This ingenious library uses a random selection approach to pair different sets of hyperparameter values, facilitating the training and evaluation of machine learning models. The algorithm follows a procedure of randomly selecting a predefined set of parameter values. It then trains the NN with these values and measures its performance against specified metrics such as precision, accuracy, and loss. This work culminated in defining precise value ranges for hyperparameters, each paired with its corresponding value.

Finally, after defining the hyperparameters, we moved on to retraining the NNs. Initially, we focused on the detection of a single vulnerability, framing it as a binary classification task. Then, as shown in Figure 1.3, the ML model module was initiated, which involved training these NNs to detect and classify multiple vulnerabilities. This transformation turned the module into a multi-label classification task.

The following sections present the three NN used for both binary and multi-label classification, along with their configurations, hyperparameters, and the results obtained.

4.3.1 Convolutional Neural Networks Architecture

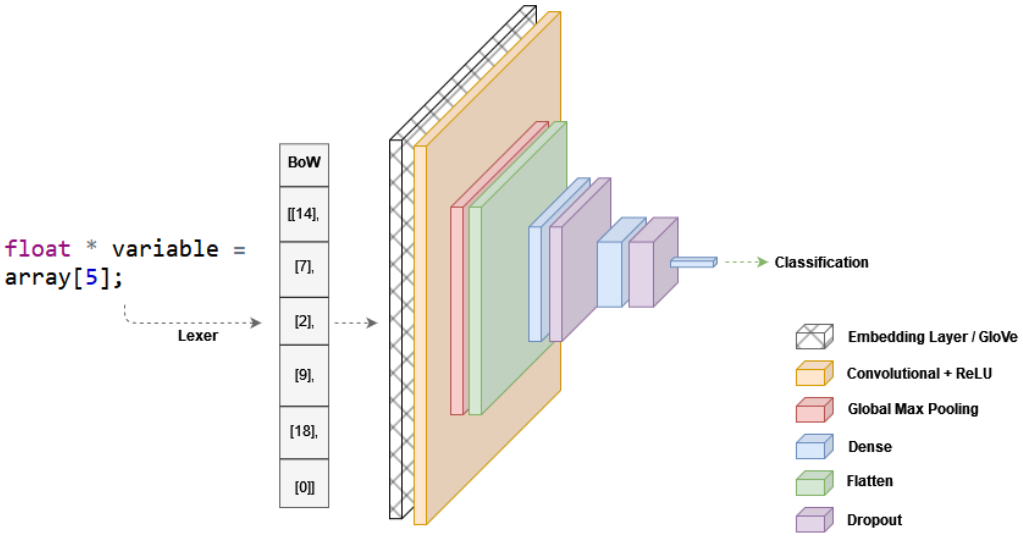


Figure 4.2: Proposed Convolutional Neural Network Architecture.

The work by Russell et. al. [19] has demonstrated the effectiveness of CNNs as powerful tools for automatically identifying vulnerabilities in C/C++ source code. In their research, CNN models have surpassed RNN-based models, both in terms of standalone classification performance and their ability to generate informative features.

This section introduces the proposed CNN architecture specifically tailored for the detection and classification of vulnerabilities in C/C++ source code. Multiple configurations were tested for vulnerability classification until the architecture depicted in Figure 4.2 was derived, exhibiting the most favorable performance.

The initial configuration proposed by R. Russell et al. [19] served as the starting point for our exploration into Convolutional Neural Networks (CNNs) for vulnerability detection in C/C++ source code. This configuration closely resembled that of R. Russell et al [19]., consisting of an Embedding Layer, a Convolutional Layer, a Sequence Max Pooling Layer, a Flatten Layer, and Dense Layers. However, in our pursuit to optimize and tailor the architecture for enhanced vulnerability classification, several adjustments were made to the original configuration. We began by evaluating R. Russell et al. [19]'s configuration, which included an Embedding Layer for semantic representation, a Convolutional Layer for local feature extraction, a Sequence Max Pooling Layer for dimensionality reduction, a Flatten Layer to prepare for subsequent layers, and Dense Layers for intricate pattern learning. While this configuration exhibited promising results, we recognized the potential for further improvements. To enhance the architecture's performance, we iteratively modified various aspects of the Convolutional Layer. Filters were altered,

added, or removed to capture more intricate patterns associated with vulnerabilities in C/C++ code. This iterative process allowed us to identify filters that specifically highlighted crucial patterns and structures related to vulnerabilities. Furthermore, adjustments were made to the layer composition following the Convolutional Layer. The Sequence Max Pooling Layer was replaced with a Global Max Pooling Layer to capture the most significant values across filters, enhancing the model's ability to recognize essential features regardless of their positions. The Flatten Layer remained to facilitate the connection to subsequent layers, but the arrangement of Dense Layers was refined to enable better learning of complex patterns. These changes were necessary to achieve an architecture that was finely tuned to the task of vulnerability classification in C/C++ code.

By refining the Convolutional Layer's filters, optimizing the pooling strategy, and adjusting the arrangement of subsequent layers, we arrived at the configuration depicted in Figure 4.2. This configuration exhibited superior performance compared to the initial one proposed by R. Russell et al. [19] , as it was designed to capture more nuanced vulnerabilities and their associated features.

The CNN model comprises several layers designed to fulfill specific functions in vulnerability detection. The initial layer is an embedding layer responsible for transforming the feature tokens into dense vectors of fixed dimensions. This layer captures the semantic relationships among the tokens and enables the model to acquire distributed representations of the keywords pertinent to vulnerability detection.

Subsequently, we employ a one-dimensional convolution layer (Conv1D) to extract local features from the token sequences. The 1D convolution analyzes the token sequences using a filter of predetermined size, thereby identifying crucial patterns and structures associated with specific vulnerabilities. By employing multiple filters of varying sizes, the model can capture diverse levels of abstraction and contextual information.

Following the Convolution layer, we incorporate a global maximum pooling layer (GlobalMaxPooling1D) to reduce the dimensionality of the convolution layer's output. This operation retains the most significant values in each filter, enabling the model to capture essential features within each source function, irrespective of their specific positions.

The Flatten layer converts the output of the pooling layer into a one-dimensional vector, making it easier to connect to subsequent fully connected layers. These dense layers are responsible for learning more complex patterns and performing the final classification of functions as either true (containing vulnerabilities) or false (not containing vulnerabilities).

To address overfitting concerns, we introduce dropout layers, which randomly deactivate some units during training. This reduces interdependencies between

units and improves the generalizability of the model. Appendix B provide a summary of the CNN model utilizing both the GloVe and Embedding Layer approaches and the results can be found on Appendix F.

The same architecture was subsequently employed for the classification of individual vulnerabilities. This implementation incorporates two word embedding methods that are GloVe and Embedding Layer.

Tables B.1, B.2, B.3, B.4, B.5, and B.6 show the results of the binary classification, while Table B.7 shows the results of the multi-label classification.

4.3.2 Gated Recurrent Unit Architecture

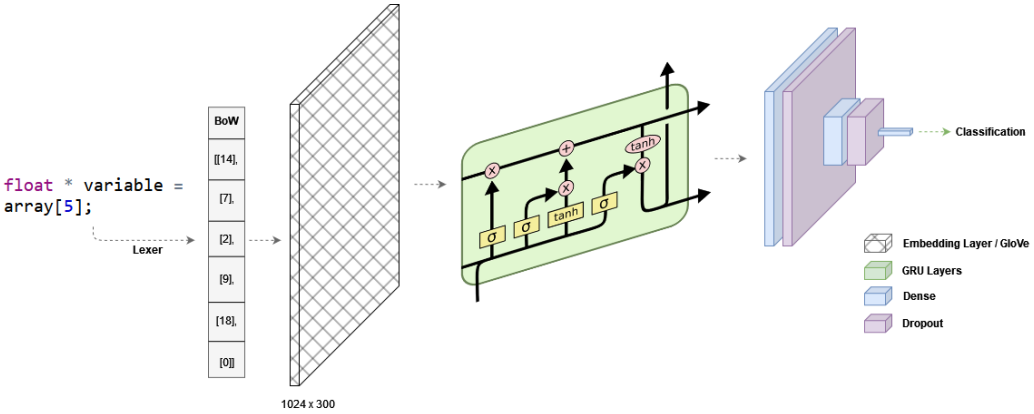


Figure 4.3: Proposed Gated Recurrent Unit Architecture.

In their study, Z. Li et al. [64] demonstrated the potential of LSTMs in effectively identifying and classifying vulnerabilities in source code, that have been transformed into code gadgets. Given the success of these neural networks in previous research, they should be leveraged for the purposes of this work to ensure effective detection and classification of vulnerabilities in lexed code. Consequently, the performance of the GRU neural network will be investigated. GRU is a variation of LSTM that offers a simplified and faster processing alternative.

Despite the limited number of studies using GRUs to detect vulnerabilities in C/C++ code, several configurations were tested. Initially, a basic architecture was used that included the Embedding layer, the GRU layer, and the dense layers. Subsequently, the addition of additional layers was found to cause overfitting in the model. In this way, it was decided to use the initial configuration shown in Figure 4.3, which gave the best performance.

The initial layer is an embedding layer that performs the mapping of input tokens to a continuous vector space with a dimension of 300. The primary purpose of this layer is to semantically represent the tokens, capturing their semantic relationships and providing a dense representation of the data.

Following the embedding layer, a Spatial Dropout Layer (SpatialDropout1D) is employed. This layer randomly deactivates activations during training, aiding in the prevention of overfitting. The spatial dropout specifically operates on a particular dimension of the input, thereby promoting regularization of the model and enhancing its ability to generalize.

The third and fourth layers consist of GRU layers. GRUs are recurrent memory cells capable of capturing long-term dependency information within data sequences. The first GRU layer propagates sequences to the subsequent layer, enabling the model to learn high-level representations of the token sequences. The second GRU layer solely returns the final state of the sequence, capturing more abstract and contextual information from the token sequences.

Following the GRU layers, a series of dense layers is employed. The initial dense layers introduce nonlinearity into the model, enabling it to learn more intricate representations and extract features relevant to vulnerability detection. Additionally, dropout layers are included, each with a dropout rate of 0.1. These layers aid in mitigating overfitting by randomly dropping out some activations during training, thereby enhancing the model's ability to generalize. The final layer is a dense layer responsible for the binary classification of vulnerabilities. It produces an output ranging from 0 to 1. Values approaching 0 indicate the absence of vulnerabilities, while values approaching 1 indicate the presence of vulnerabilities.

This identical architecture was then extended to the classification of individual vulnerabilities. Similarly to the previous implementation, this approach incorporates two word embedding methods, namely GloVe and Embedding Layer. The results of this model are presented in Appendix F.

4.3.3 Bidirectional Long Short-Term Memory Architecture

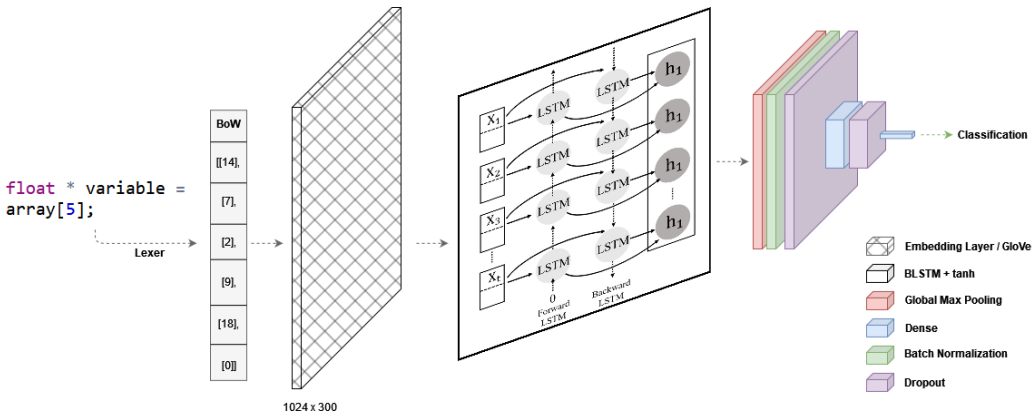


Figure 4.4: Proposed Bidirectional Long Short-Term Memory Architecture Architecture.

In recent years, BLSTM models have gained prominence in various natural language processing tasks, including text classification. BLSTM, a variant of RNNs, have shown promising results in capturing sequential dependencies and long-term dependencies in text data. BLSTM is a recurrent neural network architecture that combines two layers of LSTM: one that processes the input sequence from left to right (forward LSTM) and one that processes the sequence from right to left (backward LSTM). This combination allows the network to capture contextual information from both previous and subsequent sequences to the current position. The work by D. Zou et. al. [67] introduces a BLSTM-based architecture for the detection and classification of vulnerabilities in source code. In this work, the BLSTM performed very well, even outperforming the classic LSTM.

The advantages of using a BLSTM is its ability to capture long-range dependencies in sequential data. In traditional recurrent neural networks, such as LSTM or GRU, information flows only in one direction (from past to future or vice versa), which can limit the ability to model complex relationships. BLSTM overcomes this limitation by processing the input sequence in both directions, capturing information from both the past and the future [67].

Similar to the testing of the two neural networks mentioned above, the literature was consulted and analyzed, and based on this analysis, several different architectural configurations were tested. Figure 4.4 shows the architecture that performed best in the vulnerability classification tests and will be used for implementation.

The first layer is an embedding layer that maps the tokens of functions written in C/C++ to dense vectors of fixed size. This layer captures the semantic relationships between the tokens and represents them in a continuous vector space. Next, we have a bidirectional LSTM layer that processes the sequence of tokens in both directions. This layer captures contextual information from both the past and future of each token, allowing for a more complete understanding of the source code.

After the LSTM layer, we have the GlobalMaxPool1D layer, which extracts the maximum value along the time dimension of the sequences generated by the previous layer. This pooling operation highlights the most relevant features and helps to reduce the dimensionality of the data. To normalize the output of the pooling layer, the Batch Normalization layer is used. This layer adjusts the mean and standard deviation of the data, making it more suitable for training and improving the model's ability to generalize.

Next we have the Dropout and Dense layers, Dropout is used to combat overfitting. During training, a percentage of neurons are randomly deactivated to prevent the model from becoming too dependent on specific neurons. While the Dense layer performs a non-linear transformation on the output of the previous layer,

capturing relevant patterns in the data. The final dense layer produces a classification probability estimate for the presence of vulnerabilities in the source code.

This same architecture was used to classify a single vulnerability. Similar to the previous models, the GloVe and the Embedding Layer were used for the implementation. In the Appendix D provide a summary of the neural networks utilizing both the GloVe and Embedding Layer approaches and Appendix F contains the results.

Chapter 5

Results and Discussion

In this chapter, we will present and examine the results achieved by the Generic Algorithms and NNs discussed in the previous chapter for detecting and classifying vulnerabilities in functions written in C and C++. These results will include crucial metrics and hyperparameters that emerged during the tuning process for each approach. Since the primary focus of this work is to vulnerability identification and classification, we will comprehensively address both aspects.This includes discussing results related to binary classification and multi-label classification, where only NN were used for multi-label classification.

As previously mentioned, the F1-Score and MCC were used to directly compare the different models. In the case of evaluating the multi-label classification models, the letter "W" indicates that the metric was weighted. This implies that these metrics were calculated based on the number of vulnerabilities in each class, rather than as an average over the number of classes. Additionally, ROC-AUC curves are presented to assess the model's classification performance.

5.1 Binary Classification

In the binary classification task, the objective of the performed tests was to determine whether a specific vulnerability is present or absent in the source code dataset. As previously mentioned, separate datasets were utilized for conducting the binary classification of individual vulnerabilities. All classes were grouped into

two categories: "true", indicating the presence of the respective vulnerability in the functions, and "false", indicating its absence.

The features used to train the machine learning algorithms were the tokens obtained from the preprocessing of the functions, as already mentioned. Additionally, in the case of neural networks, GloVe was employed to conduct the tests.

5.1.1 Generic Algorithms

The Generic Algorithms, listed in Tables 4.4 and 4.5, were trained to evaluate their performance in identifying a single vulnerability (binary classification) in the source code. Initially, these algorithms were trained using default parameters for preliminary testing. Subsequently, parameter tuning was applied to the best performing algorithms to unlock their full potential.

According to the results presented in Tables 4.5 and 4.4, the algorithms that achieved the best results were Gradient Boosted Tree (GBT), Random Forest (RF) and Decision Tree Classifier. This outcome can be attributed to the nature of the training data and the construction of these algorithms. They are more robust and less prone to overfitting, providing an advantage in this binary classification for vulnerability detection. Additionally, they possess a better ability to handle non-linearly separable data. As the dataset may contain examples of vulnerabilities and non-vulnerabilities that cannot be linearly separated, decision trees can effectively learn non-linear relationships between features and classes, adapting to various scales and types of features. Moreover, these algorithms employ an ensemble approach, combining the predictions of multiple decision trees. This leads to more stable and generally improved predictions on unseen data. The training process in successive steps, focusing on previous errors, contributes to better fits to the training data, leading to superior overall performance. Furthermore, these algorithms demonstrate a propensity to adjust well to complex relationships between features and classes, as evident in the data used in this study.

Conversely, the algorithms that produced the worst results, as indicated by ROC-AUC and F1-Score, were Support Vector Machine (SVM), Gaussian Naive Bayes, and Logistic Regression. The performance of SVM was hampered by its sensitivity to the scale of the features. If the features are not properly normalized or scaled, the algorithm's performance can be adversely affected. In addition, training SVM on very large datasets can become computationally expensive, requiring significant computational resources. In addition, SVM is best suited for linearly separable datasets. If the dataset contains a combination of linearly and non-linearly separable examples, its performance may suffer. Gaussian Naive Bayes performed poorly because it assumes independence between features, which is not true for the data used in this work. This assumption can lead to an

underestimation of the correlations between the features and affect the performance of the algorithm. Similarly, Logistic Regression struggled because it cannot effectively capture the complex relationships present in the training data. Given the high complexity of the data used in this study, this limitation may have been a significant factor in its poor performance.

Figure 5.1 illustrates the comparison between the different algorithms concerning their F1-Score and ROC-AUC. GBT excelled at detecting CWE-119 vulnerabilities, achieving a high F1-Score (0.7601) and ROC-AUC (0.7565). This performance can be attributed to the overlapping patterns found in buffer overflow vulnerabilities, which GBT is adept at detecting due to its ability to identify subtle data characteristics that indicate these occurrences.

Similarly, GBT demonstrated strong performance in detecting CWE-120 vulnerabilities, achieving an F1-Score of 0.7234 and a ROC-AUC of 0.7091. As these vulnerabilities also contain intricate patterns, GBT likely captured the nuances necessary for accurate classifications.

When detecting CWE-469 vulnerabilities, RF achieved an F1-Score of 0.7578 and a ROC-AUC of 0.7183. Vulnerabilities of this type often have distinctive features that RF effectively captures. Its ability to handle non-linear features may have contributed to its favorable performance in this scenario.

Detecting the CWE-476 vulnerability was a challenge for all algorithms, with ElasticNET achieving an F1-Score of 0.6622 and GBT achieving an ROC-AUC of 0.6289. Null pointer dereference vulnerabilities can have less obvious patterns in the data, making it difficult for the models to consistently identify such cases.

Vulnerabilities classified as CWE-Others were also challenging, but GBT stood out with an F1-Score of 0.6975 and a ROC-AUC of 0.6811. These vulnerabilities may include diverse and less well-defined patterns, where GBT demonstrated an ability to generalize detection.

Combined detection of all vulnerabilities (CWE-Combined) was a significant challenge due to the number of patterns that needed to be identified. The F1-Score achieved by GBT (0.6857) suggests a moderate ability to discriminate between classes. The high ROC-AUC score (0.6714) indicates the model's ability to accurately classify positive and negative examples. A possible explanation for GBT's strong performance in this scenario is its ability to capture complex relationships between the different vulnerability categories.

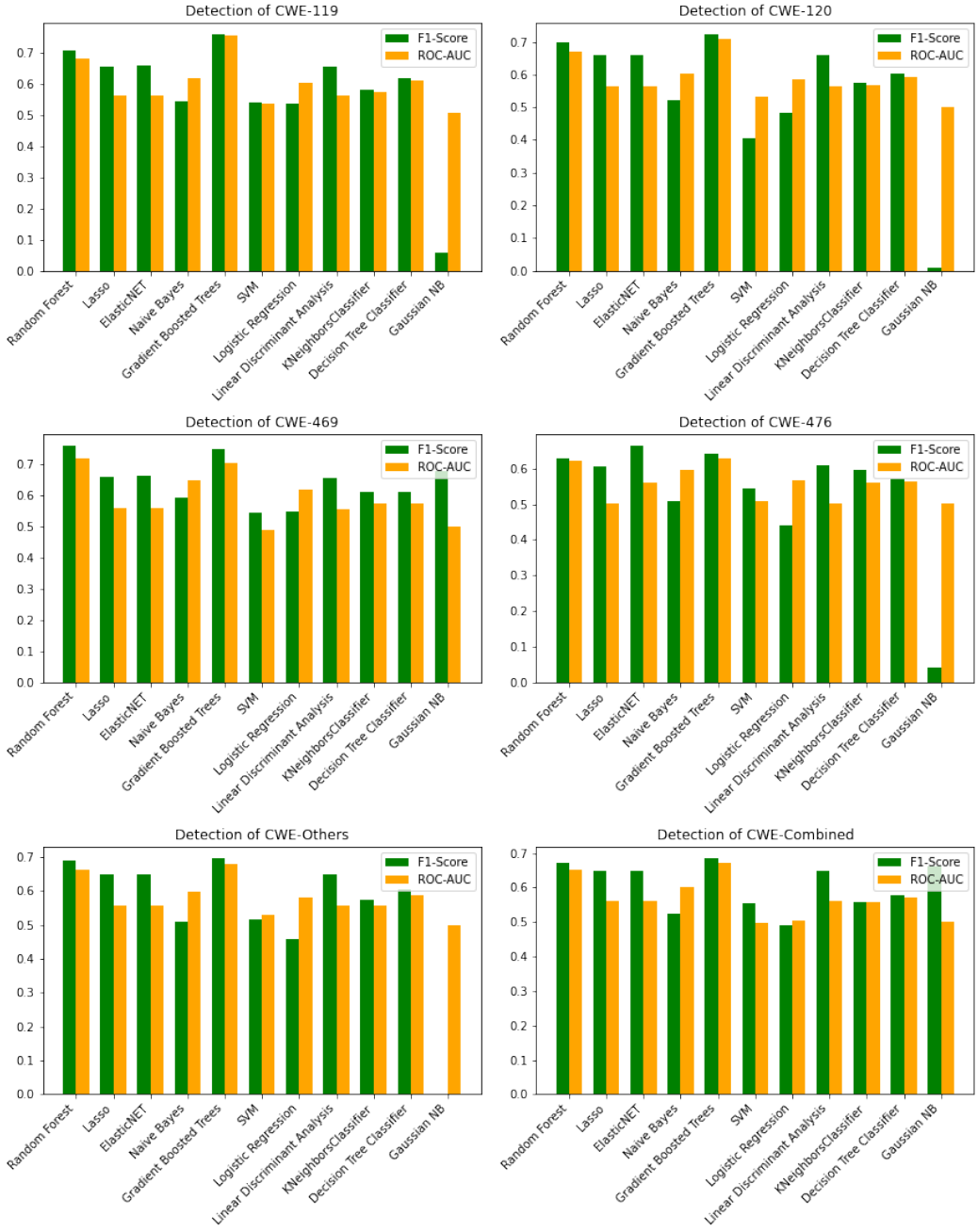


Figure 5.1: Comparison Between the Different Generic Algorithms Used for Binary Classification.

The results obtained suggest that different ML algorithms have different performance in detecting specific vulnerabilities in C/C++ source code. GBT generally excelled in categories such as CWE-119 and CWE-Others, where complex patterns and nuances were more prevalent. RF successfully detected the CWE-469 vulnerability, taking advantage of its ability to handle non-linear features. Conversely, the categories with the worst results, such as CWE-476 and CWE-Combined, may

contain less obvious patterns, thus affecting model performance. These results underscore the importance of considering pattern diversity when selecting and fine-tuning ML algorithms for source code vulnerability detection.

Appendix E presents the results of the generic algorithms. Tables E.1, E.3, E.5, E.7, E.9 and E.11 present the results of the other metrics besides F1-Score and ROC-AUC presented in Figure 5.1.

Tunning

After obtaining the results shown in Figure 5.1, the two most effective generic algorithms were tuned to maximize their performance. Despite GBT's superior performance on most vulnerabilities, a decision was made not to tune it due to its increased robustness compared to other algorithms. As a result, such tuning requires a significant amount of processing power and time to execute, so this process was performed on RF and Decision Tree Classifier. The tuning process involved multiple iterations, using a reduced range of values to reduce the time spent in the process. Thus, if the result was at the limit of the range, another iteration was performed until the result fit within the range of values determined in the tuning. This was done until all hyperparameters were within the values defined before starting the tuning.

Table E.2, E.4, E.6, E.8, E.10 and E.12 shows the results of the tuning and the hyperparameters found during the process. Figure 5.2 presents the comparison between the different algorithms concerning their F1-Score, ROC-AUC and MCC.

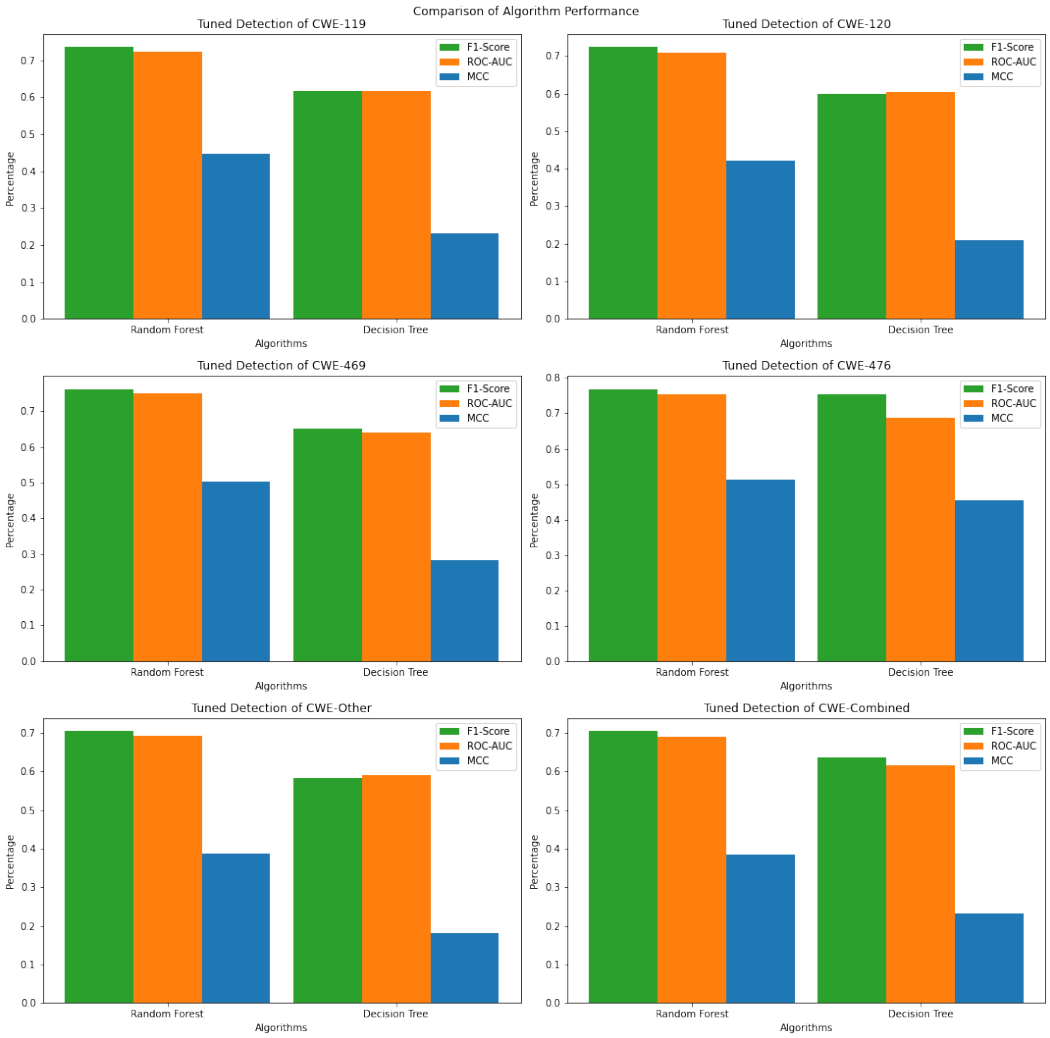


Figure 5.2: Comparison Between the Best Different Generic Algorithms Tuned Used for Binary Classification.

When analyzing the CWE-119 (Buffer Overflow) vulnerability, the algorithms showed good results. Random Forest achieved an accuracy of 72.24%, surpassing Decision Tree's performance of 61.56%. This indicates that Random Forest was more effective at classifying instances. This conclusion is supported by metrics such as precision, recall, F1-Score and MCC where Random Forest outperformed Decision Tree. In particular, Random Forest excelled at identifying true positives, with a recall of 77.03%, demonstrating its ability to detect real vulnerabilities.In addition, the F1-Score of 73.52% and MCC of 44.68% achieved by the RF Model were significantly better than the Decision Tree Model's F1-Score of 61.56% and MCC of 23.13%, respectively. The optimized hyperparameters included 768 estimators, a maximum depth of 64, and a minimum leaf of 1.

Similar patterns were observed for CWE-120 (Buffer Copy without Size Checking) vulnerabilities. Random Forest achieved an accuracy of 71.03%, surpassing Decision Tree's 60.44%. Metrics such as precision, recall, F1-Score and MCC also favored Random Forest. Once again, Random Forest demonstrated its ability to identify true positives, achieving a recall of 75.65%. Its F1-Score (72.35%) and MCC (42.23%) was significantly better than that of Decision Tree (60.02% and 20.89%). The optimized hyperparameters included 128 estimators, a maximum depth of 2048, and a minimum leaf of 1.

In the CWE-469 category (use of pointer subtraction), Random Forest continued to outperform Decision Tree. Random Forest achieved an accuracy of 75.11% compared to 64.12% for Decision Tree. Metrics such as precision, recall, F1-Score and MCC again favored Random Forest. With a recall of 77.96%, Random Forest demonstrated its effectiveness in correctly identifying vulnerabilities. In addition, the F1-Score of 76.16%) and MCC of 50.21% achieved by the RF Model were better than the Decision Tree F1-Score of 65.19% and MCC of 28.33%. The optimized hyperparameters included 32 estimators, a maximum depth of 12288, and a minimum leaf of 1.

In the case of the CWE-476 (NULL Pointer Dereference) vulnerability, there was a noticeable difference in the results of the algorithms. Random Forest achieved 75.33% accuracy and 82.18% recall, while Decision Tree achieved 68.70% accuracy and an impressive 96.76% recall. However, F1-Score and MCC favored Random Forest. The F1-Score of Random Forest (76.82%) and MCC (0.5120%) was higher than that of Decision Tree (75.45% and 45.35%). The optimized hyperparameters included 128 estimators, a maximum depth of 2048, and a minimum leaf of 1.

For CWE-Other Vulnerabilities, Random Forest again outperformed Decision Tree. Random Forest achieved 69.28% accuracy and 73.26% recall, while Decision Tree achieved 59.11% accuracy and 57.40% recall. The F1-Score and MCC of Random Forest (70.40% and 0.3871%) exceeded that of Decision Tree (58.32% and 18.22%). The optimized hyperparameters included 2048 estimators, a maximum depth of 32, and a minimum leaf of 1.

In the CWE-Combined category, the observed metrics followed similar trends. Random Forest achieved an accuracy of 69.08% and a recall of 73.97%, while Decision Tree achieved an accuracy of 61.50% and a recall of 67.41%. Random Forest's F1-Score (70.44%) and MCC (38.37%) also outperformed Decision Tree's (63.56% and 23.20%).

Random Forest's good performance in the CWE-476 category can be attributed to the low frequency and distinct nature of the instances within this category, which allowed the algorithms to effectively identify them. However, a comprehensive evaluation of all metrics is essential for an accurate analysis of the model's performance.

5.1.2 CNN Results

In evaluating the effectiveness of the CNN model for vulnerability detection, the selected algorithms were used to train and refine features, as described previously. This approach allowed for a comprehensive exploration of the model's ability to identify specific vulnerabilities within the source code. The process began with the use of default parameters to perform initial assessments, followed by careful parameter tuning to optimize the most promising parameters. This iterative process was designed to maximize the CNN model's performance potential and demonstrate its ability to detect vulnerabilities. Figure 5.3 shows the comparison between these approaches using F1-Score, while Figure 5.4 shows the comparison of the same approaches using MCC.

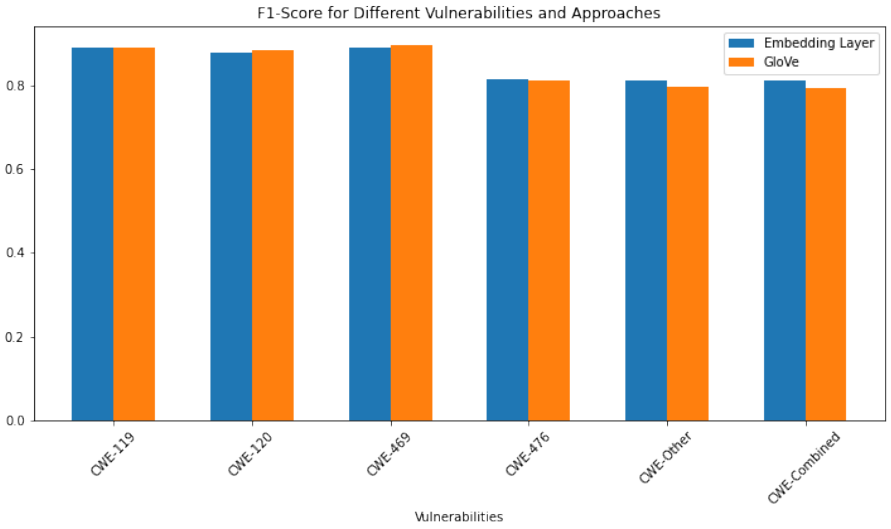


Figure 5.3: Comparison between F1-Score of the GRU Model for Embedding Layer and GloVe Approaches identifying different types of vulnerabilities.

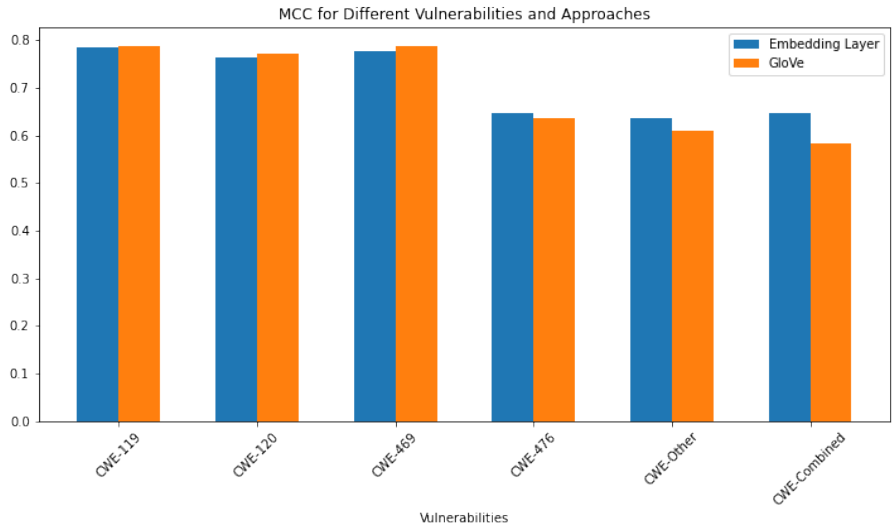


Figure 5.4: Comparison between MCC of the GRU Model for Embedding Layer and GloVe Approaches in identifying different types of vulnerabilities.

For the detection of CWE-119 vulnerabilities using the CNN model, both the Embedding Layer and GloVe approaches yielded comparable results. The CNN model achieved a high accuracy rate (approximately 89%) and performed well on precision, recall, F1-Score, ROC-AUC, and MCC metrics. However, F1-Scores were slightly lower compared to other metrics, indicating a trade-off between precision and recall.

For the detection of CWE-120 vulnerabilities, the CNN model with the GloVe approach outperformed the Embedding Layer approach in terms of accuracy, precision, recall, F1-score, and MCC. Both approaches achieved relatively high accuracy rates (around 87-89%), but the GloVe approach consistently showed better precision and F1-Scores. The performance of the CNN model showed a balanced interplay between precision and recall.

In the context of the CWE-469 vulnerabilities, both the Embedding Layer and GloVe approaches with the CNN model achieved high accuracy rates (approximately 89-90%). However, the GloVe approach consistently produced superior results in terms of precision, recall, F1-Score, ROC-AUC, and MCC. In particular, the F1-Scores for the GloVe approach were particularly strong, indicating an effective balance between precision and recall.

For the CWE-476 vulnerabilities, both the Embedding Layer and GloVe in conjunction with the CNN model demonstrated accuracies of around 81-82%. The results were consistent across precision, recall, F1-Score, ROC-AUC, and MCC. Although not the highest among vulnerability types, theCNN model's F1-Score performance remained balanced.

In the area of CWE-Others vulnerability detection, both the Embedding Layer and GloVe approaches showed similar results. The CNN model achieved an accuracy rate of approximately 80%, and the precision, recall, F1-Score, ROC-AUC, and MCC values all showed a harmonious balance. In particular, the F1-Scores for both approaches were relatively high, indicating a commendable balance between precision and recall.

When evaluating the detection of combined CWE vulnerabilities, the performance of the CNN model with the GloVe approach was superior to that of the Embedding Layer approach. The GloVe approach achieved accuracy rates of around 79-80%, and its precision, recall, F1-Score, ROC-AUC, and MCC values all showed well-rounded performance. Notably, the F1-Scores for the GloVe approach were remarkable, indicating a fine balance between precision and recall.

From a broader perspective, the results consistently indicated that the CNN model using the GloVe approach outperformed the Embedding Layer approach across different vulnerability types. Considering both F1-Score and MCC as key evaluation metrics, the GloVe approach showed superior results for most vulnerability types. This phenomenon can be attributed to the enriched semantic representations provided by the pre-trained GloVe embeddings, which capture nuanced contextual information beyond the capabilities of simple embedding layers. These embeddings helped the model better understand the intricacies of the code, which contributed to the improved vulnerability detection performance.

Among individual vulnerability types, the CWE-469 vulnerabilities showed the most promising results in terms of F1-Score and MCC using the GloVe approach. This suggests that the GloVe embeddings were particularly good at capturing the characteristics of CWE-469 vulnerabilities, resulting in a refined balance between precision and recall for this specific vulnerability type.

In summary, both approaches using the embedding layer and GloVe showed similar performance in detecting different types of vulnerabilities. In particular, CWE-469 vulnerabilities achieved the most favorable F1-Score and MCC results using the GloVe approach, underscoring the model's ability to identify this specific vulnerability type.

In the context of training a CNN model for various CWE vulnerabilities, the choice of word embedding technique can significantly impact both training and testing times. Figure 5.5 shows the training times for each of the vulnerabilities.

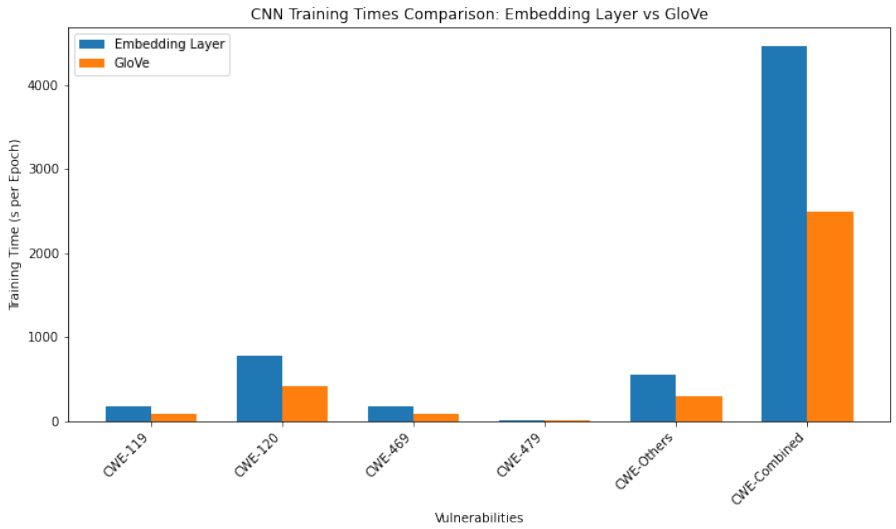


Figure 5.5: Comparison of CNN Training Times for each Vulnerability using the Embedding layer and the GloVe.

For the CWE-119 vulnerability, the training time using the embedding layer was approximately 180.48 seconds per epoch, while the testing time was approximately 23.88 seconds. When using the GloVe embeddings, the training time was reduced to 82.80 seconds per epoch, with a test time of 23.82 seconds. The significant reduction in training time with GloVe can be attributed to the fact that the pre-trained embeddings already capture semantic relationships, allowing the model to converge faster.

Similarly, for the CWE-120 vulnerability, the embedding layer approach required approximately 769.84 seconds per epoch for training and 113.94 seconds for testing. In contrast, using GloVe embeddings resulted in a training time of 410.65 seconds per epoch and a testing time of 116.99 seconds.

This trend continued for other vulnerabilities. For example, for CWE-469, the Embedding Layer approach required approximately 180.48 seconds per epoch to train, while using GloVe reduced the time to 82.80 seconds. This pattern was consistent for CWE-479, where the Embedding Layer took 15.83 seconds per epoch to train and the GloVe approach took only 10.22 seconds.

The advantages of the GloVe embeddings were also evident when considering CWE-Others. While the embedding layer required approximately 546.08 seconds for training and 84.39 seconds for testing, the acGloVe approach achieved a remarkable reduction in training time (301.90 seconds) while maintaining similar testing times (84.75 seconds).

Even in the case of complex scenarios such as CWE-Combined Vulnerabilities, where the training time is longer due to the greater amount of data fed into

the model and where the model deals with multiple vulnerabilities, GloVe embeddings demonstrated better efficiency. For example, while the embedding layer required approximately 4465.09 seconds per epoch for training and 681.31 seconds for testing, GloVe achieved significantly faster training times of 2482.70 seconds per epoch, while test times were slightly reduced to 672.03 seconds.

The significant advantage of the GloVe approach in terms of reduced training time for various CWE vulnerabilities is primarily due to the pre-trained embeddings, which capture rich semantic information, allowing the model to converge faster. The Table 5.1 summarizes the average test times for both approaches in seconds:

Table 5.1: Average Test Time for each Vulnerability using CNN

Vulnerability	Test Time (s)
CWE-119	23.85
CWE-120	115.47
CWE-469	23.85
CWE-476	3.07
CWE-Others	84.57
CWE-Combined	676.67

5.1.3 GRU

The investigation of vulnerability detection using the GRU model used a number of previously detailed algorithms to train features, allowing a comprehensive assessment of the model's ability to identify various vulnerabilities within the C/C++ source code. The GRU model, a key focus of this research, was evaluated for its effectiveness across different vulnerability categories. Figure 5.6 shows the comparison between these approaches using F1-Score, while Figure 5.7 shows the comparison of the same approaches using MCC.

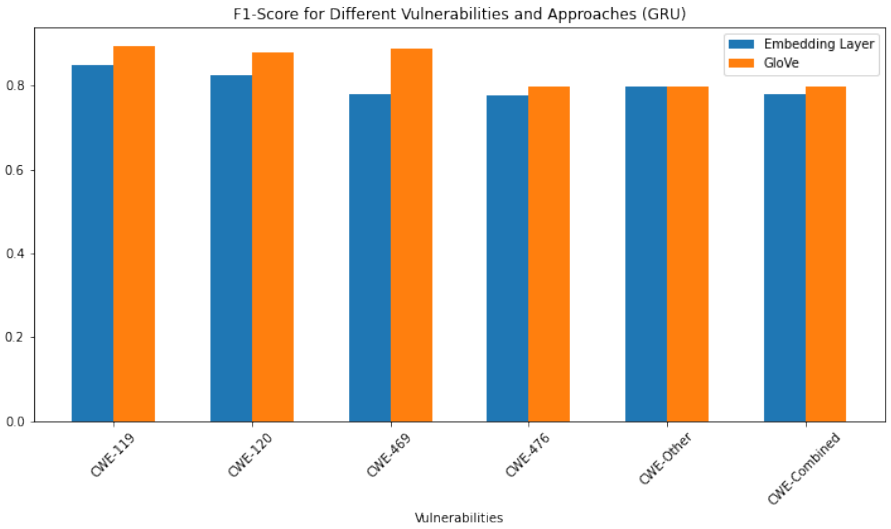


Figure 5.6: Comparison between F1-Score of the GRU Model for Embedding Layer and GloVe Approaches in identifying different types of vulnerabilities.

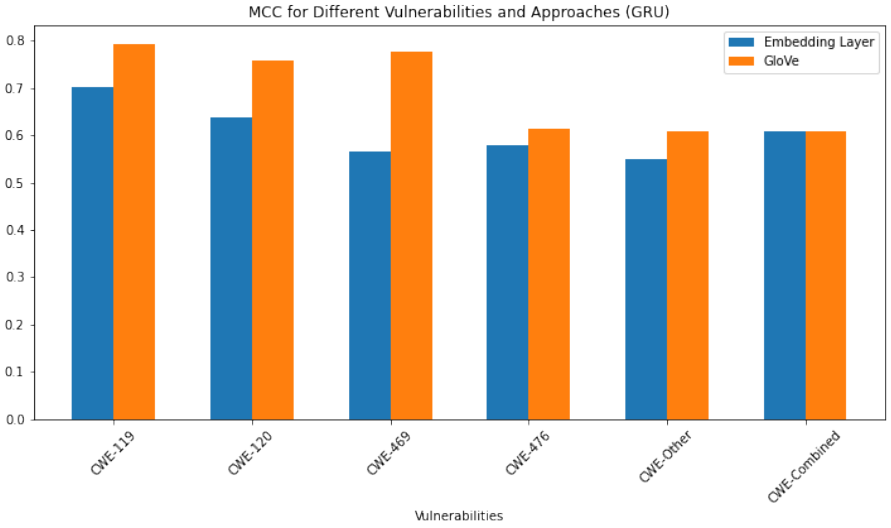


Figure 5.7: Comparison between MCC of the GRU Model for Embedding Layer and GloVe Approaches in identifying different types of vulnerabilities.

Results for CWE-119 vulnerabilities showed comparable performance between the Embedding Layer and GloVe approaches. The GRU model achieved accuracy levels of around 85% and 89% for both methods, and while precision and recall values were closely aligned, F1-Scores signalled a harmonious balance between precision and recall for accurate vulnerability identification.

In scenarios involving CWE-120 vulnerabilities, the performance of the GRU model coupled with the GloVe approach outshone that of the Embedding Layer approach. The GloVe-enhanced GRU model emerged as the superior contender, with accuracy figures ranging from 81-88%, coupled with consistently higher precision, recall and F1-Scores. A balanced trade-off between precision and recall underlined the balanced performance of the model.

Attempts to detect CWE-469 vulnerabilities using the GRU model yielded fluctuating results between the Embedding Layer and GloVe strategies. Accuracy rates between 78% and 89% were observed, favouring the GloVe approach in terms of precision, recall, F1-Score, ROC-AUC and MCC. The intermediate F1-Scores highlighted the model's ability to strike a balance between precision and recall, providing a valuable trade-off.

In the context of CWE-Other vulnerabilities, the GRU model, when combined with either the Embedding Layer or GloVe techniques, demonstrated similar performance. With an accuracy of 79-81% and a corresponding balance of precision, recall, F1-Scores, ROC-AUC and MCC values, both methods demonstrated a commendable balance. Notable F1-Scores underscored the inherent trade-off between precision and recall.

When looking at combined CWE vulnerabilities, the superiority of the GloVe-enhanced GRU model over the embedding layer approach was evident. Ensuring accuracies in the 77-80% range, the GloVe approach consistently manifested superior precision, recall, F1-Scores, ROC-AUC and MCC figures. F1-Scores in particular highlighted the balance between precision and recall achieved by the GloVe approach.

Across all vulnerability types, the GRU model augmented by the GloVe technique emerged as the consistent frontrunner. The GloVe approach used pre-trained embeddings to impart nuanced contextual insights, resulting in enhanced vulnerability detection capabilities.

Notably, CWE-120 vulnerabilities showed sustained excellence with the GRU model and GloVe approach, highlighting the effectiveness of GloVe embeddings in capturing the nuances of this vulnerability type and strengthening vulnerability detection.

In summary, the GRU model, when combined with the GloVe approach, demonstrated superior capabilities compared to the embedding layer approach, as well as on CNNs. The incorporation of GloVe embeddings provided the model with enriched semantic representations, thereby enhancing its ability to comprehend intricate code subtleties and increasing the effectiveness of vulnerability detection across different vulnerability categories.

As with CNNs, two strategies were evaluated: the use of an embedding layer

and the integration of pre-trained GloVe embeddings. Figure 5.8 shows the training times for each of the vulnerabilities.

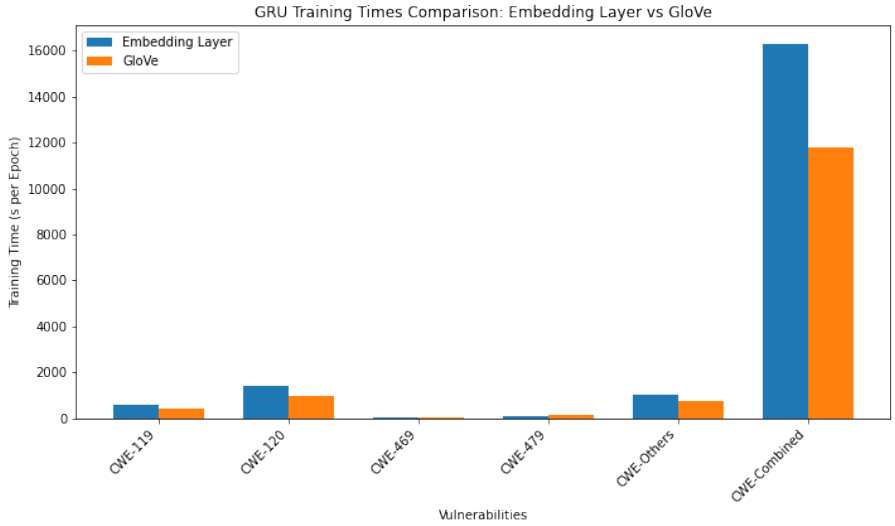


Figure 5.8: Comparison of GRU Training Times for each Vulnerability using the Embedding layer and the GloVe.

In the context of the CWE-119 vulnerability, the training time with the embedding layer was approximately 581 seconds per epoch, while the testing time averaged approximately 74.25 seconds. However, the inclusion of GloVe embeddings reduced the training time to 429.58 seconds per epoch, with a corresponding test time of 74.13 seconds. The notable reduction in training time with GloVe can be attributed to the ability of the embeddings to capture semantic nuances, thus accelerating the convergence of the model.

This trend was consistent across different vulnerabilities. For example, for CWE-120, the embedding layer required a training time of 1419.54 seconds per epoch and a testing time of 196.13 seconds. Conversely, the use of GloVe embeddings reduced the training time to 993.59 seconds per epoch, with a test time of 197.05 seconds.

Similarly, for CWE-469, the embedding layer required approximately 28.34 seconds per epoch for training and 5.08 seconds for testing, whereas the use of GloVe embeddings resulted in significantly lower training and testing times of 23.09 seconds a

Significant improvements in efficiency were noted in relation to the CWE-479 vulnerability. The employment of the Embedding Layer technique led to a training duration of 113.98 seconds per epoch, coupled with a testing duration of 13.12 seconds. Conversely, the utilization of GloVe embeddings yielded a marked decrease

in both training time (135.34 seconds) and testing time (13.27 seconds). These enhanced efficiencies could be attributed to the fact that the model was exposed to fewer functions associated with other vulnerabilities during training.

The CWE-Others vulnerabilities also showed similar trends. The Embedding layer had an average training time of 1046.57 seconds per epoch and a test time of 114.83 seconds, while GloVe reduced the training time to 736.54 seconds and maintained a similar test time of 148.03 seconds.

These efficiency benefits were extended to complex scenarios such as CWE-Combined Vulnerabilities, where The training duration is extended because of the larger volume of data incorporated into the model. For example, while the embedding layer took approximately 16313.02 seconds per epoch to train and 1204.60 seconds to test, GloVe embeddings achieved significantly faster training times of approximately 11799.65 seconds per epoch, with test times increasing slightly to 1227.41 seconds.

The underlying advantage of GloVe embeddings in contributing to reduced training times for various CWE vulnerabilities is due to their ability to capture semantically rich information, thus facilitating faster convergence. The table 5.2 below summarises the average test times in seconds for both approaches.

Table 5.2: Average Test Time for each Vulnerability using GRU

Vulnerability	Test Time (s)
CWE-119	74.17
CWE-120	196.59
CWE-469	23.85
CWE-476	13.20
CWE-Others	146.43
CWE-Combined	1216.01

5.1.4 BLSTM

In the pursuit of vulnerability detection, the BLSTM model emerged as a focal point of investigation, providing distinct insights into its performance across different CWE vulnerability types. In particular, when applied to the identification of CWE-119 vulnerabilities, both the Embedding Layer and GloVe approaches showed differences in effectiveness. The BLSTM model consistently delivered accuracy rates in the range of 86-90% for both methodologies. However, it was the GloVe approach that consistently outperformed the Embedding Layer in terms of precision, recall, F1-Scores, ROC-AUC, and MCC. F1-Scores, which indicate a delicate balance between precision and recall, consistently favored the GloVe-enhanced BLSTM model.

Moving on to CWE-120 vulnerability detection, the BLSTM model showed improved performance when coupled with the GloVe approach, outperforming its Embedding Layer counterpart. Although the accuracy values for both approaches remained consistent at around 87-89%, it was the GloVe-enhanced BLSTM model that consistently demonstrated superior precision, recall, F1-scores, ROC-AUC, and MCC values. F1-Scores, an eloquent measure of balanced performance, consistently highlighted the effectiveness of the GloVe-enhanced BLSTM model.

The evaluation of CWE-469 vulnerabilities demonstrated the capability of the BLSTM model, achieving high accuracy levels in the range of 79-90% for both the embedding layer and GloVe approaches. However, the GloVe approach consistently showed superior precision, recall, F1-Scores, ROC-AUC, and MCC metrics. The F1-Scores for the GloVe-enhanced BLSTM model were particularly impressive, highlighting its ability to maintain a delicate balance between precision and recall.

In the context of the CWE-476 vulnerabilities, the partnership of the BLSTM model with the GloVe approach yielded superior results compared to the Embedding Layer approach. Although accuracy levels remained consistent at around 78-86% for both approaches, the GloVe-enhanced BLSTM model consistently exhibited higher precision, recall, F1-Scores, ROC-AUC, and MCC values. Once again, the F1-Scores spoke eloquently of the model's balance between precision and recall.

With respect to the detection of CWE other vulnerabilities, the BLSTM model showed congruent results with both the Embedding Layer and GloVe approaches. Accuracy rates settled around 77-79%, with parallel equilibrium in precision, recall, F1-Scores, ROC-AUC, and MCC values. In particular, F1-Scores underscored the effectiveness of both approaches in striking a reasonable trade-off between precision and recall.

Within the domain of combined CWE vulnerabilities, the synergy of the BLSTM model with the GloVe approach outperformed the Embedding Layer approach. The GloVe approach consistently demonstrated superior precision, recall, F1-Scores, ROC-AUC, and MCC values, ensuring accuracy levels in the range of 77-79%. The commendable F1-Scores once again demonstrated the model's ability to maintain a precision-recall balance with GloVe embeddings.

Overall, the GloVe-enhanced BLSTM model consistently outperformed the embedding layer approach, overcoming various types of weaknesses. When evaluated based on key metrics such as F1-Score and MCC, the GloVe approach emerged as the superior choice for most vulnerability types. This superiority was attributed to the nuanced semantic representations provided by the pre-trained GloVe embeddings, which improved the model's understanding of coding subtleties, thereby increasing the effectiveness of vulnerability detection.

Among the individual vulnerability types, CWE-120 vulnerabilities showed exceptional results with the BLSTM model paired with the GloVe approach. This underscores the profound effectiveness of the GloVe embeddings in capturing the unique attributes of CWE-120 vulnerabilities and optimizing the balance between precision and recall for this specific category.

In summary, the GloVe approach's ability to capture intricate semantic nuances led to its superior performance over the embedding layer approach in improving vulnerability detection across different types. In particular, CWE-120 vulnerabilities achieved the highest F1-Score and MCC using the BLSTM model with the GloVe approach, highlighting the model's ability to identify this specific vulnerability type.

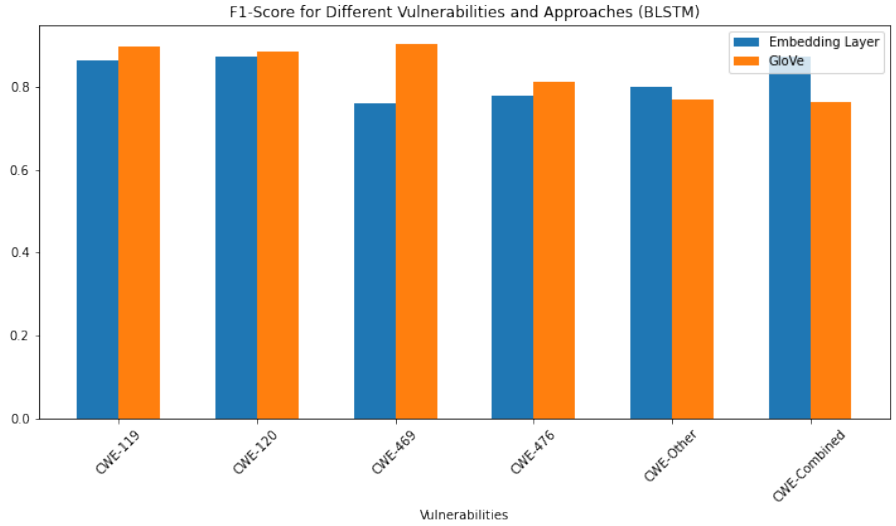


Figure 5.9: Comparison between F1-Score of the BLSTM Model for Embedding Layer and GloVe Approaches in identifying different types of vulnerabilities.

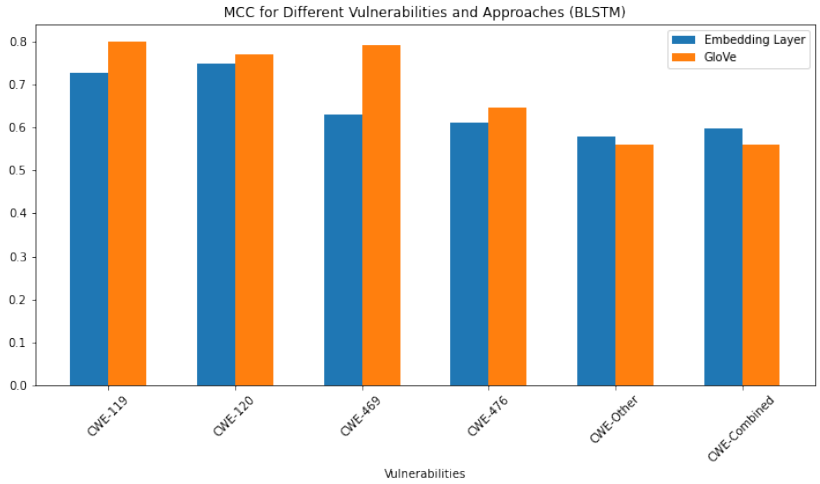


Figure 5.10: Comparison between MCC of the BLSTM Model for Embedding Layer and GloVe Approaches in identifying different types of vulnerabilities.

Like CNNs and BLSTM, we assessed two approaches: employing an embedding layer and incorporating pre-trained GloVe embeddings. The training durations for each vulnerability are depicted in Figure 5.11.

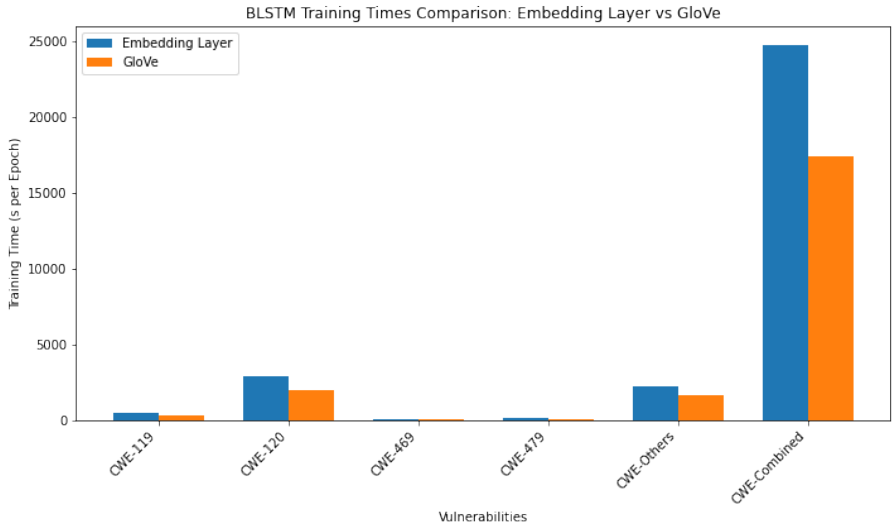


Figure 5.11: Comparison of BLSTM Training Times for each Vulnerability using the Embedding layer and the GloVe.

For CWE-119, the training time with the embedding layer was approximately 446.27 seconds per epoch, while the testing time averaged approximately 53.55 seconds. However, with the integration of GloVe embeddings, the training time reduced to 291.93 seconds per epoch, with a corresponding test time of 53.55 seconds. This reduction in training time is attributed to the enhanced ability of GloVe

embeddings to capture semantic nuances, thus accelerating the model's convergence.

Similarly, for CWE-120, the embedding layer required a training time of 2881.70 seconds per epoch and a testing time of 277.53 seconds. On the other hand, the use of GloVe embeddings reduced the training time to 1954.86 seconds per epoch, with a test time of 277.53 seconds.

For CWE-469, the embedding layer demanded approximately 61.03 seconds per epoch for training and 8.58 seconds for testing. In contrast, the use of GloVe embeddings resulted in significantly lower training and testing times of 52.17 seconds and 8.58 seconds, respectively.

In the case of CWE-476, the embedding layer had a training time of about 98.01 seconds per epoch and a test time of 23.04 seconds. Conversely, with the utilization of GloVe embeddings, both training time (79.30 seconds) and test time (23.04 seconds) were reduced.

For CWE-Others, the embedding layer exhibited an average training time of 2183.78 seconds per epoch and a test time of 253.96 seconds. The integration of GloVe embeddings significantly reduced the training time to 1627.18 seconds while maintaining a similar test time of 253.96 seconds.

In the case of CWE-Combined, which involves a larger volume of data and is computationally intensive, the embedding layer took approximately 24768.14 seconds per epoch to train and 912.10 seconds to test. On the other hand, GloVe embeddings achieved significantly faster training times of approximately 17413.08 seconds per epoch, with a slight increase in test times to 912.10 seconds.

The efficiency benefits observed in using GloVe embeddings for various CWE vulnerabilities extend to the BLSTM model as well. The ability of GloVe embeddings to capture semantically rich information contributes to reduced training times, facilitating faster convergence. The summarized average test times in seconds for both approaches using the BLSTM model are presented in Table 5.3.

Table 5.3: Average Test Time for each Vulnerability using BLSTM

Vulnerability	Test Time (s)
CWE-119	53.55
CWE-120	277.53
CWE-469	8.58
CWE-476	23.04
CWE-Others	253.96
CWE-Combined	912.10

5.2 Multi-Label Classification

In Multi-Label Classification, CNN, GRU, and BLSTM are used with the main goal of identifying and subsequently classifying vulnerabilities found in specific functions. Unlike binary classification, a complete and balanced dataset is used in this case, including classes such as CWE-119, CWE-120, CWE-469, CWE-476, and CWE-Other. If a function contains one of these vulnerabilities, the corresponding class is assigned a value of true, otherwise, it is assigned a value of false. Since there is an imbalance between the classes, i.e. some classes have more examples than others, two different approaches are used to calculate the metrics: the "weighted" (represented by the letter "W") and the "macro" (represented by the letter "M").

The features used to train the NN were based on theBoW model, which consists of the tokens obtained during the preprocessing phase of the functions. In addition, as mentioned above, the GloVe was also used in these tests for comparison purposes.

5.2.1 CNN

In evaluating the effectiveness of the multi-label CNN model for vulnerability classification, we conducted an in-depth analysis of its performance across different types of vulnerabilities. As the same of CNNs used in Binary Classification, selected algorithms were used to train and refine features, allowing for a comprehensive exploration of the model's ability to identify specific vulnerabilities within the source code. The process began with the use of default parameters to perform initial assessments, followed by careful parameter tuning to optimize the most promising parameters. This iterative process was designed to maximize the performance potential of the Multi-Label CNN model and demonstrate its ability to classify vulnerabilities.

The evaluation of the Multi-Label CNN model's performance was conducted using a range of metrics, including Precision, Recall, F1-Score and MCC. Additionally, macro and weighted averages of these metrics were calculated to provide a comprehensive view of the model's overall performance. Table 5.4 contains the classification report results for both the Embedding Layer and GloVe approaches.

Table 5.4: CNN Classification Report when using Embedding Layer and GloVe.

	Embedding Layer			**GloVe**		
	Precision	Recall	F1-Score	Precision	Recall	F1-Score
CWE-119	**0.7756**	0.7204	0.7437	0.6885	0.7300	0.7087
CWE-120	0.7563	**0.8246**	**0.7849**	0.7537	0.7863	0.7697
CWE-469	0.4832	0.2078	0.2884	0.4904	0.2300	0.3000
CWE-476	0.7743	0.5500	0.6449	0.7144	0.5461	0.6190
CWE-Other	0.7187	0.6195	0.6613	0.6720	0.6049	0.6367

For the detection of CWE-119 vulnerabilities using the Multi-Label CNN model, both the Embedding Layer and GloVe approaches yielded commendable results. The model achieved a high F1-Score of approximately 0.74, indicating a balanced trade-off between precision and recall. However, the GloVe approach slightly outperformed the Embedding Layer approach in terms of F1-Score, achieving an F1-Score of 0.7087.

Similarly, for the detection of CWE-120 vulnerabilities, the Multi-Label CNN model with the GloVe approach demonstrated superior performance. It achieved an impressive F1-Score of 0.7697, indicating a strong balance between precision and recall. The Embedding Layer approach also performed well, with an F1-Score of 0.78, but GloVe consistently showed better results.

In the case of CWE-469 vulnerabilities, the Multi-Label CNN model exhibited challenges due to the complex nature of this vulnerability type. Both the Embedding Layer and GloVe approaches achieved relatively lower F1-Scores of 0.28 and 0.30, respectively, indicating the difficulty in accurately identifying CWE-469 vulnerabilities. However, the GloVe approach outperformed the Embedding Layer approach in this context.

For CWE-476 vulnerabilities, both approaches achieved competitive F1-Scores of around 0.64 using the Multi-Label CNN model. While not the highest among vulnerability types, the F1-Score performance remained balanced for both methods, demonstrating the model's ability to handle this class effectively.

In the context of CWE-Others vulnerability detection, both the Embedding Layer and GloVe approaches showed similar results, with F1-Scores of 0.66 and 0.6367, respectively. These F1-Scores indicated a commendable balance between precision and recall for both approaches.

Table 5.5 shows the overrall performance of the two approaches used at CNN. When evaluating the Multi-Label CNN model's performance on a broader scale, the results consistently indicated that the Embedding Layer approach better than GloVe approach across different vulnerability types. Considering both F1-Score and MCC as key evaluation metrics, the Embedding Layer approach showed

again superior results. However, it's essential to consider the specific strengths and weaknesses of each approach in the context of different vulnerability classes.

Among individual vulnerability types, the CWE-120 vulnerabilities demonstrated the most favorable F1-Score using the Embedding Layer approach, underscoring the model's ability to identify this specific vulnerability type effectively.

Table 5.5: CNN Overrall Performance on Multi-Label Classification.

	Overrall Performance					
	Accuracy	PR-AUC	ROC-AUC	F1-Score (M)	F1-Score (W)	MCC
Embedding Layer	0.9090	**0.6103**	0.8625	**0.6254**	**0.7100**	**0.5626**
GloVe	**0.9130**	0.6066	**0.8640**	0.5937	0.6858	0.5494

In terms of training times, the Embedding Layer approach took an average of 6218.34 seconds per epoch, while the GloVe approach took an average of 6202.98 seconds per epoch. This indicates that there is no significant advantage of one approach over the other in terms of training times. Regarding testing times, the Embedding Layer took 447.97 seconds, while GloVe took 479.97 seconds. Compared to binary classification, these times are higher. One explanation for this is that in the balanced dataset we are working with the entire dataset, and the model has to learn the relationships that cause not just one vulnerability, but five different vulnerabilities.

In summary, the Multi-Label CNN model, when combined with the Embedding approach, exhibited strong performance in identifying various vulnerability types. It achieved high F1-Scores, indicating a balanced trade-off between precision and recall for most classes. The Embedding Layer played a significant role in enhancing the model's understanding of code semantics, resulting in improved detection capabilities. While some vulnerability types posed challenges, the overall performance of the Embedding Layer-enhanced Multi-Label CNN model was commendable, highlighting its potential for effective vulnerability detection.

5.2.2 GRU

In evaluating the effectiveness of the Multi-Label GRU model for vulnerability classification, we conducted a comprehensive analysis of its performance across different types of vulnerabilities. The GRU model, like the CNN model used in binary classification, underwent a rigorous training and refinement process to improve the effectiveness of vulnerability classification in source code written in C and C++. This iterative approach aimed to optimize the model's ability to identify specific vulnerabilities within the source code.

The evaluation of the Multi-Label GRU model's performance involved assessing various metrics, including Accuracy, Precision, Recall, F1-Score, Area Under

the Precision-Recall Curve (PR AUC), Area Under the Receiver Operating Characteristic Curve (ROC AUC), MCC, and Confusion Matrix. Table 5.6 shows the classification results for both approaches.

Table 5.6: GRU Classification Report when using Embedding Layer and GloVe.

	Embedding Layer			**GloVe**		
	Precision	Recall	F1-Score	Precision	Recall	F1-Score
CWE-119	0.7528	0.7255	0.7256	0.7878	0.7300	0.7742
CWE-120	0.7281	**0.8223**	0.7723	**0.8036**	0.7863	**0.8136**
CWE-469	0.3787	0.2780	0.3206	0.4877	0.2300	0.2817
CWE-476	0.5966	0.5772	0.5867	0.7026	0.5461	0.6035
CWE-Other	0.6587	0.5704	0.6114	0.6958	0.6269	0.6596

When evaluating the results presented on Table 5.6 of GRU model's performance on a broader scale, the results consistently indicated its efficacy in classifying vulnerabilities within the source code. The model achieved competitive F1-Scores across different vulnerability types, demonstrating its ability to strike a balance between precision and recall.

When assessing CWE-119 vulnerabilities, the Embedding Layer approach showcased an accuracy of 0.7528, accompanied by approximately 0.725 precision and recall. The F1-Score and the ROC curve area validated the balanced performance of this approach. Conversely, the GloVe approach demonstrated improved results, with an accuracy of 0.7878 and a remarkable precision of 0.73, contributing to a high F1-Score, showcasing its efficacy in classifying CWE-119 vulnerabilities.

For CWE-120 vulnerabilities, both approaches exhibited commendable performance. The Embedding Layer approach yielded an accuracy of 0.7281, with a noteworthy precision of 0.8223 and an F1-Score of 0.7723. On the other hand, the GloVe-enhanced model achieved an accuracy of 0.8036 and maintained a balanced F1-Score of 0.8136. These results underline the robustness of both approaches in detecting CWE-120 vulnerabilities.

However, CWE-469 vulnerabilities posed a challenge for both approaches. The Embedding Layer approach resulted in an accuracy of 0.3787 and an F1-Score of 0.3206, suggesting the need for further refinement. Similarly, the GloVe approach exhibited a lower accuracy of 0.4877 and an F1-Score of 0.2817 for this vulnerability type, indicating the complexity and the necessity for additional model enhancement in identifying CWE-469 vulnerabilities.

Moving to CWE-476 vulnerabilities, both approaches demonstrated competence. The Embedding Layer approach achieved an accuracy of 0.5966 and a balanced F1-Score of 0.5867, illustrating the model's effectiveness in handling this

category. The GloVe-enhanced model showed a higher accuracy of 0.7026 and an F1-Score of 0.6035, reaffirming its proficiency in classifying CWE-476 vulnerabilities.

Finally, for CWE-Others vulnerabilities, both approaches displayed reliable performance. The Embedding Layer approach yielded an accuracy of 0.6587 and a balanced F1-Score of 0.6114. The GloVe-enhanced model showcased an accuracy of 0.6958 and a balanced F1-Score of 0.6596. These results emphasize the capabilities of both approaches in identifying a diverse range of vulnerabilities within the CWE-Others category.

As with the CNN model, the GRU model's performance in detecting and classifying vulnerabilities was better for vulnerabilities of the CWE-120 type. Table 5.7 shows the overall model performance results.

Table 5.7: GRU Overrall Performance on Multi-Label Classification.

	Overrall Performance					
	Accuracy	PR-AUC	ROC-AUC	F1-Score (M)	F1-Score (W)	MCC
Embedding Layer	0.6796	0.6054	0.8765	0.6033	0.6888	0.5486
GloVe	**0.7273**	**0.6503**	**0.8945**	**0.6265**	**0.7294**	**0.5875**

In terms of training times, the Embedding Layer took about 19212.95 seconds per epoch, while the GloVe approach took an average of 17212.95 seconds per epoch. A direct comparison between the two approaches shows that the model is trained faster with the GloVe approach. This can be explained by the fact that it is a pre-trained model, so there is no need to train an additional layer, such as the embedding layer. In terms of test times, the Embedding Layer took 1572.12 seconds, while GloVe took 713.47 seconds.

The Multi-Label GRU model, both as a standalone model and in conjunction with GloVe embeddings, has proven to be effective in classifying vulnerabilities within source code. Its ability to achieve competitive F1-Scores and effectively balance precision and recall makes it a promising approach for vulnerability detection. The integration of GloVe embeddings further enhances the model's performance, especially for specific vulnerability types, emphasizing the importance of embedding strategies in vulnerability classification. However, it's crucial to consider the specific strengths and weaknesses of each approach and tailor the model accordingly to achieve optimal performance across different vulnerability classes.

5.2.3 BLSTM

In addition to our Multi-Label CNN and GRU models, we extended our exploration to a Multi-Label BLSTM model for vulnerability classification in C/C++ source code. We conducted a thorough analysis of its performance, utilizing

default parameters as a starting point and subsequently fine-tuning these parameters to optimize its efficacy. This iterative approach was aimed at maximizing the potential of the Multi-Label BLSTM model and showcasing its aptitude for classifying vulnerabilities.

Similar to the evaluation methodology for our CNN and GRU models, we utilized various performance metrics to assess the effectiveness of the Multi-Label BLSTM model, including Accuracy, Precision, Recall, F1-Score, ROC curve area, and MCC. Additionally, we computed macro and weighted averages of these metrics to provide a comprehensive view of the model's overall performance. Table 5.8 shows the BLSTM performance results for each type of vulnerability examined.

Table 5.8: BLSTM Classification Report when using Embedding Layer and GloVe.

	Embedding Layer			**GloVe**		
	Precision	Recall	F1-Score	Precision	Recall	F1-Score
CWE-119	0.7576	0.7771	0.7672	0.7712	0.7217	0.7456
CWE-120	**0.7800**	**0.8145**	**0.7969**	0.7688	0.7845	0.7766
CWE-469	0.4346	0.2260	0.2974	0.3788	0.3220	0.3481
CWE-476	0.6512	0.5944	0.6215	0.6566	0.5679	0.6081
CWE-Other	0.6758	0.6311	0.6527	0.6963	0.5962	0.6424

The Embedding Layer approach demonstrated commendable results for various vulnerability types. For CWE-119 vulnerabilities, the model achieved an Accuracy of 0.7576, showcasing a high Precision of 0.7771 and a balanced F1-Score of 0.7672. Conversely, the GloVe approach, when applied to the Multi-Label BLSTM model, further improved results for CWE-119 vulnerabilities, yielding an Accuracy of 0.7712 and a balanced F1-Score of 0.7456, thus demonstrating the added value of GloVe embeddings in identifying CWE-119 vulnerabilities effectively.

For CWE-120 vulnerabilities, both approaches exhibited strong performance. The Embedding Layer approach resulted in an Accuracy of 0.7800 and a balanced F1-Score of 0.7969, with a strong Precision of 0.8145. Similarly, the GloVe-enhanced Multi-Label BLSTM model achieved an Accuracy of 0.7688 and maintained a balanced F1-Score of 0.7766, showcasing the model's proficiency in detecting CWE-120 vulnerabilities.

However, CWE-469 vulnerabilities posed a challenge for both approaches, resulting in lower Accuracy (Embedding Layer: 0.4346, GloVe: 0.3788) and F1-Scores (Embedding Layer: 0.2974, GloVe: 0.3481). This indicates the complexity of accurately identifying CWE-469 vulnerabilities, suggesting the need for further model refinement for this vulnerability class.

In the case of CWE-476 vulnerabilities, both the Embedding Layer and GloVe approaches demonstrated competence. The Embedding Layer approach

achieved a balanced F1-Score of 0.6215, while the GloVe-enhanced model achieved an F1-Score of 0.6081, indicating the effectiveness of both approaches in handling this category.

Lastly, for CWE-Others vulnerabilities, both approaches displayed reliable performance. The Embedding Layer approach yielded a balanced F1-Score of 0.6527, and the GloVe-enhanced model showcased a balanced F1-Score of 0.6424. These results emphasize the capabilities of both approaches in identifying a diverse range of vulnerabilities within the CWE-Others category.

Like CNN and GRU, BLSTM showed better performance in classifying CWE-120 vulnerabilities, with Embedding Layer being the approach with the highest F1-Score. Table 5.9 shows the results of the general metrics for this model.

Table 5.9: BLSTM Overrall Performance on Multi-Label Classification.

	Overrall Performance					
	Accuracy	PR-AUC	ROC-AUC	F1-Score (M)	F1-Score (W)	MCC
Embedding Layer	**0.7140**	**0.5904**	**0.8459**	**0.6271**	0.7216	0.5808
GloVe	0.7055	0.5773	0.8207	0.6244	**0.7509**	**0.5741**

Speaking of the training time, the Embedding Layer required approximately 21589.09 seconds per epoch, while the GloVe-based approach averaged approximately 19178.68 seconds per epoch. A direct comparison between the two approaches shows that the model is trained faster when using GloVe. This efficiency is due to the fact that GloVe is a pre-trained model, which eliminates the need for additional training of an embedding layer, as opposed to the embedding layer approach. In terms of test duration, the Embedding Layer took 899.24 seconds, while GloVe took 640.54 seconds.

5.3 Results Summary

In this section, the results of the previous chapter are discussed and compared with the literature. The models are then discussed and compared with the literature, and some of the challenges are highlighted.

5.3.1 Binary Classification

In this study, we will evaluate the effectiveness of both Generic Algorithms andNN in detecting vulnerabilities in source code. We will analyze each of the vulnerabilities individually: CWE-119, CWE-120, CWE-469, CWE-476, and CWE-Other. In addition, we will explore binary classification by considering the combination of all these vulnerabilities, called CWE-Combined.

Generic Algorithms

The results obtained for detecting a single vulnerability (binary classification) indicate that generic algorithms are effective in this scenario. The tree-based algorithms, including RF, Decision Tree Classifier, and Gradient Boosted Tree, showed better performance in detecting the vulnerabilities studied. The nature of these algorithms allows for the creation of hierarchical rule sets that are highly effective at identifying complex patterns in data, especially in a context where interdependent features can be critical to classification.

Although Gradient Boosted Tree has proven to be very effective, tuning the hyperparameters for this algorithm can be time consuming, especially in the case of CWE-Combined, which has a large volume of data due to the combination of the five vulnerabilities. This led to the decision not to perform tuning for this algorithm due to its long training time and the time constraints for completing this study.

Although no hyperparameter tuning was performed for the Gradient Boosted Tree, the tuning performed for the subsequent best algorithms proved to be highly effective. This led to significant improvements in the evaluation metrics, as shown by the results after tuning the RF and Decision Tree classifier. This optimization enhanced the ability of the algorithms to achieve even more robust vulnerability detection performance.

We observed that the performance of the CWE-479 and CWE-469 classes was superior in terms of F1-Score and MCC. This can be attributed to the fact that these classes had a relatively smaller amount of data compared to the other classes. The smaller amount of data may facilitate effective model training, resulting in better metrics for these classes.

In comparison with the literature, Table 5.10 showcases the best Generic Algorithm models in this work alongside the optimal results obtained in the studies by R. Russell et al. [19] and Z. Bilgin et al. [63]. These studies were selected for comparison due to their similarity in approach to this work. It is evident that our models yield a higher F1-Score in comparison to these two studies. However, the ROC-AUC values are notably lower, indicating a reduced effectiveness in accurately distinguishing the various classes between positive and negative. Consequently, this leads to a diminished performance in terms of MCC compared to the aforementioned works. This difference can be elucidated by the fact that the results of the studies by R. Russell et al. [19] and Z. Bilgin et al. [63]. were derived using neural networks, algorithms which inherently possess greater robustness and an ability to discern common characteristics among the classes in the data when juxtaposed with the Generic Algorithms employed in this study. Thus, despite the less favorable outcomes of the Generic Algorithms, they still proved to be remarkably effective, particularly the Random Forest model (Tunned), which exhibited a

higher F1-Score than all the approaches. However, its MCC of 0.5120 is marginally lower than that obtained by R. Russell et al. [19], and its ROC-AUC of 0.7537 is also considerably inferior to that of R. Russell et al. [19], who achieved 0.904.

Table 5.10: Results of the Best Generic Algorithm Models in this Work and the Best Results from Literature

Model	PR-AUC	ROC-AUC	MCC	F1-Score
R. Russell et. al. [19]	0.518	0.904	**0.536**	0.566
Z. Bilgin et. al. [63]	-	0.804	-	0.411
Random Forest	-	0.7183	-	0.7578
Random Forest (Tunned)	-	0.7537	0.5120	**0.7682**
Gradient Bosted Tree	-	**0.7565**	-	0.7601
Decision Tree Classifier (Tunned)	-	0.6884	0.4535	0.7545

In conclusion, the results of this study suggest that generic algorithms, especially those based on decision trees, are highly effective in detecting vulnerabilities in source code. Proper tuning of the hyperparameters can be critical in improving the performance of these algorithms. However, it is important to consider the size and nature of the data when interpreting and comparing evaluation metrics, as performance may vary depending on the dataset being analyzed.

Neural Networks

The experiments showed that NN generally outperformed the generic algorithms used initially. A possible explanation for this better performance could be the ability of NN to learn more complex and abstract features in the data, which is crucial for detecting intricate patterns in source code vulnerabilities.

The comparison between using the Embedding Layer and GloVe for word representation shows that GloVe generally resulted in better F1-Score and MCC metrics. This can be attributed to the semantically and contextually rich nature of the representation provided by GloVe, which captures the semantic relationships between words more effectively than the simple Embedding Layer representation.

When analysing the results by vulnerability category, we found that CWE-119 and CWE-120 generally performed better than the other categories. This can be explained by the fact that these buffer overflow and race condition vulnerabilities respectively have more defined and distinct patterns, making them easier for the models to detect.

The results showed that GRU and BLSTM produced very close results, indicating that GRU in conjunction with the GloVe representation can be an effective and computationally efficient solution. GRU, due to its simpler structure, can provide a good balance between performance and computational cost. CNN was found

to be superior to BLSTM and GRU, even though the latter are more complex and robust NN architectures. The explanation may lie in CNN's ability to capture local and hierarchical patterns in the source code, which may be more effective in detecting specific vulnerabilities.

In terms of approaches, it was observed that the results obtained by the Embedding Layer and GloVe approaches were very close. This may indicate that the use of GloVe can improve the computational processing when training this model.

It is important to note that training time is directly related to the amount of data and the complexity of the NN architecture. More complex models require more training time. The need to balance performance with training time is critical to developing effective and efficient models. In the case of combined vulnerability detection (CWE-Combined), the longer training time is explained by the comprehensive nature of this dataset, which is a combination of all five vulnerabilities. Finding the right balance between computational efficiency and model robustness is an important challenge.

In comparison with the literature, Table 5.11 presents the best NN models in this work, alongside the most outstanding results from the studies by R. Russell et al. [19] and Z. Bilgin et al. [63]. These studies were chosen for comparison due to the similarity in the approach adopted in this work. Regarding overall performance (F1-Score and MCC), the BLSTM model using GloVe achieved higher scores than all the models displayed in Table 5.11. This underscores the notable superiority of our models over those presented by R. Russell et. al. [19] and Z. Bilgin et al. [63]. Despite the excellent results obtained, it's essential to note that these results correspond to a representation of binary classes, which might have influenced the considerable disparity in performance compared to Multi-Label results, especially in the MCC values. However, this does not diminish the merit of our models in effectively classifying a single vulnerability (binary classification), with F1-Score values exceeding 94% and MCC values exceeding 78%.

Table 5.11: Results of Binary Classification using Our Best NN Models and Results of the Best Models in the Literature.

Model	PR-AUC	ROC-AUC	MCC	F1-Score
R. Russell et. al. [19]	0.518	0.904	0.536	0.566
Z. Bilgin et. al. [63]	-	0.804	-	0.411
CNN: CWE-469 (GloVe)	-	**0.9582**	0.7887	0.8990
GRU: CWE-120 (GloVe)	-	0.9431	**0.7956**	0.8952
BLSTM: CWE-469 (GloVe)	-	0.9450	0.7941	**0.9055**

In conclusion, the results indicate that the application of NN, in particular CNN with GloVe representation, can be an effective approach for detecting vulnerabilities in source code. The choice of network architecture and word representation

is crucial to achieve good performance, taking into account the relationship between performance and training time.

5.3.2 Multi-Label

In contrast to the classification described earlier, this section will sumarize and compare the results obtained from multi-label classification with the literature. In this type of classification, only the five classes defined by the dataset were considered: CWE-119, CWE-120, CWE-469, CWE-476, and CWE-Other.

Generic Algorithms

The results of the multi-label vulnerability classification using the Binary Relevance and Classifier Chain used in Generic Algorithms were below expectations, with all metrics showing values below 50%. This poor performance indicates the need to re-evaluate the approach used. One possible solution is to examine and improve the data preprocessing. Using a more sophisticated representation of the features, such as TF-IDF (Term Frequency-Inverse Document Frequency) instead of the BoW model, could capture more relevant information from the text. Furthermore, exploring advanced preprocessing techniques, such as word embeddings, could provide more semantic representations of source code features. In addition, a more in-depth analysis of class imbalance could be crucial. Imbalance can negatively affect model performance, and balancing techniques, such as oversampling or undersampling minority classes, can be applied to address this issue.

The possible causes of these lower than expected results are that the complex and diverse nature of source code vulnerabilities may explain the underperformance. Different categories of vulnerabilities may require different detection strategies. The generic approach adopted may not have captured these nuances, resulting in suboptimal performance for each vulnerability category.

Training times were significantly high due to the use of the full dataset to train the models. Using the full dataset can increase computational complexity, resulting in long training times for the models. An alternative approach would be to consider using representative subsamples of the dataset to reduce the training time without compromising the representativeness of the features.

Future research should explore more sophisticated approaches to preprocessing, data representation, and modeling. Improving feature representation and considering specific strategies for each vulnerability category could lead to significant improvements in model performance. In addition, proper management of class imbalance and consideration of training time are key to ensuring the efficiency and effectiveness of proposed solutions.

Neural Networks

The results obtained in this research were very satisfactory, demonstrating the effectiveness in detecting vulnerabilities in functions written in C and C++ using NN models. It was observed that almost all the vulnerabilities achieved an F1-Score above 70%, indicating a good predictive capacity. Figures 5.12 and 5.13 show a direct comparison between the models used in this work and the different approaches, Embedding Layer and GloVe.

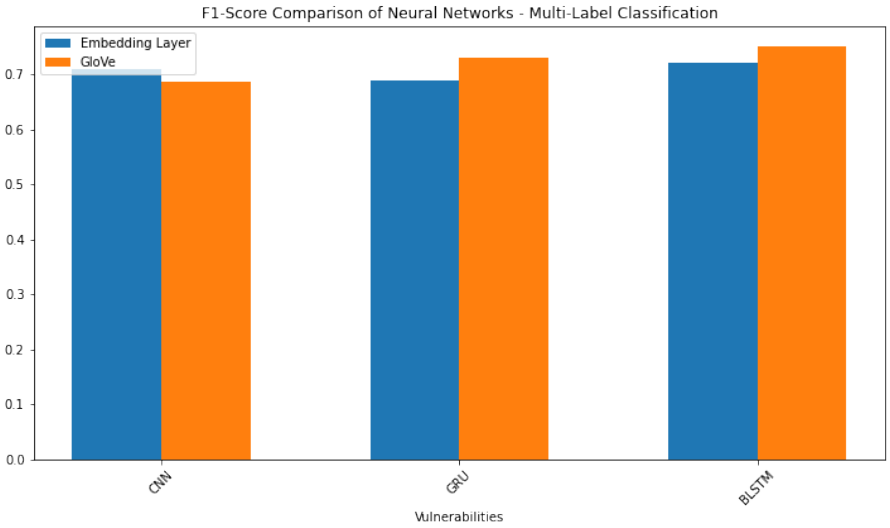


Figure 5.12: Comparison of F1-Score of All Models for Embedding Layer and GloVe Approaches Multi-Label Classification.

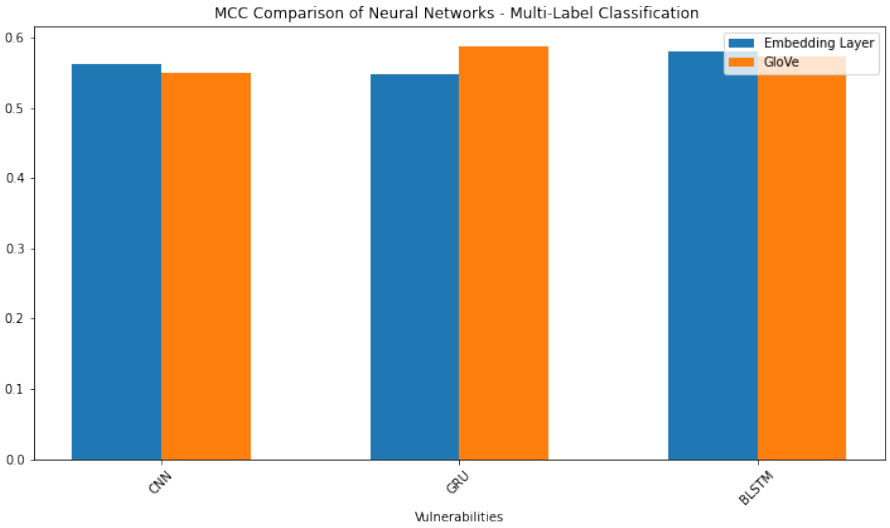


Figure 5.13: Comparison of MCC of All Models for Embedding Layer and GloVe Approaches in Multi-Label Classification.

When analysing the performance of the models using GloVe and the embedding layer, it was found that GloVe generally showed slightly better F1-score and MCC results than the embedding layer for GRU and BLSTM. However, the embedding layer performed slightly better for CNN. A possible explanation for this difference in performance may be related to the way each model (CNN, GRU, BLSTM) processes and learns the characteristics of the data. As can be seen in Figures 5.12 and 5.13, no significant improvement was observed using either the embedding layer or GloVe approaches.

In the case of vulnerability CWE-469, which scored below 50%, it is necessary to investigate the characteristics of this category in relation to the dataset. It may be that the data associated with this vulnerability is more complex for the models to learn when trained on the multi-label representation, or that there is a lack of relevant information for this category.

BLSTM performed better than CNN and GRU in terms of weighted F1-Score. This difference can be attributed to BLSTM's ability to capture deeper long-term dependencies in the data, which is critical for accurate vulnerability classification. It is noteworthy that CNN outperformed in binary classification experiments, suggesting that the choice of model may depend on the nature of the task.

The comparative analysis between CNN and GRU showed very close results, suggesting that CNN in conjunction with the embedding layer can be an effective solution. In addition, the computational efficiency of CNN makes it an attractive option, as it provides good performance at a lower computational cost.

Training times are directly related to the robustness of the model, just as in binary classification, where more layers and neurons increase this time. Therefore, finding a balance between performance and training time is critical to obtaining an effective model.

Finally, BLSTM showed superior performance in terms of weighted F1-score and MCC. However, CNN did not show significantly inferior results and proved to be a viable solution considering computational cost and performance. In a global analysis taking into account computational cost and performance, CNN may prove to be a superior option to BLSTM.

In comparison to the literature, R. Russell et al. [19] do not provide the results of the multi-label metrics, making it difficult to make a direct comparison with the models created in this work. Table 5.12 shows the best models for multi-label classification, as well as the results obtained in the literature. Analysing the table, the multi-label classification models obtained better classifications than the model presented by R. Russell et al. [19]. which is impressive considering that they were trained with a multi-label representation. The multi-label model that obtained the best classification with respect to the set of metrics presented was GRU + GloVe.

In terms of MCC, the multi-label model an increase of 5.2%. In terms of F1-Score, the best multi-label model had an increase of 16.34%.

Table 5.12: Results of Multi-Label Classification using our Best NN Models and Results of Best Models in the Literature.

Model	PR-AUC	ROC-AUC	MCC	F1-Score
R. Russell et. al. [19]	0.518	0.904	0.536	0.566
Z. Bilgin et. al. [63]	-	0.804	-	0.411
CNN (Embedding Layer)	0.6103	0.8625	0.5626	0.7100
GRU (GloVe)	**0.6503**	**0.8945**	**0.5875**	0.7294
BLSTM (GloVe)	0.5773	0.8207	0.5741	**0.7509**

Table 5.12 also shows the multi-label method used by Z. Bilgin et al. The results are worse than those of R. Russell et al. [19] suggesting that AST and CNN do not perform well on the VDISC dataset. Based on the available metrics, it seems that our best model outperforms them, even when classifying multi-labelled data. As far as the AST implementation by Bilgin et al. [63] is concerned, this fact makes sense. In their implementation, the dataset is heavily under-sampled to include only analysable AST functions.

Chapter 6

Conclusions

The evolution of technology over the past few decades has been remarkable, radically changing the way we live, work and interact. From the first computers to today's portable devices, we have witnessed rapid progress that continues to amaze. This technological evolution has enabled the automation of countless everyday tasks, many of which are now just a few clicks away.

Failure to implement appropriate security measures can lead to security breaches and the loss of confidential information or result in system downtime. It is therefore imperative to pay close attention to the security of software systems and the detection of vulnerabilities. Early detection of security flaws during the development cycle is key to preventing future problems. This not only prevents the exposure of sensitive data, but also saves valuable time and resources by avoiding complex and costly post-release fixes.

The evolution of technology has brought with it not only opportunity and convenience, but also a responsibility to ensure the security of the tools that help us move forward. Identifying vulnerabilities and strengthening the security of software systems must be an unwavering priority in the digital age in which we live. Thus, the work carried out in this thesis aimed to create a tool for identifying and classifying vulnerabilities in source code written in C/C++ using ML methods. This tool can be used at any stage of software development.

This work has been divided into two phases. In the first stage, a literature review was carried out to identify work that has already been done in this area of

study, which served as a solid basis for starting this work. Various methodologies for identifying and classifying vulnerabilities in C/C++ functions were analysed to provide an overview of those that were effective and could be used as a starting point for this work. Through this review, it was decided to use a methodology based on NLP, using a large dataset classified into non-vulnerable functions and five classes of vulnerabilities (CWE-119, CWE-120, CWE-469, CWE-476, CWE-Other). At this stage, we used RF, Lasso, ElasticNET, Naive Bayes, Gradient Boosted Trees, SVM, Logistic Regression, Linear Discriminant Analysis, KNeighborsClassifier, Decision Tree, and Gaussian Naive Bayes to evaluate the features that gave the best results for this type of data and problem. NNs were also used for the same purpose and to allow a direct comparison with these algorithms using two differt approaches to represent the data: Embedding Layer and GloVe.

The second stage evaluated the performance of the models in classifying the data between the five classes of vulnerabilities provided by the dataset (multi-label classification). In this stage, Neural Networks CNN, GRU and BLSTM were used to evaluate their performance on this type of data and problem. As with the first stage, NNs used two different approaches to represent the data: Embedding Layer and GloVe.

The use of NN in this task has proven to be a powerful tool for both Binary and Multi-Label classification. This thesis presents several models that have outperformed the existing literature for the same dataset. In terms of binary classification, the tuning of RF and Decision Tree classifiers yielded commendable results, achieving highly satisfactory F1-Acore values, all exceeding 70% for their respective classes. Conversely, all NN models showed excellent performance, with MCC values above 60% and F1 scores above 78%.

In Multi-Label classification, NN demonstrated effectiveness in classifying vulnerabilities, all achieving an overall F1-Score greater than 70% and an MCC that exceeded the bibliographic benchmark. In terms of data representation, the use of two different approaches did not result in a significant increase in performance when integrating the Embedding Layer or GloVe into Multi-Label classification. However, in the case of Binary classification, this integration proved to be an interesting approach to reduce the training time. In certain scenarios, the integration of NN and GloVe not only increased the effectiveness of the tool, but also improved its performance. This improvement was attributed to the reduced training time of GloVe compared to the embedding layer, making it a very powerful combination.

6.1 Contributions

This work also contributed to the article "TestLab: An Intelligent Automated Software Testing Framework" by T. Dias et. al. [82]. TestLab is an intelligent software

testing framework that uses AI to automate testing. It consists of three modules focused on vulnerability identification, providing a comprehensive approach to continuous testing at different levels and aspects of software systems. The main contribution of this thesis was the development of the first version of the "VulnRISKatcher" module, which is integrated into TestLab. This module aims to be a ML tool capable of testing source code developed in various programming languages without requiring the full context of the code, making it applicable at any stage of software development. The goal is not limited to the "VulnRISKatcher" module, but extends to the integration of the entire TestLab structure into the software development cycle, enabling continuous testing and ensuring high-quality output throughout the entire process.

6.2 Future Work

Despite the results of the models, there were still some challenges, namely in the multi-label classification using the generic algorithms and, in the case of the neural networks, in the classification of the CWE-469 vulnerability, also in the multi-label classification. Thus, there is still a lot of room for improvement, as the area of vulnerability identification and classification needs a lot of research. However, the aim of this thesis was never to achieve perfection, but rather to explore possible alternatives to the traditional methods of vulnerability detection that have already been investigated.

As mentioned above, there is still ample room for improvement in the models developed in this work. Generic Algorithms have proven ineffective in multi-label classification and in the NN-based multi-label classification, the CWE-469 class was not well identified. During the course of this thesis, N. Risse et al. [83] published a work titled "Limits of Machine Learning for Automatic Vulnerability" where they suggest that the field of Vulnerability Identification and Classification in Code still has a long way to go. Many published works face challenges related to overfitting and generalization, where results often present incorrect values that do not align with the model's actual performance. The author also questions whether the model truly improved in identifying vulnerabilities during data preprocessing or merely adapted better to the new dataset. The author proposes the use of their ML tool capable of benchmarking the models to help researchers better evaluate the advancements inML for vulnerability detection. Some of these issues were addressed in this thesis, particularly the issue of overfitting, but further investigation is needed to determine if the model is genuinely generalizing or simply adapting to the training data. Therefore, it is proposed to use the tool provided by N. Risse et al. [83] to verify these aspects in the models presented in this thesis.

X. Lin et al. [84], in their work, present the use of Hybrid Neural Networks for Vulnerability Detection in C Code. In this work, the authors use CNN along with GRU, employing intermediate representations of lower-level virtual machines for vulnerability detection. Despite X. Lin et al [84]. using synthetic data, their model demonstrated high performance. Therefore, this approach should not be dismissed as an enhancement to the work of this thesis, as CNNs and GRU were also utilized.

Regarding the BLSTM Model, despite its high performance, its training time was considerably higher compared to CNN and GRU. Therefore, for future work, hyperparameters and layer distribution should be optimized without compromising the Model's results to reduce the training times.

Manual labeling of data is too time-consuming, and statically analyzed labeled code is not ideal, as the number of undetected cases, false negatives, is often unacceptably high. G. Bhandari et al. [85] published their article titled CVEfixes. A CVE is a vulnerability registered in some published software. This paper proposes a method to automatically collect vulnerable code and fixed code for all CVEs recorded in the U.S. National Vulnerability Database. Implementing this solution and utilizing vulnerable functions and fixes as data for our models is a significant step in the right direction.

The next step clearly involves utilizing a Model or set of Models that first detect the presence of a vulnerability and then classify it. As evident, the binary classification models outperformed the multi-label classification models. Therefore, if we begin by detecting with binary classification, we can ensure a higher detection rate and then sequentially achieve the same or better CWE classification.

References

[1] T. H. Bureau, "Google paid $12 million as bug bounty; fixed over 2,900 security issues in 2022 - The Hindu." Available at https://www.thehindu.com/sci-tech/technology/google-paid-12-million-as-bug-bounty-fixed-over-2900-security-issues-in-2022/article66552415.ece, 2 2023. Acessed: 2023-02-25. [Cited on page 1]

[2] MITRE, "Common weakness enumeration - Overview." Available at https://cwe.mitre.org/about/index.html, 11 2022. Accessed: 2023-02-04. [Cited on page 2]

[3] MITRE, "Common weakness enumeration - History." Available at https://cwe.mitre.org/about/history.html, 9 2022. Accessed: 2023-02-04. [Cited on page 2]

[4] "R&d centers | mitre." Available at https://www.mitre.org/our-impact/rd-centers, 2023. Accessed: 2023-06-30. [Cited on page 2]

[5] Mend, "Mend.io (formerly whitesource) | improving appsec outcomes." Available at https://www.mend.io/. Accessed: 2023-02-02. [Cited on page 2]

[6] Mend, "Is one programming language more secure than the rest?." Available at https://www.mend.io/resources/blog/is-one-programming-language-more-secure/, 3 2019. Accessed: 2023-02-05. [Cited on page 2]

[7] Mend, "What are the most secure programming languages?." Available at https://www.mend.io/most-secure-programming-languages/. Accessed: 2023-02-02. [Cited on pages xi, 2, and 12]

[8] E. Targett, "We analysed 90,000+ software vulnerabilities: Here's what we learned." Available at https://thestack.technology/analysis-of-cves-in-2022-software-vulnerabilities-cwes-most-dangerous/, 1 2023. Acessed: 2023-01-27. [Cited on pages xi and 3]

[9] M. Braunwarth, "Why are security vulnerabilities increasing? | Netspi Blog." Available at https://www.netspi.com/blog/executive/vulnerability-management/vulnerabilities-and-cybersecurity-investments/, 11 2022. Acessed: 2023-01-27. [Cited on page 3]

[10] GuardRails, "Ai-assisted coding: Risks and rewards | guardrails." Available at https://www.guardrails.io/blog/ai-assisted-coding-a-double-edged-sword/, 5 2023. Accessed: 2023-08-13. [Cited on page 5]

[11] A. M. Turing *et al.*, "On computable numbers, with an application to the Entscheidungsproblem," *J. of Math*, vol. 58, no. 345-363, p. 5, 1936. [Cited on page 9]

[12] N. J. Macias, "Context-dependent functions: Narrowing the realm of turing's halting problem," *arXiv preprint arXiv:1501.03018*, 2015. [Cited on page 9]

[13] D. Erik, "How has static code analysis changed through the years? - NDepend." Available at https://blog.ndepend.com/static-code-analysis-changed-years/. Accessed: 2023-01-28. [Cited on page 10]

[14] A. Kaur and R. Nayyar, "A comparative study of static code analysis tools for vulnerability detection in C/C++ and JAVA source code," *Procedia Computer Science*, vol. 171, pp. 2023–2029, 1 2020. [Cited on page 10]

[15] J. Jang and H. K. Kim, "Fuzzbuilder: Automated building greybox fuzzing environment for C/C++ library," in *Proceedings of the 35th Annual Computer Security Applications Conference*, pp. 627–637, 2019. [Cited on page 10]

[16] R. Baldoni, E. Coppa, D. C. D'elia, C. Demetrescu, and I. Finocchi, "A survey of symbolic execution techniques," *ACM Computing Surveys (CSUR)*, vol. 51, no. 3, pp. 1–39, 2018. [Cited on page 10]

[17] A. Meneely, B. Smith, and L. Williams, "Validating software metrics: A spectrum of philosophies," *ACM Trans. Softw. Eng. Methodol. 21, 4, Article*, vol. 24, pp. 1–2, 2012. [Cited on page 10]

[18] R. Lincke, J. Lundberg, and W. Löwe, "Comparing software metrics tools," in *Proceedings of the 2008 international symposium on Software testing and analysis*, pp. 131–142, 2008. [Cited on page 11]

[19] R. L. Russell, L. Kim, L. H. Hamilton, T. Lazovich, J. A. Harer, O. Ozdemir, P. M. Ellingwood, and M. W. McConley, "Automated vulnerability detection in source code using deep representation learning," in *2018 17th IEEE international conference on machine learning and applications (ICMLA)*, pp. 757–762, IEEE, 2018. [Cited on pages 11, 33, 34, 37, 40, 43, 46, 54, 61, 67, 68, 100, 101, 102, 105, and 106]

[20] D. M. Ritchie, "The development of the C programming language," in *History of Programming languages—II*, pp. 671–698, Prentice Hall, 1996. [Cited on page 11]

[21] P. Tyagi and C. M. Sharma, "A tribute to C programming language: History and modern applications," *International Journal of Computer Applications*, vol. 975, p. 8887, 2014. [Cited on pages 11 and 12]

[22] B. Stroustrup, "An overview of the C++ programming language," *Handbook of object technology*, p. 72, 1999. [Cited on page 12]

[23] J. Regehr, "A guide to undefined behavior in c and c++, part 1," 2010. [Cited on page 12]

[24] J. D. Pereira, N. Ivaki, and M. Vieira, "Characterizing buffer overflow vulnerabilities in large C/C++ projects," *IEEE Access*, vol. 9, pp. 142879–142892, 2021. [Cited on pages xvii, 12, and 13]

[25] Y. Younan, W. Joosen, F. Piessens, and H. den Eynden, "Security of memory allocators for C and C++," tech. rep., Technical Report CW 419, Department of Computer Science, Katholieke ..., 2005. [Cited on page 13]

[26] A. M. Turing, *Computing Machinery and Intelligence*, vol. LIX. Springer, 10 1950. [Cited on page 14]

[27] R. Anyoha, "The history of artificial intelligence." Available at `https://sitn.hms.harvard.edu/flash/2017/history-artificial-intelligence/`. Accessed: 2023-02-12. [Cited on page 14]

[28] Wikipedia, "Deep blue (chess computer) - Wikipedia, the free encyclopedia." Available at `https://en.wikipedia.org/wiki/Deep_Blue_(chess_computer)`, 2023. Accessed: 2023-02-12. [Cited on pages xi and 14]

[29] S. C. Society, "Simplifying the difference: Machine learning vs deep learning - Singapore Computer Society." Available at `https://www.scs.org.sg/articles/machine-learning-vs-deep-learning`, 2021. Accessed: 2023-02-12. [Cited on pages xi and 15]

[30] C. Janiesch, P. Zschech, and K. Heinrich, "Machine learning and deep learning," *Electronic Markets*, vol. 31, pp. 685–695, 9 2021. [Cited on pages 15 and 16]

[31] P. P. Shinde and S. Shah, "A review of machine learning and deep learning applications," in *2018 Fourth international conference on computing communication control and automation (ICCUBEA)*, pp. 1–6, Institute of Electrical and Electronics Engineers Inc., 7 2018. [Cited on pages 15 and 16]

[32] V. Nasteski, "An overview of the supervised machine learning methods," *Horizons. b*, vol. 4, pp. 51–62, 12 2017. [Cited on page 16]

[33] Y. Lecun, Y. Bengio, and G. Hinton, "Deep learning," *Nature 2015 521:7553*, vol. 521, pp. 436–444, 5 2015. [Cited on pages 16 and 17]

[34] IBM, "What are neural networks? | IBM." Available at `https://www.ibm.com/topics/neural-networks`, 9 2022. Accessed: 2023-02-17. [Cited on pages 17 and 19]

[35] TIBICO, "What is a neural network? | TIBCO Software." Available at `https://www.tibco.com/reference-center/what-is-a-neural-network`. Accessed: 2023-02-17. [Cited on pages xi and 17]

[36] K. Melcher, "A friendly introduction to [deep] neural networks | KNIME." Available at `https://www.knime.com/blog/a-friendly-introduction-to-deep-neural-networks`, 8 2021. Accessed: 2023-02-17. [Cited on pages xi and 17]

[37] V. Zhou, "Machine learning for beginners: An introduction to neural networks | Towards Data Science." Available at https://towardsdatascience.com/machine-learning-for-beginners-an-introduction-to-neural-networks-d49f22d238f9, 3 2019. Accessed: 2023-02-19. [Cited on page 20]

[38] A. Grupta, "A comprehensive guide on optimizers in deep learning." Available at `https://www.analyticsvidhya.com/blog/2021/10/a-comprehensive-guide-on-deep-learning-optimizers/#What_Are_Optimizers_in_Deep_Learning?`, 10 2023. Accessed: 2023-03-06. [Cited on page 21]

[39] S. Ding, H. Li, C. Su, J. Yu, and F. Jin, "Evolutionary artificial neural networks: a review," *Artificial Intelligence Review*, vol. 39, 2011. [Cited on page 21]

[40] M. Hagan and H. Demuth, "Neural networks for control," in *Proceedings of the 1999 American Control Conference (Cat. No. 99CH36251)*, vol. 3, pp. 1642–1656 vol.3, 1999. [Cited on page 21]

[41] A.-N. Sharkawy, "Principle of neural network and its main types: Review," *Journal of Advances in Applied & Computational Mathematics*, vol. 7, pp. 8–19, 8 2020. [Cited on pages xiv, 22, and 120]

[42] DeepAI, "Feed forward neural network definition." Available at `https://deepai.org/machine-learning-glossary-and-terms/feed-forward-neural-network`. Accessed: 2023-02-22. [Cited on pages xi and 22]

[43] M. W. Gardner and S. R. Dorling, "Artificial neural networks (the multilayer perceptron)—a review of applications in the atmospheric sciences," *Atmospheric Environment*, vol. 32, pp. 2627–2636, 8 1998. [Cited on pages xi and 23]

[44] Great Learning, "Types of neural networks and definition of neural network." Available at https://www.mygreatlearning.com/blog/types-of-neural-networks/, 11 2022. Accessed: 2023-02-22. [Cited on pages xi, xiv, 23, 24, 27, and 120]

[45] Z. Li, F. Liu, W. Yang, S. Peng, and J. Zhou, "A survey of convolutional neural networks: analysis, applications, and prospects," *IEEE transactions on neural networks and learning systems*, 2021. [Cited on page 23]

[46] Trimble, "Using deep learning models / convolutional neural networks." Available at https://docs.ecognition.com/eCognition_documentation/User%20Guide%20Developer/8%20Classification%20-%20Deep%20Learning.htm. Accessed: 2023-02-23. [Cited on pages xi and 24]

[47] G. A. Montazer, D. Giveki, M. Karami, and H. Rastegar, "Radial basis function neural networks: A review," *Computer Reviews Journal*, vol. 1, pp. 2581–6640, 2018. [Cited on page 24]

[48] C.-C. Lee, P.-C. Chung, J.-R. Tsai, C.-I. Chang, and S. Member, "Robust radial basis function neural networks," *IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics)*, vol. 29, pp. 674–685, 1999. [Cited on page 24]

[49] M. Chris, "Radial basis function network (rbfn) tutorial." Available at https://mccormickml.com/2013/08/15/radial-basis-function-network-rbfn-tutorial/, 8 2013. Accessed: 2023-02-23. [Cited on pages xi and 25]

[50] L. Medsker, D. L. Jain, and B. R. L. N. Y. Washington, "Recurrent neural networks," *Design and Applications*, vol. 5, 2001. [Cited on page 25]

[51] C. H. Bennett, R. A. Dellana, T. Xiao, B. Feinberg, S. Agarwal, S. G. Cardwell, M. J. Marinella, W. M. Severa, and J. B. Aimone, "Evaluating complexity and resilience trade-offs in emerging memory inference machines.," in *Proceedings of the Neuro-inspired Computational Elements Workshop*, p. 2, 3 2021. [Cited on pages xi and 25]

[52] M. S. Aliaa Rassem, Mohammed El-Beltagy, "Cross-country skiing gears classification using deep learning," *arXiv preprint arXiv:1706.08924*, p. 6, 6 2017. [Cited on pages xi and 26]

[53] J. Chung, C. Gulcehre, and K. Cho, "Empirical evaluation of gated recurrent neural networks on sequence modeling," *arXiv preprint arXiv:1412.3555*, 2014. [Cited on page 26]

[54] I. Bibi, A. Akhunzada, J. Malik, J. Iqbal, A. Mussaddiq, and S. Kim, "A dynamic dl-driven architecture to combat sophisticated android malware," *IEEE Access*, vol. 8, pp. 129600–129612, 2020. [Cited on pages xi and 27]

[55] G. Neubig, "Neural machine translation and sequence-to-sequence models: A tutorial," *arXiv preprint arXiv:1703.01619*, 2017. [Cited on page 27]

[56] J. F. Qiao, X. Meng, W. J. Li, and B. M. Wilamowski, "A novel modular rbf neural network based on a brain-like partition method," *Neural Computing and Applications*, vol. 32, pp. 899–911, 2 2020. [Cited on pages xi and 28]

[57] M. Grandini, E. Bagli, and G. Visani, "Metrics for multi-class classification: an overview," *arXiv preprint arXiv:2008.05756*, 8 2020. [Cited on pages 29, 30, and 31]

[58] M. Navim and R. Pankaja, "Performance analysis of text classification algorithms using confusion matrix," *Int. J. Eng. Tech. Res. IJETR*, vol. 6, pp. 75–78, 2016. [Cited on page 29]

[59] D. Chicco and G. Jurman, "The advantages of the matthews correlation coefficient (mcc) over f1 score and accuracy in binary classification evaluation," *BMC Genomics*, vol. 21, pp. 1–13, 1 2020. [Cited on page 31]

[60] S. Narkhede, "Understanding auc-roc curve," *Towards Data Science*, vol. 26, pp. 220–227, 2018. [Cited on page 32]

[61] J. Karlberg and M. Axén, "Binary classification for predicting customer churn," Master's thesis, Umeå University, 2020. [Cited on pages xi and 32]

[62] C. Project, "Clang: a c language family frontend for llvm." Available at `https://clang.llvm.org/docs/ClangTools.html`. Accessed: 2023-03-29. [Cited on page 33]

[63] Z. Bilgin, M. A. Ersoy, E. U. Soykan, E. Tomur, P. Comak, and L. Karacay, "Vulnerability prediction from source code using machine learning," *IEEE Access*, vol. 8, pp. 150672–150684, 2020. [Cited on pages 34, 40, 100, 101, 102, and 106]

[64] Z. Li, D. Zou, J. Tang, Z. Zhang, M. Sun, and H. Jin, "A comparative study of deep learning-based vulnerability detection system," *IEEE Access*, vol. 7, pp. 103184–103197, 2019. [Cited on pages 34, 35, 40, and 69]

[65] Z. Li, D. Zou, S. Xu, H. Jin, Y. Zhu, and Z. Chen, "Sysevr: A framework for using deep learning to detect software vulnerabilities," *IEEE Transactions on Dependable and Secure Computing*, vol. 19, pp. 2244–2258, 2022. [Cited on page 34]

[66] Joern, "Joern: The bug hunter's workbench." Available at `https://joern.io/`. Accessed: 2023-03-29. [Cited on pages 34 and 36]

[67] D. Zou, S. Wang, S. Xu, Z. Li, and H. Jin, "Ieee transactions on dependable and secure computing 1 μvuldeepecker: A deep learning-based system for multiclass vulnerability detection," *IEEE Transactions on Dependable and Secure Computing*, vol. 18, no. 5, pp. 2224–2236, 2019. [Cited on pages 34, 35, 38, 40, and 71]

[68] "Database of "μVulDeePecker"." Available at `https://github.com/muVulDeePecker/muVulDeePecker`, 2019. Accessed: 2023-03-26. [Cited on pages 34 and 38]

[69] G. Tang, L. Yang, S. Ren, L. Meng, F. Yang, and H. Wang, "An automatic source code vulnerability detection approach based on KELM," *Security and Communication Networks*, vol. 2021, pp. 1–12, 2021. [Cited on pages 35 and 40]

[70] Y. Zhou, S. Liu, J. Siow, X. Du, and Y. Liu, "Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks," *Advances in neural information processing systems*, vol. 32, 2019. [Cited on pages xiii, 36, 39, and 40]

[71] "Database of "Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks"." Available at `https://sites.google.com/view/devign`, 2019. Accessed: 2023-03-26. [Cited on pages 36 and 39]

[72] J. Wang, H. Xiao, S. Zhong, and Y. Xiao, "DeepVulSeeker: A novel vulnerability identification framework via code graph structure and pre-training mechanism," *arXiv preprint arXiv:2211.13097*, 11 2022. [Cited on pages 36 and 40]

[73] M. T. B. Nazim, M. J. H. Faruk, H. Shahriar, M. A. Khan, M. Masum, N. Sakib, and F. Wu, "Systematic analysis of deep learning model for vulnerable code detection," in *2022 IEEE 46th Annual Computers, Software, and Applications Conference (COMPSAC)*, pp. 1768–1773, Institute of Electrical and Electronics Engineers Inc., 2022. [Cited on pages 37 and 38]

[74] L. Kim and R. Russell, "Osf | Draper VDISC dataset - vulnerability detection in source code wiki." Available at `https://osf.io/d45bw/wiki/home/`, 2021. Accessed: 25-03-2023. [Cited on pages xiii, 37, and 38]

[75] N. I. of Standards and Technology, "Nist: Software assurance reference dataset." Available at `https://samate.nist.gov/SARD/`. Accessed: 2023-03-26. [Cited on page 38]

[76] NDSS, "Database of "VulDeePecker: A deep learning-based system for vulnerability detection"." Available at `https://github.com/CGCL-codes/VulDeePecker`, 2018. Accessed: 2023-03-26. [Cited on page 38]

[77] S. Chakraborty, R. Krishna, Y. Ding, and B. Ray, "Deep learning based vulnerability detection: Are we there yet?," *IEEE Transactions on Software Engineering*, vol. 48, pp. 3280–3296, 9 2022. [Cited on page 39]

[78] "Database of "VulDetProject/ReVeal"." Available at `https://github.com/VulDetProject/ReVeal`, 2021. Accessed: 2023-03-26. [Cited on page 39]

[79] MITRE, "Cwe-469: Use of pointer subtraction to determine size (4.12)," 2016. Accessed: 2023-04-05. [Cited on pages xvii and 45]

[80] B. Chernis and R. Verma, "Machine learning methods for software vulnerability detection," *IWSPA 2018 - Proceedings of the 4th ACM International Workshop on Security and Privacy Analytics, Co-located with CODASPY 2018*, vol. 2018-January, pp. 31–39, 3 2018. [Cited on page 52]

[81] P. Jeffrey, S. Richard, and D. M. Christopher, "Glove: Global vectors for word representation." Available at `https://nlp.stanford.edu/projects/glove/`. Accessed: 2023-05-29. [Cited on page 53]

[82] T. Dias, A. Batista, E. Maia, and I. Praça, "Testlab: An intelligent automated software testing framework," *arXiv preprint arXiv:2306.03602*, 6 2023. [Cited on page 108]

[83] N. Risse and M. Böhme, "Limits of machine learning for automatic vulnerability detection," *Proceedings of ACM Conference (Conference'17)*, vol. 1, 6 2023. [Cited on page 109]

[84] X. Li, L. Wang, Y. Xin, Y. Yang, Q. Tang, and Y. Chen, "Automated software vulnerability detection based on hybrid neural network," *Applied Sciences 2021, Vol. 11, Page 3201*, vol. 11, p. 3201, 4 2021. [Cited on page 110]

[85] G. Bhandari, A. Naseer, and L. Moonen, "Cvefixes: Automated collection of vulnerabilities and their fixes from open-source software," *PROMISE 2021 - Proceedings of the 17th International Conference on Predictive Models and Data Analytics in Software Engineering, co-located with ESEC/FSE 2021*, pp. 30–39, 8 2021. [Cited on page 110]

Appendix A

Advantages and Disadvantages of Neural Networks

Table A.1: Advantages and Disadvantages of the most common Neural Networks [44] [41].

Neural Network	Advantages	Disadvantages
Perceptron	• Implement logic gates like AND, OR and NAND	• Can only learn linearly separable problems such as boolean AND problem • For non-linear problems such as the boolean XOR problem, it does not work
FNN	• Less complex, easy to design and maintain • Fast and speedy • Highly responsive to noisy data	• Cannot be used for deep learning • Gas turbine engine is sized for peak power conditions
MLP	• Used for Deep Learning	• Comparatively complex to design and maintain and slow
CNN	• Used for deep learning with few parameters • Less parameters to learn as compared to fully connected layer	• Comparatively complex to design and maintain • Comparatively slow
RBFN	• Can approximate nonlinear functions with high accuracy • Can performs faster • Can handle noisy data	• Its structure and training algorithm present various issues that prevent it from effectively modeling a strongly nonlinear system. • The complexity grows in proportion to the quantity of neurons present in the hidden layer.
RNN	• Ability to model sequential data where each sample can be assumed to depend on historical ones • Used with convolution layers to extend the pixel effectiveness • The size is significantly compact compared with feedforward networks	• Gradient vanishing and exploding gradient problem • Processing long sequential data using ReLU or tanh as an activation function can be challenging.
LSTM	• Are designed to remember information over long periods of time • Overcome vanishing gradients and exploding gradients problems • Can be pre-trained on a large dataset, and the learned features can be transferred to a different task or dataset	• Take longer to train • Require more memory to train • Are easy to overfit

Appendix B

Summary of CNN Models

B.1 Detection of CWE-119 Vulnerabilities

Table B.1: Summary of the CNN model to detect CWE-119 vulnerabilities.

Layer Name	Output Shape	Param. Count	Info
Embedding	[1061, 300]	46461300	output_dim=300
Conv1D	[1057, 1024]	1537024	filters = 1024 , kernel_size = 5
GlobalMaxPooling	[1024]	0	
Flatten	[1024]	0	
Dense 1	[512]	524800	units = 512
Dropout 1	[512]	0	rate = 0.2
Dense 2	[256]	131328	units = 256
Dropout 2	[256]	0	rate = 0.2
Dense 3	[1]	257	units = 1

B.2 Detection of CWE-120 Vulnerabilities

Table B.2: Summary of the CNN model to detect CWE-120 vulnerabilities.

Layer Name	Output Shape	Param. Count	Info
Embedding	[2878, 300]	72970200	output_dim=300
Conv1D	[2878, 1024]	1537024	filters = 1024 , kenel_size = 5
GlobalMaxPooling	[1024]	0	
Flatten	[1024]	0	
Dense 1	[512]	524800	units = 512
Dropout 1	[512]	0	rate = 0.2
Dense 2	[256]	131328	units = 256
Dropout 2	[256]	0	rate = 0.2
Dense 3	[1]	257	units = 1

B.3 Detection of CWE-469 Vulnerabilities

Table B.3: Summary of the CNN model to detect CWE-469 vulnerabilities.

Layer Name	Output Shape	Param. Count	Info
Embedding	[1176, 300]	9663600	output_dim=300
Conv1D	[1172, 1024]	1537024	filters = 1024 , kenel_size = 5
GlobalMaxPooling	[1024]	0	
Flatten	[1024]	0	
Dense 1	[512]	524800	units = 512
Dropout 1	[512]	0	rate = 0.2
Dense 2	[256]	131328	units = 256
Dropout 2	[256]	0	rate = 0.2
Dense 3	[1]	257	units = 1

B.4 Detection of CWE-476 Vulnerabilities

Table B.4: Summary of the CNN model to detect CWE-476 vulnerabilities.

Layer Name	Output Shape	Param. Count	Info
Embedding	[689, 300]	28737300	output_dim=300
Conv1D	[685, 1024]	1537024	filters = 1024 , kenel_size = 5
GlobalMaxPooling	[1024]	0	
Flatten	[1024]	0	
Dense 1	[512]	524800	units = 512
Dropout 1	[512]	0	rate = 0.2
Dense 2	[256]	131328	units = 256
Dropout 2	[256]	0	rate = 0.2
Dense 3	[1]	257	units = 1

B.5 Detection of CWE-Others Vulnerabilities

Table B.5: Summary of the CNN model to detect CWE-Others vulnerabilities.

Layer Name	Output Shape	Param. Count	Info
Embedding	[2878, 300]	61234500	output_dim=300
Conv1D	[2874, 1024]	1537024	filters = 1024 , kenel_size = 5
GlobalMaxPooling	[1024]	0	
Flatten	[1024]	0	
Dense 1	[512]	524800	units = 512
Dropout 1	[512]	0	rate = 0.2
Dense 2	[256]	131328	units = 256
Dropout 2	[256]	0	rate = 0.2
Dense 3	[1]	257	units = 1

B.6 Detection of Any Vulnerability (CWE-Combined)

Table B.6: Summary of the CNN model to detect CWE-Combined vulnerabilities.

Layer Name	Output Shape	Param. Count	Info
Embedding	[10225, 300]	106941000	output_dim=300
Conv1D	[10221, 1024]	1537024	filters = 1024 , kenel_size = 5
GlobalMaxPooling	[1024]	0	
Flatten	[1024]	0	
Dense 1	[512]	524800	units = 512
Dropout 1	[512]	0	rate = 0.2
Dense 2	[256]	131328	units = 256
Dropout 2	[256]	0	rate = 0.2
Dense 3	[1]	257	units = 1

B.7 Multi-Label Classification

Table B.7: Summary of the CNN model to Identify and Classify All vulnerabilities.

Layer Name	Output Shape	Param. Count	Info
Embedding	[10225, 300]	106941000	output_dim=300
Conv1D	[10221, 1024]	1537024	filters = 1024 , kenel_size = 5
GlobalMaxPooling	[1024]	0	
Flatten	[1024]	0	
Dense 1	[512]	524800	units = 512
Dropout 1	[512]	0	rate = 0.2
Dense 2	[256]	131328	units = 256
Dropout 2	[256]	0	rate = 0.2
Dense 3	[1]	1285	units = 1

Appendix C

Summary of GRU Models

C.1 Detection of CWE-119 Vulnerabilities

Table C.1: Summary of the GRU model to detect CWE-119 Vulnerabilities.

Layer Name	Output Shape	Param. Count	Info
Embedding	[1061, 300]	46461300	output_dim=300
SpartialDropout	[1061, 300]	0	rate = 0.2
GRU 1	[1061, 512]	1250304	units = 1024 (Return Sequences)
GRU 2	[256]	591360	units = 512
Dropout 2	[256]	0	rate = 0.1
Dense 3	[1]	257	units = 1

C.2 Detection of CWE-120 Vulnerabilities

Table C.2: Summary of the GRU model to detect CWE-120 Vulnerabilities.

Layer Name	Output Shape	Param. Count	Info
Embedding	[2878, 300]	72970200	output_dim=300
SpartialDropout	[2878, 300]	0	rate = 0.2
GRU 1	[2878, 512]	1250304	units = 1024 (Return Sequences)
GRU 2	[256]	591360	units = 512
Dropout 2	[256]	0	rate = 0.1
Dense 3	[1]	257	units = 1

C.3 Detection of CWE-469 Vulnerabilities

Table C.3: Summary of the GRU model to detect CWE-469 Vulnerabilities.

Layer Name	Output Shape	Param. Count	Info
Embedding	[1176, 300]	9663600	output_dim=300
SpartialDropout	[1176, 300]	0	rate = 0.2
GRU 1	[1176, 512]	1250304	units = 1024 (Return Sequences)
GRU 2	[256]	591360	units = 512
Dropout 2	[256]	0	rate = 0.1
Dense 3	[1]	257	units = 1

C.4 Detection of CWE-476 Vulnerabilities

Table C.4: Summary of the GRU model to detect CWE-476 Vulnerabilities.

Layer Name	Output Shape	Param. Count	Info
Embedding	[689, 300]	28737300	output_dim=300
SpartialDropout	[689, 300]	0	rate = 0.2
GRU 1	[689, 512]	1250304	units = 1024 (Return Sequences)
GRU 2	[256]	591360	units = 512
Dropout 2	[256]	0	rate = 0.1
Dense 3	[1]	257	units = 1

C.5 Detection of CWE-Others Vulnerabilities

Table C.5: Summary of the GRU model to detect CWE-Others Vulnerabilities.

Layer Name	Output Shape	Param. Count	Info
Embedding	[2878, 300]	61234500	output_dim=300
SpartialDropout	[2878, 300]	0	rate = 0.2
GRU 1	[2878, 512]	1250304	units = 1024 (Return Sequences)
GRU 2	[256]	591360	units = 512
Dropout 2	[256]	0	rate = 0.1
Dense 3	[1]	257	units = 1

C.6 Detection of Any Vulnerability (CWE-Combined)

Table C.6: Summary of the GRU model to detect CWE-Combined Vulnerabilities.

Layer Name	Output Shape	Param. Count	Info
Embedding	[10225, 300]	106941000	output_dim=300
SpartialDropout	[10225, 300]	0	rate = 0.2
GRU 1	[10225, 512]	1250304	units = 1024 (Return Sequences)
GRU 2	[256]	591360	units = 512
Dropout 2	[256]	0	rate = 0.1
Dense 3	[1]	257	units = 1

C.7 Multi-Label Classification

Table C.7: Summary of the CNN model to Identify and Classify All Vulnerabilities.

Layer Name	Output Shape	Param. Count	Info
Embedding	[10225, 300]	106941000	output_dim=300
SpartialDropout	[10225, 300]	0	rate = 0.2
GRU 1	[10225, 512]	1250304	units = 1024 (Return Sequences)
GRU 2	[256]	591360	units = 512
Dropout 2	[256]	0	rate = 0.1
Dense 3	[5]	1285	units = 1

Appendix D

Summary of BLSTM Models

D.1 Detection of CWE-119 Vulnerabilities

Table D.1: Summary of the BLSTM model to detect CWE-119 Vulnerabilities.

Layer Name	Output Shape	Param. Count	Info
Embedding	[1061, 300]	46461300	output_dim=300
Bidirectional BLSTM	[1061, 1024]	3330048	units = 512
Global Max Pooling	[1024]	0	Default
Batch Normalization	[1024]	4096	Default
Dropout 1	[1024]	0	rate = 0.2
Dense 1	[1061]	1087525	units = 1061
Dropout 2	[1061]	0	rate = 0.2
Dense 2	[805]	854910	units = 256
Dropout 3	[805]	0	rate = 0.2
Dense 3	[1]	806	units = 1

D.2 Detection of CWE-120 Vulnerabilities

Table D.2: Summary of the BLSTM model to detect CWE-120 Vulnerabilities.

Layer Name	Output Shape	Param. Count	Info
Embedding	[2878, 300]	72970200	output_dim=300
Bidirectional BLSTM	[2878, 1024]	3330048	units = 512
Global Max Pooling	[1024]	0	Default
Batch Normalization	[1024]	4096	Default
Dropout 1	[1024]	0	rate = 0.2
Dense 1	[2878]	2949950	units = 1061
Dropout 2	[2878]	0	rate = 0.2
Dense 2	[2622]	7548738	units = 256
Dropout 3	[2622]	0	rate = 0.2
Dense 3	[1]	2623	units = 1

D.3 Detection of CWE-469 Vulnerabilities

Table D.3: Summary of the BLSTM model to detect CWE-469 Vulnerabilities.

Layer Name	Output Shape	Param. Count	Info
Embedding	[1176, 300]	9663600	output_dim=300
Bidirectional BLSTM	[1176, 1024]	3330048	units = 512
Global Max Pooling	[1024]	0	Default
Batch Normalization	[1024]	4096	Default
Dropout 1	[1024]	0	rate = 0.2
Dense 1	[1176]	1205400	units = 1061
Dropout 2	[1176]	0	rate = 0.2
Dense 2	[920]	1082840	units = 256
Dropout 3	[920]	0	rate = 0.2
Dense 3	[1]	921	units = 1

D.4 Detection of CWE-476 Vulnerabilities

Table D.4: Summary of the BLSTM model to detect CWE-476 Vulnerabilities.

Layer Name	Output Shape	Param. Count	Info
Embedding	[689, 300]	28737300	output_dim=300
Bidirectional BLSTM	[689, 1024]	3330048	units = 512
Global Max Pooling	[1024]	0	Default
Batch Normalization	[1024]	4096	Default
Dropout 1	[1024]	0	rate = 0.2
Dense 1	[689]	706225	units = 1061
Dropout 2	[689]	0	rate = 0.2
Dense 2	[433]	298770	units = 256
Dropout 3	[433]	0	rate = 0.2
Dense 3	[1]	434	units = 1

D.5 Detection of CWE-Others Vulnerabilities

Table D.5: Summary of the BLSTM model to detect CWE-Other Vulnerabilities.

Layer Name	Output Shape	Param. Count	Info
Embedding	[2878, 300]	61234500	output_dim=300
Bidirectional BLSTM	[2878, 1024]	3330048	units = 512
Global Max Pooling	[1024]	0	Default
Batch Normalization	[1024]	4096	Default
Dropout 1	[1024]	0	rate = 0.2
Dense 1	[2878]	2949950	units = 1061
Dropout 2	[2878]	0	rate = 0.2
Dense 2	[2622]	7548738	units = 256
Dropout 3	[2622]	0	rate = 0.2
Dense 3	[1]	2623	units = 1

D.6 Detection of Any Vulnerability (CWE-Combined)

Table D.6: Summary of the BLSTM model to detect CWE-Combined Vulnerabilities.

Layer Name	Output Shape	Param. Count	Info
Embedding	[10225, 300]	106941000	output_dim=300
Bidirectional BLSTM	[10225, 1024]	3330048	units = 512
Global Max Pooling	[1024]	0	Default
Batch Normalization	[1024]	4096	Default
Dropout 1	[1024]	0	rate = 0.2
Dense 1	[10225]	10480625	units = 1061
Dropout 2	[10225]	0	rate = 0.2
Dense 2	[9969]	101942994	units = 256
Dropout 3	[9969]	0	rate = 0.2
Dense 3	[1]	2623	units = 1

D.7 Multi-Label Classification

Table D.7: Summary of the BLSTM model to detect All Vulnerabilities.

Layer Name	Output Shape	Param. Count	Info
Embedding	[10225, 300]	106941000	output_dim=300
Bidirectional BLSTM	[10225, 1024]	3330048	units = 512
Global Max Pooling	[1024]	0	Default
Batch Normalization	[1024]	4096	Default
Dropout 1	[1024]	0	rate = 0.2
Dense 1	[10225]	10480625	units = 1061
Dropout 2	[10225]	0	rate = 0.2
Dense 2	[9969]	101942994	units = 256
Dropout 3	[9969]	0	rate = 0.2
Dense 3	[5]	49850	units = 5

Appendix E

Results of Generic Algorithms on Binary Classification

E.1 Detection of CWE-119 Vulnerabilities

Table E.1: Results of the Generic Algorithms for CWE-119 Vulnerability Detection

Model	Accuracy	Precision	Recall	F1-Score	ROC-AUC
Random Forest	0.6793	0.6433	0.7895	0.7089	0.6806
Lasso	0.561	0.5353	**0.8529**	0.6578	0.564
ElasticNET	0.5613	0.5355	0.8536	0.6581	0.5644
Naive Bayes	0.621	0.6697	0.4615	0.5464	0.6193
Gradient Boosted Trees	**0.7563**	**0.7407**	0.7805	**0.7601**	**0.7565**
SVM	0.5366	0.5304	0.5506	0.5403	0.5367
Logistic Regression	0.6053	0.6386	0.4655	0.5385	0.6038
Linear Discriminant Analysis	0.5604	0.5349	0.8525	0.6574	0.5635
KNeighborsClassifier	0.5735	0.5645	0.6029	0.5831	0.5738
Decision Tree Classifier	0.6093	0.598	0.6414	0.619	0.6097
Gaussian NB	0.5138	0.6898	0.0311	0.0596	0.5087

E.1.1 Tunning

Table E.2: Results of the Generic Algorithms for CWE-119 Vulnerability Detection with Tunning.

Algorithm	Accuracy	Precision	Recall	F1-Score	ROC-AUC	MCC	Best Hyperparameters
Random Forest	**0.7224**	**0.7032**	**0.7703**	**0.7352**	**0.7225**	**0.4468**	n_estimators = 768 min_samples_leaf = 1 max_depth = 64
Decision Tree	0.6156	0.6191	0.6023	0.6156	0.6156	0.2313	max_depth = 16384 min_samples_leaf = 10 splitter = best

E.2 Detection of CWE-120 Vulnerabilities

Table E.3: Results of the Generic Algorithms for CWE-120 Vulnerability Detection

Model	Accuracy	Precision	Recall	F1-Score	ROC-AUC
Random Forest	0.6722	0.6494	0.755	0.6982	0.6718
Lasso	0.5661	0.5445	**0.8345**	0.659	0.565
ElasticNET	0.5661	0.5445	**0.8345**	0.659	0.565
Naive Bayes	0.603	0.6605	0.4315	0.522	0.6038
Gradient Boosted Trees	**0.7093**	**0.6930**	0.7564	**0.7234**	**0.7091**
SVM	0.5332	0.5633	0.3145	0.4037	0.5342
Logistic Regression	0.5844	0.6447	0.3847	0.4818	0.5853
Linear Discriminant Analysis	0.566	0.5444	0.8335	0.6587	0.5649
KNeighborsClassifier	0.5679	0.5686	0.5798	0.5741	0.5679
Decision Tree Classifier	0.5912	0.589	0.6155	0.602	0.5911
Gaussian NB	0.498	0.5455	0.0037	0.0074	0.5003

E.2.1 Tunning

Table E.4: Results of the Generic Algorithms for CWE-120 Vulnerability Detection with Tunning.

Algorithm	Accuracy	Precision	Recall	F1-Score	ROC-AUC	MCC	Best Hyperparameters
Random Forest	**0.7103**	**0.6933**	**0.7565**	**0.7235**	**0.7102**	**0.4223**	n_estimators = 128 min_samples_leaf = 1 max_depth = 2048
Decision Tree	0.6044	0.6079	0.5927	0.6002	0.6044	0.2089	max_depth = 8192 min_samples_leaf = 10 splitter = best

E.3 Detection of CWE-469 Vulnerabilities

Table E.5: Results of the Generic Algorithms for CWE-469 Vulnerability Detection

Model	Accuracy	Precision	Recall	F1-Score	ROC-AUC
Random Forest	**0.7219**	0.6862	0.8462	**0.7578**	**0.7183**
Lasso	0.5667	0.553	0.8204	0.6607	0.5592
ElasticNET	0.5676	0.5535	0.8241	0.6622	0.56
Naive Bayes	0.6457	**0.7234**	0.5037	0.5939	0.6499
Gradient Boosted Trees	0.7066	0.669	0.85	0.7488	0.7025
SVM	0.4914	0.5047	0.5889	0.5436	0.4886
Logistic Regression	0.6143	0.688	0.4574	0.5495	0.6189
Linear Discriminant Analysis	0.5638	0.5515	0.813	0.6572	0.5565
KNeighborsClassifier	0.5771	0.5795	0.6481	0.6119	0.5751
Decision Tree Classifier	0.5762	0.578	0.6519	0.6127	0.574
Gaussian NB	0.5123	0.5133	**0.9944**	0.6771	0.4982

E.3.1 Tunning

Table E.6: Results of the Generic Algorithms for CWE-469 Vulnerability Detection with Tunning.

Algorithm	Accuracy	Precision	Recall	F1-Score	ROC-AUC	MCC	Best Hyperparameters
Random Forest	**0.7511**	**0.7444**	**0.7796**	**0.7616**	**0.7505**	**0.5021**	n_estimators = 32 min_samples_leaf = 1 max_depth = 12288
Decision Tree	0.6412	0.6463	0.6575	0.6519	0.6416	0.2833	max_depth = 1024 min_samples_leaf = 1 splitter = best

E.4 Detection of CWE-476 Vulnerabilities

Table E.7: Results of the Generic Algorithms for CWE-476 Vulnerability Detection

Model	Accuracy	Precision	Recall	F1-Score	ROC-AUC
Random Forest	0.6222	0.61	0.6503	0.6295	0.6225
Lasso	0.4971	0.4941	0.7852	0.6065	0.5007
ElasticNET	0.5676	0.5535	0.8241	**0.6622**	0.5601
Naive Bayes	0.5969	**0.6380**	0.4237	0.5093	0.5948
Gradient Boosted Trees	**0.6284**	0.613	0.67	0.6402	**0.6289**
SVM	0.5062	0.4998	0.6001	0.5454	0.5074
Logistic Regression	0.5702	0.6159	0.3438	0.4412	0.5674
Linear Discriminant Analysis	0.4988	0.4951	**0.7864**	0.6077	0.5024
KNeighborsClassifier	0.5604	0.5453	0.6583	0.5965	0.5616
Decision Tree Classifier	0.5622	0.5534	0.5854	0.5689	0.5625
Gaussian NB	0.5095	0.5824	0.0221	0.0427	0.5033

E.4.1 Tunning

Table E.8: Results of the Generic Algorithms for CWE-476 Vulnerability Detection with Tunning.

Algorithm	Accuracy	Precision	Recall	F1-Score	ROC-AUC	MCC	Best Hyperparameters
Random Forest	**0.7533**	**0.7211**	0.8218	**0.7682**	**0.7537**	**0.5120**	n_estimators = 128 min_samples_leaf = 1 max_depth = 2048
Decision Tree	0.6870	0.6184	**0.967**6	0.7545	0.6884	0.4535	max_depth = 32 min_samples_leaf = 10 splitter = best

E.5 Detection of CWE-Others Vulnerabilities

Table E.9: Results of the Generic Algorithms for CWE-Others Vulnerability Detection

Model	Accuracy	Precision	Recall	F1-Score	ROC-AUC
Random Forest	0.6613	0.6351	0.7542	0.6896	0.6615
Lasso	0.5557	0.5354	**0.8280**	0.6503	0.5564
ElasticNET	0.5557	0.5354	0.8276	0.6501	0.5563
Naive Bayes	0.5980	0.6500	0.4206	0.5107	0.5976
Gradient Boosted Trees	**0.6810**	**0.6619**	0.7372	**0.6975**	**0.6811**
SVM	0.5311	0.5317	0.5039	0.5174	0.5311
Logistic Regression	0.5816	0.6466	0.3557	0.4589	0.5811
Linear Discriminant Analysis	0.5550	0.5349	0.8263	0.6494	0.5556
KNeighborsClassifier	0.5579	0.5527	0.5966	0.5738	0.5580
Decision Tree Classifier	0.5881	0.5796	0.6346	0.6059	0.5882
Gaussian NB	0.5009	0.4	0.0008	0.0017	0.4997

E.5.1 Tunning

Table E.10: Results of the Generic Algorithms for CWE-Other Vulnerability Detection with Tunning.

Algorithm	Accuracy	Precision	Recall	F1-Score	ROC-AUC	MCC	Best Hyperparameters
Random Forest	**0.6928**	**0.6775**	**0.7326**	**0.7040**	**0.6930**	**0.3871**	n_estimators = 2048 min_samples_leaf = 1 max_depth = 32
Decision Tree	0.5911	0.5928	0.5740	0.5832	0.5910	0.1822	max_depth = 2048 min_samples_leaf = 10 splitter = best

E.6 Detection of CWE-Combined Vulnerabilities

Table E.11: Results of the Generic Algorithms for CWE-Combined Vulnerability Detection

Model	Accuracy	Precision	Recall	F1-Score	ROC-AUC
Random Forest	0.6510	0.6288	0.7231	0.6727	0.6516
Lasso	0.5589	0.5362	0.8197	0.6483	0.5610
ElasticNET	0.5589	0.5362	0.8197	0.6483	0.5610
Naive Bayes	0.6042	0.6498	0.4378	0.5231	0.6028
Gradient Boosted Trees	**0.6710**	**0.6515**	0.7239	**0.6857**	**0.6714**
SVM	0.5033	0.4993	0.6255	0.5553	0.4974
Logistic Regression	0.5901	0.6387	0.3994	0.4915	0.5043
Linear Discriminant Analysis	0.5587	0.5360	0.8192	0.6480	0.5608
KNeighborsClassifier	0.5573	0.5530	0.5599	0.5564	0.5574
Decision Tree Classifier	0.5722	0.5655	0.5927	0.5788	0.5723
Gaussian NB	0.4957	0.4958	**0.9987**	0.6627	0.4998

E.6.1 Tunning

Table E.12: Results of the Generic Algorithms for CWE-Combined Vulnerability Detection with Tunning.

Algorithm	Accuracy	Precision	Recall	F1-Score	ROC-AUC	MCC	Best Hyperparameters
Random Forest	**0.6908**	**0.6724**	**0.7397**	**0.7044**	**0.6910**	**0.3837**	n_estimators = 2048 min_samples_leaf = 1 max_depth = 32
Decision Tree	0.6150	0.6013	0.6741	0.6356	0.6152	0.2320	max_depth = 32 min_samples_leaf = 10 splitter = best

Appendix F

Results of Neural Network

F.1 Detection of CWE-119 Vulnerabilities

F.1.1 Embedding Layer Approach

Table F.1: Results of the NN Models for CWE-119 Vulnerability Detection using Embedding Layer.

	Accuracy	Precision	Recall	F1-Score	ROC-AUC	MCC
CNN	**0.8914**	**0.9047**	**0.8994**	**0.8913**	**0.9461**	**0.7863**
BLSTM	0.8631	0.8387	0.8954	0.8661	0.9147	0.7280
GRU	0.8510	0.8584	0.8370	0.8476	0.9028	0.7023

F.1.2 GloVe Approach

Table F.2: Results of the NN Models for CWE-119 Vulnerability Detection using GloVe.

	Accuracy	Precision	Recall	F1-Score	ROC-AUC	MCC
CNN	0.8938	0.8983	0.8856	0.8919	**0.9500**	0.7877
BLSTM	**0.9009**	**0.9106**	**0.8866**	0.8985	0.9304	**0.8019**
GRU	0.8968	0.8995	0.8910	0.8952	0.9431	0.7936

F.2 Detection of CWE-120 Vulnerabilities

F.2.1 Embedding Layer Approach

Table F.3: Results of the NN Models for CWE-120 Vulnerability Detection using Embedding Layer.

	Accuracy	Precision	Recall	F1-Score	ROC-AUC	MCC
CNN	**0.8817**	**0.9060**	**0.8639**	**0.8800**	**0.9345**	**0.7639**
BLSTM	0.8746	0.8825	0.8583	0.8730	0.8956	0.7496
GRU	0.8184	0.8041	0.8441	0.8236	0.8694	0.6375

F.2.2 GloVe

Table F.4: Results of the NN Models for CWE-120 Vulnerability Detection using GloVe.

	Accuracy	Precision	Recall	F1-Score	ROC-AUC	MCC
CNN	0.8674	0.8412	0.9074	0.8730	0.9232	0.7371
BLSTM	**0.8858**	**0.8860**	0.8867	**0.8864**	0.9218	**0.7716**
GRU	0.8782	0.8670	**0.8947**	0.8800	**0.9234**	0.7567

F.3 Detection of CWE-469 Vulnerabilities

F.3.1 Embedding Layer Approach

Table F.5: Results of the NN Models for CWE-469 Vulnerability Detection using Embedding Layer.

	Accuracy	Precision	Recall	F1-Score	ROC-AUC	MCC
CNN	**0.8886**	0.8838	**0.9019**	**0.8927**	**0.9572**	**0.7770**
BLSTM	0.7952	**0.9501**	0.6342	0.7614	0.9450	0.6312
GRU	0.7819	0.8154	0.7444	0.7783	0.8370	0.5668

F.3.2 GloVe

Table F.6: Results of the NN Models for CWE-469 Vulnerability Detection using GloVe.

	Accuracy	Precision	Recall	F1-Score	ROC-AUC	MCC
CNN	0.8943	0.8837	**0.9148**	0.8990	**0.9582**	0.7887
BLSTM	**0.8975**	0.8956	0.9056	**0.9055**	0.9450	**0.7981**
GRU	0.8876	**0.9073**	0.8704	0.8885	0.9284	0.7760

F.4 Detection of CWE-476 Vulnerabilities

F.4.1 Embedding Layer Approach

Table F.7: Results of the NN Models for CWE-476 Vulnerability Detection using Embedding Layer.

	Accuracy	Precision	Recall	F1-Score	ROC-AUC	MCC
CNN	**0.8238**	0.8371	**0.7986**	**0.8174**	**0.9051**	**0.6482**
BLSTM	0.8016	**0.8688**	0.7044	0.7780	0.8639	0.6130
GRU	0.7830	0.8134	0.7412	0.7756	0.8386	0.5783

F.4.2 GloVe

Table F.8: Results of the NN Models for CWE-476 Vulnerability Detection using GloVe.

	Accuracy	Precision	Recall	F1-Score	ROC-AUC	MCC
CNN	0.8181	0.8298	**0.7944**	**0.8117**	**0.8922**	**0.6365**
BLSTM	**0.8222**	**0.8514**	0.7751	0.8115	0.8904	0.6465
GRU	0.8067	0.8295	0.7659	0.7964	0.8634	0.6148

F.5 Detection of CWE-Others Vulnerabilities

F.5.1 Embedding Layer Approach

Table F.9: Results of the NN Models for CWE-Other Vulnerability Detection using Embedding Layer.

	Accuracy	Precision	Recall	F1-Score	ROC-AUC	MCC
CNN	**0.8181**	**0.8298**	0.7944	**0.8117**	**0.8922**	**0.6365**
BLSTM	0.7862	0.7499	**0.8574**	0.8000	0.8445	0.5785
GRU	0.8067	0.8295	0.7659	0.7964	0.8634	0.6148

F.5.2 GloVe

Table F.10: Results of the NN Models for CWE-Other Vulnerability Detection using GloVe.

	Accuracy	Precision	Recall	F1-Score	ROC-AUC	MCC
CNN	**0.8041**	**0.8217**	0.7755	**0.7980**	0.8764	**0.6091**
BLSTM	0.7793	0.8006	0.7426	0.7705	0.8544	0.5605
GRU	0.7979	0.7992	**0.7944**	0.7968	**0.8825**	0.5958

F.6 Detection of CWE-Combined Vulnerabilities

F.6.1 Embedding Layer Approach

Table F.11: Results of the NN Models for CWE-Combined Vulnerability Detection using Embedding Layer.

	Accuracy	Precision	Recall	F1-Score	ROC-AUC	MCC
CNN	**0.8221**	**0.8519**	0.7763	**0.8124**	**0.8853**	**0.6465**
BLSTM	0.7927	0.8167	0.7741	0.7851	0.8735	0.5985
GRU	0.7745	0.7545	**0.8083**	0.7805	0.8421	0.5506

F.6.2 GloVe

Table F.12: Results of the NN Models for CWE-Combined Vulnerability Detection using GloVe.

	Accuracy	Precision	Recall	F1-Score	ROC-AUC	MCC
CNN	0.7916	0.7797	**0.8080**	0.7937	0.8609	0.5837
BLSTM	0.7855	0.7993	0.7875	0.7646	0.8524	0.5611
GRU	**0.8043**	**0.8140**	0.7847	**0.7991**	**0.8714**	**0.6089**