Machine Learning
**Second year**

# DETECTING VULNERABILITIES IN C/C++ SOURCE CODE USING MACHINE LEARNING APPROACHES

**Candidate: Sorin-Octavian FURDUI**

**Research supervisor: Assoc.prof. Dr.eng. Dan PESCARU**

June 2025

# ABSTRACT

As software systems grow in complexity, identifying and mitigating security vulnerabilities at the source code level has become increasingly important. This research focuses on leveraging machine learning techniques for vulnerability detection in C/C++ programs, with an emphasis on comparing a diverse set of models and data representations. Previous work in this field has been also analyzed, in order to make some improvements.

I decided to evaluate classical machine learning models, including Random Forests, Decision Trees, Boosting, k-Nearest Neighbors and Logistic Regression, using multiple vectorization techniques such as Count Vectorizer, TF-IDF and Hashing. These approaches are applied to Abstract Syntax Trees (ASTs) representations of source code, aiming to understand how well traditional models perform in code-based security classification tasks.

In addition to classical methods, I explored the structural aspects of code through deep learning. Abstract Syntax Trees (ASTs) are processed using convolutional and recurrent neural networks (CNN and LSTM) to capture syntactic patterns indicative of vulnerabilities. Furthermore, I investigated the use of Control Flow Graphs (CFGs) in combination with Graph Neural Networks (GNNs) in order to capture more complex, execution-based relationships within code.

To ensure a comprehensive evaluation, I incorporated various datasets that range in complexity—from artificially generated simple code snippets to real-world, complex software projects. This makes it possible to analyze model performance across different code granularities and abstraction levels.

The goal is to highlight the strengths and trade-offs between different machine learning approaches in the context of source code vulnerability detection. Overall, this research suggests a hybrid approach, combining both traditional and deep learning techniques. In addition, the analysis of current datasets used for this problematic can help future work in this area.

**Keywords**: Source code analysis, software vulnerabilities, machine learning, Abstract Syntax Tree (AST), Control Flow Graph (CFG), neural networks, C/C++ security

# TABLE OF CONTENTS

# LIST OF FIGURES AND TABLES

# 1. INTRODUCTION

## 1.1 Problem Statement

The detection of software vulnerabilities in C/C++ code remains a critical yet complex challenge in software security. Traditional static and dynamic analysis tools, while effective in some scenarios, often generate numerous false positives and require expert interpretation. Furthermore, these tools struggle to adapt to new patterns of vulnerabilities and to analyze large-scale or real-world codebases efficiently.

With the advent of machine learning, particularly deep learning, new methods have emerged that can automatically learn from large code repositories. However, questions remain regarding the performance, reliability, and generalization capabilities of these models when applied to diverse C/C++ codebases. Additionally, there is a lack of standardization in datasets, feature engineering techniques, and evaluation metrics, making it difficult to benchmark and compare different approaches.

This research seeks to bridge the gap between traditional detection methods and modern machine learning techniques by systematically evaluating various machine learning models—both classical and deep learning—on well-established datasets, focusing on C/C++ code vulnerabilities.

## 1.2 Objectives

The primary objectives of this research are to conduct a comprehensive analysis of both classical and deep learning-based models for detecting vulnerabilities in C and C++ code, assessing their respective strengths and limitations. The study aims to construct and preprocess relevant datasets that include both synthetic and real-world examples, ensuring that they are properly labeled and structured to support machine learning tasks.

Another key objective is to explore a range of feature extraction techniques, including textual vectorization methods and structural analysis through abstract syntax trees (ASTs) and control flow graphs (CFGs). The research involves the implementation and evaluation of various machine learning models—such as Decision Trees, Random Forests, Convolutional Neural Networks (CNNs), Long Short-Term Memory networks (LSTMs), and Graph Neural Networks (GNNs)—using these extracted features. Finally, the study seeks to compare the performance of these models across different code representations and feature sets, highlighting trade-offs in terms of accuracy, interpretability, and computational complexity.

## 1.3 Scope and Limitations

The work presented in this research focuses on the detection of software vulnerabilities in C and C++ source code using supervised machine learning techniques. It highlights the integration of both traditional and deep learning models, employing syntactic, textual, and structural representations of code to improve detection accuracy.

The scope of the research includes the analysis of both synthetic datasets, such as Juliet (Okun et al., 2013) and complex real-world vulnerability datasets, such as DiverseVul (Chen et al., 2023), BigVul (Fan et al., 2020) and MegaVul (Ni et al., 2024). It evaluates feature extraction methods from raw code text, abstract syntax trees (ASTs), and control flow graphs (CFGs). Furthermore, it considers a range of machine learning model families, including classical algorithms, deep learning architectures, and graph neural networks (GNNs). The study is grounded in supervised learning methodologies, relying on labeled data that often includes vulnerability annotations such as CVEs.

However, the research is subject to certain limitations. It is specifically confined to the C and C++ programming languages, and its findings may not be directly applicable to other languages. Additionally, dynamic runtime behaviors, including techniques like symbolic execution or fuzz testing, are not explored in detail. The study assumes the availability of high-quality labeled data, which may not always be representative of all real-world conditions. Moreover, it does not account for adversarial or obfuscated code, which is commonly encountered in more sophisticated security threats.

## 1.4 Structure of the Dissertation

This dissertation is structured into seven core chapters, each addressing a specific component of the research process, from foundational concepts to evaluation and final conclusions.

*Chapter 2 - Literature Review*: This chapter provides a comprehensive review of existing research and techniques related to software vulnerability detection, with an emphasis on C/C++ codebases. It begins with an overview of common software vulnerabilities in these languages and discusses both static and dynamic code analysis methods. The chapter also examines the evolution and application of machine learning in software security, with specific attention to code representation techniques such as Abstract Syntax Trees (ASTs) and Control Flow Graphs (CFGs), which have gained prominence in recent ML-driven detection systems.

*Chapter 3 - Datasets*: This chapter details the datasets used in this study. It presents an overview of publicly available datasets such as the Juliet Test Suite for C/C++ and other complex real-world datasets that include vulnerability annotations (e.g., CVEs).

*Chapter 4 - Feature Engineering*: Feature extraction plays a pivotal role in the success of machine learning models. This chapter outlines the different types of features considered in this work, including textual representations (Count Vectorizer, TF-IDF and Hashing Vectorizer) and structural features derived from ASTs and CFGs. The challenges in representing code, such as variability in syntax and semantics, are also discussed.

*Chapter 5 - Methodology*: This chapter introduces the machine learning models employed in the experiments. It is divided into three main sections: classical machine learning algorithms (Decision Trees, Random Forests, Boosting, kNN and Logistic Regression), deep learning models applied to AST representations (CNNs and LSTMs), and Graph Neural Networks (GNNs) applied to CFGs. The training process and model architecture strategies are explained.

*Chapter 6 - Experiments and Results*: In this chapter, the results of experiments conducted using the different models and feature sets are presented and compared. Performance metrics such as precision, recall, F1-score, and accuracy are used to assess each model's effectiveness. The experiments are grouped based on the type of code representation and learning approach used.

*Chapter 7 - Discussion*: This chapter analyzes the outcomes of the experiments, highlighting the strengths and weaknesses of each model and approach. Issues related to model interpretability, generalization across datasets, scalability, and real-world applicability are discussed.

*Chapter 8 - Conclusion*: The final chapter summarizes the key findings of the dissertation. It reiterates the contributions made to the field of vulnerability detection and machine learning, outlines the limitations encountered during the study, and presents opportunities for future work, including the integration of runtime analysis and adversarial robustness into ML-based detection frameworks.

# 2. LITERATURE REVIEW

## 2.1 Software Vulnerabilities in C/C++

C and C++ are among the most widely used programming languages, especially in systems programming, embedded systems, and performance-critical applications. However, their lack of built-in memory safety features makes them highly susceptible to various types of vulnerabilities. Common issues include buffer overflows, use-after-free errors, integer overflows, null pointer dereferencing and format string vulnerabilities. These bugs can be exploited to gain unauthorized access, execute arbitrary code, or cause program crashes.

The persistence of these vulnerabilities is partly due to the complexity of the languages and the subtlety of many security flaws. Even experienced developers can introduce vulnerabilities. Traditional detection methods like static and dynamic analysis tools provide some support but often lack adaptability and may generate significant false positives or miss complex vulnerability patterns.

## 2.2 Static and Dynamic Code Analysis Techniques

Static analysis inspects the code without executing it, aiming to find potential vulnerabilities through code pattern recognition and logic inference. Tools like Clang Static Analyzer (Kremenek, 2008) and SonarQube (Campbell and Papapetrou, 2013) exemplify this approach. While efficient in catching syntax-based errors, static analysis struggles with complex logic and runtime-dependent bugs.

Dynamic analysis, on the other hand, involves executing the code and monitoring its behavior under various inputs. Techniques like fuzz testing, runtime instrumentation, and taint analysis are commonly used. Tools such as Valgrind (Nethercote and Seward, 2007) and AddressSanitizer (Serebryany et al., 2012) are well-known in this category. However, dynamic analysis can be time-consuming and might not cover all execution paths, leading to incomplete results.

The combination of these techniques often improves coverage but still lacks generalization and adaptability to unseen code, which has led to the exploration of machine learning-based approaches.

(Vándor et al., 2022) conducted a comparative analysis of machine learning-based vulnerability detection methods versus traditional static analysis tools. Their research highlighted the strengths and limitations of both approaches when applied to C and C++ source code. While static analyzers are rule-based and rely on predefined heuristics, machine learning models can adapt to diverse coding styles and learn complex patterns from labeled data. The study found that ML-based models were particularly effective in reducing false positives, a common shortcoming of static tools. However, the authors also noted that static analyzers still played a valuable role in detecting certain categories of vulnerabilities that machine learning models struggled with. Overall, their work supports a hybrid approach, advocating for the complementary use of both ML techniques and traditional tools to improve the reliability of vulnerability detection systems.

## 2.3 Machine Learning in Software Security

Recent advancements in machine learning (ML) have opened new possibilities for detecting vulnerabilities in source code. ML models can be trained on labeled datasets of vulnerable and non-vulnerable code to learn patterns indicative of security flaws. Techniques range from classical models like decision trees and logistic regression to deep learning architectures such as convolutional neural networks (CNNs) and recurrent neural networks (RNNs).

Studies such as Allamanis et al. (2018) and Harer et al. (2018) laid the groundwork by using ML for source code modeling and vulnerability classification. More recent research has extended these efforts using natural language processing (NLP) techniques and deep representation learning, showing significant improvements in detection accuracy.

Despite promising results, challenges remain in terms of data availability, model generalization, and interpretability. Moreover, there is an ongoing debate on whether shallow or deep models are more effective, as discussed by Mazuera-Rozo et al. (2021).

One of the most cited papers in this field, (Russell et al., 2018) proposed an automated vulnerability detection approach based on a model that processes source code as sequences of tokens and applies recurrent neural networks (RNNs), particularly Long Short-Term Memory (LSTM) networks, to learn semantic patterns associated with software vulnerabilities. By avoiding hand-crafted features and instead relying on learned embeddings, the model demonstrated promising performance in identifying vulnerable code segments. The study emphasized the importance of deep learning for capturing contextual and structural nuances in source code, thereby improving detection capabilities compared to traditional static analysis tools.

Another paper that is well known for contributions in this field is (Li et al., 2018), where a model named VulDeePecker is built. The model is built based on BLSTM layers. In other words, “the output of the learning phase is vulnerability patterns, which are coded into a BLSTM neural network”. First step is to extract library/API function calls, then transform all the code gadgets into symbolic representation, keeping the semantic meaning behind them. Those representations will be then served as input to the model. A lexical analysis is done, splitting code gadgets into keywords. The CGD contains a number of 6,166,401 tokens, of which 23,464 are different. After mapping user-defined variable names and function names to some symbolic names, the number of different tokens was reduced to 10,480.

In terms of limitations, VulDeePecker is trained on a dataset with only two vulnerabilities. Experiments with more programming languages and more vulnerabilities are considered as future work. In addition, it only deals with vulnerabilities related to library/API function calls and only accommodates data flow analysis, but not control-flow analysis.

(Hanif et al., 2021) provide a comprehensive review of software vulnerability detection techniques with a strong focus on machine learning approaches. The paper begins by outlining a taxonomy of vulnerabilities, drawing from widely recognized standards such as the Common

Weakness Enumeration (CWE), OWASP Top 10, and the SANS 25. These vulnerabilities are categorized by their origin—such as input validation flaws or configuration errors—and by their potential impact, including denial of service, arbitrary code execution, and information disclosure. The authors then delve into various machine learning techniques used to detect these vulnerabilities, analyzing both traditional models like decision trees, support vector machines, and random forests, as well as more advanced deep learning architectures such as convolutional neural networks (CNNs) and recurrent neural networks (RNNs). Each technique is examined in terms of how it represents code features (like tokens, syntax trees, and semantic information), the type of training data it uses, and the performance metrics used to evaluate it, such as accuracy, precision, recall, and F1-score.

(Knutsen and Lervik, 2022) utilize NLP techniques such as tokenization, embedding, and sequence modeling to preprocess and represent source code, enabling its use as input for ML models. They experiment with various algorithms, including traditional classifiers and deep learning models like Long Short-Term Memory (LSTM) networks, to determine their effectiveness in identifying vulnerabilities. One of the core contributions of the thesis is the demonstration that NLP-based preprocessing significantly enhances the performance of ML models in vulnerability detection tasks, especially when applied to code datasets annotated with known vulnerability labels, like SATE IV Juliet dataset. The study also discusses challenges such as class imbalance, data sparsity, and the need for high-quality labeled data. The thesis concludes that the combination of machine learning and NLP holds strong promise for automating the early detection of software vulnerabilities, though further research is needed to improve model generalization and real-world applicability.

An interesting approach was written in (Yamaguchi and Rieck, 2011). They introduced the concept of vulnerability extrapolation, a machine learning-driven technique aimed at assisting the discovery of previously unknown vulnerabilities in software code. Rather than directly classifying code as vulnerable or not, their approach involves identifying structural and semantic similarities between known vulnerable code patterns and unseen code segments. By leveraging features extracted from code syntax and control flow, their model could generalize from known examples to locate related vulnerabilities in a broader codebase. This semi-supervised strategy showed potential in guiding security analysts toward overlooked flaws, highlighting the usefulness of machine learning not only in classification tasks but also in exploratory vulnerability assessment.

There are also more methods to be explored. The technology nowadays will continue to grow and multiple ways of finding vulnerabilities will appear. An approach involving quantum computing could be observed in (Akter et al., 2022). The authors proposed a novel method for detecting vulnerabilities in source code by applying Quantum Natural Language Processing (QNLP). Their approach uses quantum computing principles to analyze code structures as if they were language, aiming to improve the precision of vulnerability detection beyond classical techniques.

## 2.4 Abstract Syntax Trees (ASTs) in C/C++ Vulnerability Detection

Abstract Syntax Trees (ASTs) are hierarchical representations of the syntactic structure of source code. They preserve the logical structure of code and are commonly used in compilers and static analyzers. In the context of ML-based vulnerability detection, ASTs can be used to extract features or fed directly into neural networks for learning structural patterns.

The research written by (Bilgin et al., 2020) uses a machine learning approach for inspecting the Abstract Syntax Tree form of source code in order to classify vulnerable and non-vulnerable code snippets. There are two traditional methods that are used in order to detect vulnerabilities: static and dynamic analysis of source code. Static analysis means that the code is examined without executing, while dynamic means that the code is executed to check how the software will perform in a run-time environment. Since machine learning has become more and more present in our lives, this article contributes mainly to some areas: development of an ML-based prediction toold that is experimentally evaluated, a method for vectorial representation of source code with keeping the syntactic and semantic relations and performance comparisons.

The challenges of this approach are the same as presented in other pieces of work. Firstly, the lack of reliable vulnerability dataset and lack of replication framework for comparative analysis of existing methods. Then, the highly imbalanced cases, that should be treated as good as possible, because the distribution of data in nature will be imbalanced. One of the steps in order to process the source code are lexical and semantic analysis. The code is split into tokens and then an AST is built. AST will help the algorithm understand the source code representation. The AST will be transformed into an one-dimensional numerical array, using some mapping techniques that keep the structural and semantic information contained in the source code. In order to map the AST into a one-dimensional array, first we need to convert it into a Complete Binary AST. We cannot convert the tree directly into an array by taking each node in order, like a tree traversal, because the relationship between nodes would be lost.

In order to preserve these relationships, we need to convert the tree by using some rotating techniques so that each parent node has exactly two children. Examples are presented in order to understand the algorithm. Finally, an encoding is done in order to obtain the one-dimensional numerical array. As the dimensions of the functions will not be the same, there are two methods that can be used in order to have an array that can be fed to a machine learning model: adding padding or cut the binary AST at a certain pre-determined level for all functions. The second approach is more efficient.

CNNs and LSTMs have been applied to linearized AST representations to learn complex code semantics. (Shen, 2018) and (Rajapaksha et al., 2022) utilized ASTs to enhance vulnerability prediction in C/C++ code. AST-based models often outperform purely textual methods due to their ability to capture code structure and logic flow more effectively.

However, parsing and linearizing ASTs require language-specific tooling and may lose contextual information during transformation. Additionally, tree structures introduce complexity in model design and increase computational cost.

## 2.5 Control Flow Graphs (CFGs) in C/C++ Vulnerability Detection

Control Flow Graphs (CFGs) represent the execution flow of programs and are particularly useful in modeling program behavior and understanding data dependencies. Each node in a CFG corresponds to a basic block of code, and edges represent possible execution paths. CFGs are instrumental in detecting vulnerabilities that depend on control paths, such as logic flaws and unreachable code.

Graph-based neural networks (GNNs) have shown great promise in processing CFGs due to their ability to capture relational and topological information. Recent works like DiverseVul (Chen et al., 2023) and efforts by (Chakraborty et al., 2021) incorporate CFG-based learning for vulnerability detection, achieving state-of-the-art results.

Models like VulDeePecker extract program slices to capture semantic context, enhancing model performance. In contrast, as shown in (Chakraborty et al., 2021), graph-based models can incorporate semantic dependencies, such as data dependency edges, allowing for more effective reasoning about code connections. However, they tend to be computationally expensive and may not perform well in resource constrained environments. A challenge with current approaches is that trained models often fail to sufficiently distinguish between vulnerable and non-vulnerable code samples, leading to false classifications. Additionally, imbalances in data between vulnerable and clean code can bias models towards non-vulnerable examples, impacting performance. For the feature extraction phase, a code property graph (CPG) was used, because it is a data structure that helps in order to keep track of the semantics of the source code. To address class imbalance, the researchers utilize the Synthetic Minority Over-sampling Technique (SMOTE). SMOTE adjusts class frequencies by sub-sampling the majority class and super-sampling the minority class until all classes have equal representation. In vulnerability prediction, the vulnerable class typically constitutes the minority. SMOTE effectively creates synthetic examples by interpolating between a minority example and its nearest neighbors from the same class. This process continues until balance is achieved between vulnerable and non-vulnerable examples. SMOTE has demonstrated effectiveness in various domains with imbalanced datasets.

# 3. DATASETS

## 3.1 Overview of Used Datasets

This research utilizes a combination of synthetic and real-world datasets to evaluate the effectiveness of machine learning models in detecting software vulnerabilities in C and C++ source code. Synthetic datasets, such as the widely used Juliet Test Suite, are particularly valuable due to their controlled structure and precise labeling. They contain thousands of examples that are deliberately constructed to include specific vulnerability types, such as buffer overflows, use-after-free errors, and null pointer dereferencing. These datasets enable researchers to develop and fine-tune models under well-defined conditions, facilitating initial experiments and reproducibility of results.

In contrast, real-world datasets introduce a much higher degree of complexity. They are typically sourced from open-source repositories, software projects with known Common Vulnerabilities and Exposures (CVEs), or curated datasets like DiverseVul. Real-world code often exhibits inconsistent formatting, varying coding styles, incomplete documentation, and intricate control structures, all of which present additional challenges for automated analysis. These datasets reflect the unpredictable and messy nature of code encountered in practical development environments and are therefore critical for assessing how well machine learning models generalize beyond synthetic examples.

By combining both types of datasets, this study ensures a robust evaluation across the spectrum of code environments. Synthetic data offers a foundation for benchmarking model accuracy, while real-world data provides insight into the models' practical applicability and resilience. Furthermore, this dual approach allows for the identification of strengths and weaknesses in different model architectures, especially in how they respond to controlled versus uncontrolled inputs. The integration of diverse datasets ultimately contributes to a more reliable and holistic understanding of how machine learning techniques can assist in securing modern software systems.

## 3.2 Juliet C/C++ Code Dataset

The Juliet Test Suite (Okun et al., 2013) is a widely used synthetic dataset developed by the National Institute of Standards and Technology (NIST) to evaluate the accuracy of static and dynamic analysis tools. It consists of thousands of small, isolated C and C++ code samples, each annotated with specific vulnerability types based on the Common Weakness Enumeration (CWE) system. Examples include buffer overflows, memory leaks, null pointer dereferences, and integer overflows.

The dataset is organized into test cases that include both "good" and "bad" code paths, facilitating supervised learning with clear binary labels. The consistency and clarity of the Juliet dataset make it an ideal starting point for training machine learning models, especially when focusing on specific vulnerability categories.

## 3.3 Complex Real-World Code Datasets

To complement the synthetic examples, this study also incorporates real-world code datasets that reflect actual vulnerabilities found in production software. These datasets are compiled from publicly available codebases and security advisories, often labeled with Common Vulnerabilities and Exposures (CVE) identifiers. Notable examples include DiverseVul, Devign, and datasets from software repositories with linked CVE disclosures.

Unlike synthetic datasets, real-world code presents a higher degree of variability, with complex control flows, mixed coding styles, and subtle flaws that are harder to detect. These characteristics make real-world datasets critical for evaluating the generalization capability and robustness of machine learning models in practical scenarios.

## 3.4 Data Preprocessing and Annotation

Before applying machine learning techniques, all datasets undergo a standardized preprocessing pipeline. This includes code cleaning, tokenization, normalization, and parsing into structural representations such as Abstract Syntax Trees (ASTs) and Control Flow Graphs (CFGs). In textual feature pipelines, techniques like stopword removal and vectorization (e.g., TF-IDF, CountVectorizer) are applied.

Annotation quality is critical in supervised learning. For synthetic datasets like Juliet, labels are directly inherited from the test suite's structure. In real-world datasets, labels are derived from CVE tags and manual verification, ensuring the accurate classification of vulnerable versus non-vulnerable code samples. The preprocessing step also includes deduplication and balancing techniques to mitigate issues of class imbalance, which are common in vulnerability datasets.

## 3.5 Exploratory Data Analysis

For the next pages, there is presented some exploratory data analysis. In *Figures 1-3*, there is a wordcloud with most used words from each dataset. Then, *Table 1* presents some statistics regarding the datasets, like number of lines, number of characters, number of tokens, cyclomatic complexity, average token length and average line length. Cyclomatic complexity was calculated by counting all the loop or conditional operators from functions. In *Figures 4-6*, there is a distribution of function length across all the datasets, with a hue for vulnerability. In addition, we can observe the final dataset structure in the pie chart from *Figure 7*.

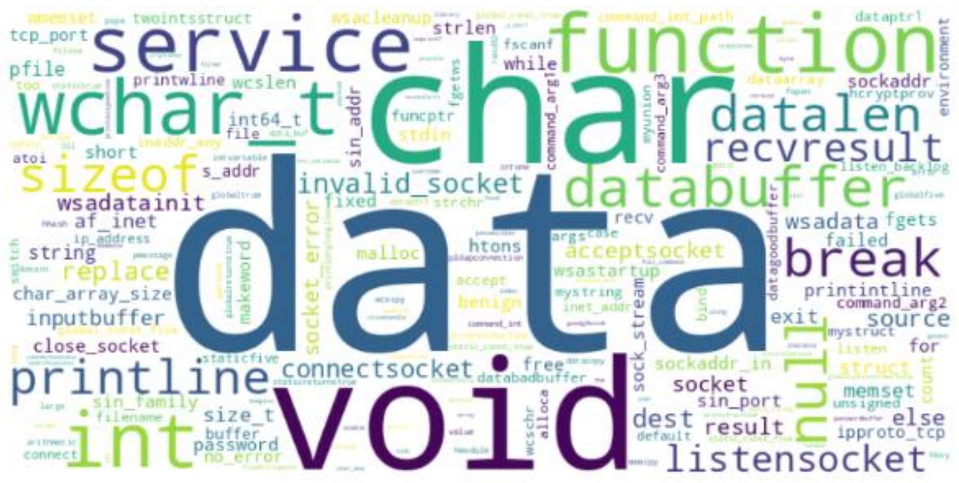


*Figure 1 - Wordcloud of most used words for Juliet dataset*

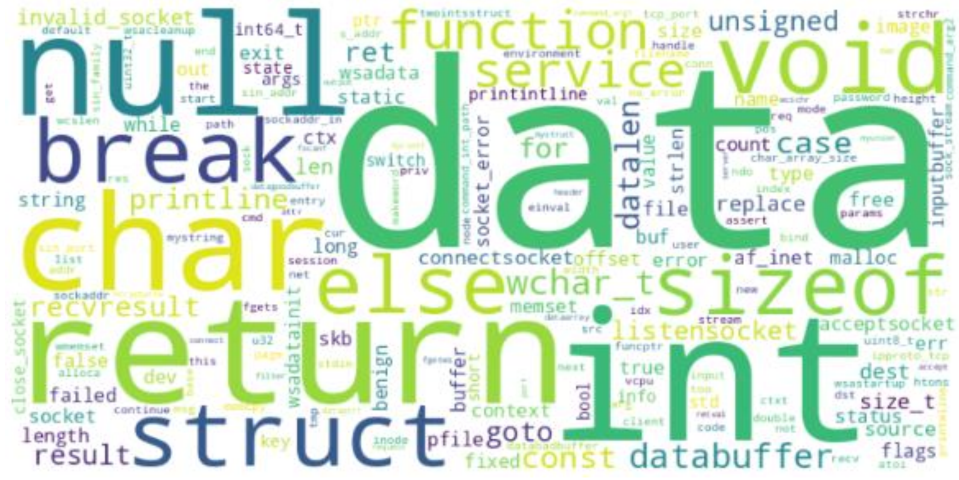


*Figure 2 - Wordcloud of most used words for combined dataset*

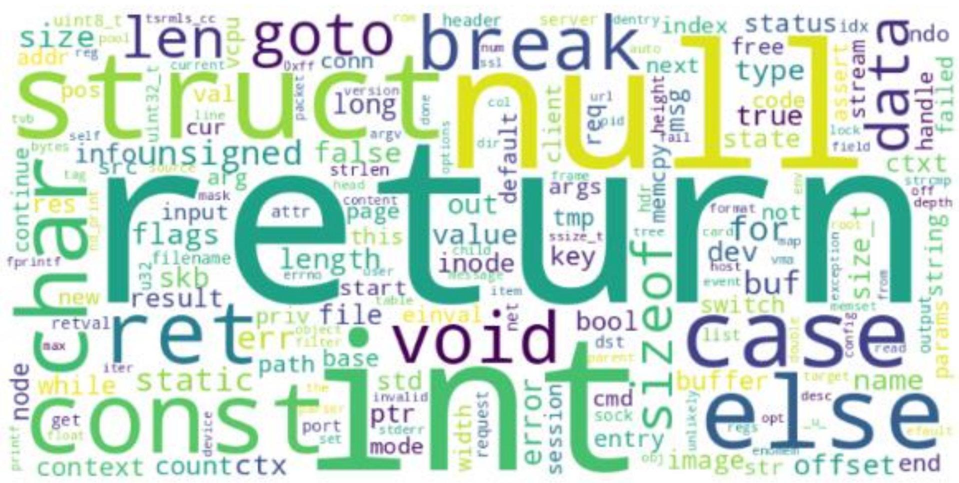


*Figure 3 - Wordcloud of most used words for real-world code dataset*

| Dataset | Metric | Number of lines | Average line length | Cyclomatic complexity | Number of chars | Number of tokens | Average token length |
|---|---|---|---|---|---|---|---|
| Juliet Dataset | Min | 2 | 9.67 | 1 | 30 | 5 | 5.11 |
| | Max | 288 | 60.33 | 45 | 6018 | 746 | 18.25 |
| | Mean | 26.62 | 24.59 | 4.24 | 585.47 | 77.58 | 7.63 |
| | Std | 23.79 | 7.26 | 3.71 | 456.28 | 56.20 | 1.26 |
| DiverseVul + MegaVul +BigVul dataset | Min | 1 | 8.03 | 1 | 75 | 4 | 3.21 |
| | Max | 278 | 379 | 120 | 4707 | 924 | 34.38 |
| | Mean | 43.21 | 25.90 | 7.48 | 1063.52 | 128.09 | 8.89 |
| | Std | 29.67 | 7.82 | 7.89 | 938.28 | 119.10 | 2.21 |
| All | Min | 1 | 8.03 | 1 | 30 | 4 | 3.21 |
| | Max | 288 | 379 | 120 | 6018 | 924 | 34.38 |
| | Mean | 34.58 | 25.22 | 5.79 | 814.78 | 101.81 | 8.24 |
| | Std | 32.87 | 7.56 | 6.29 | 766.58 | 95.31 | 1.89 |

*Table 1 – Statistical features for all datasets*

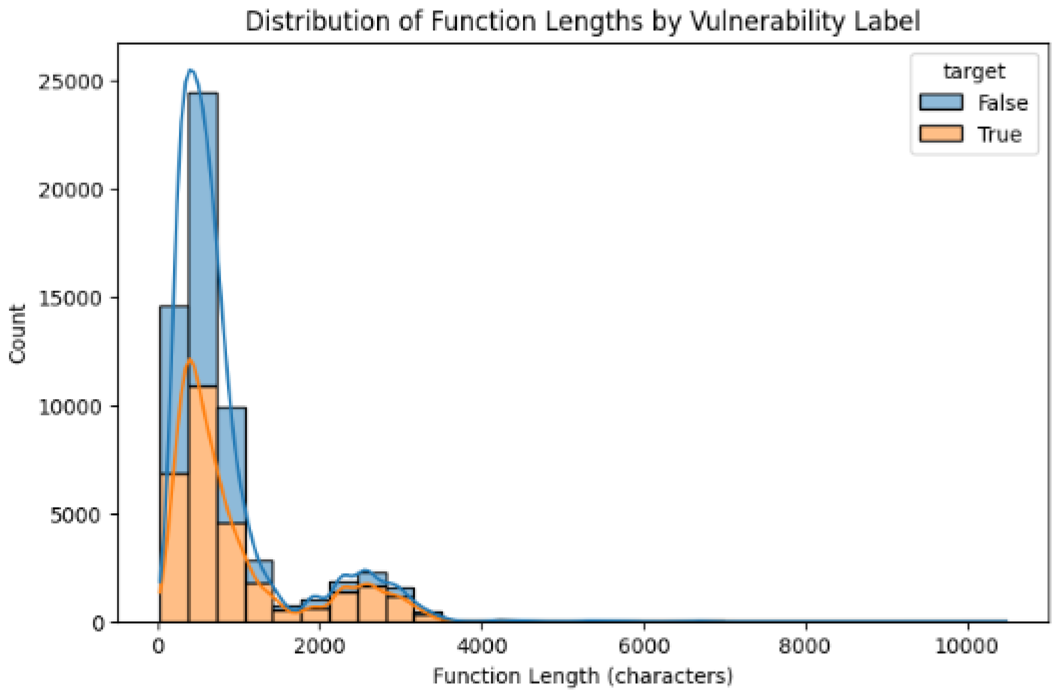


*Figure 4 – Distribution of function length for Juliet dataset*

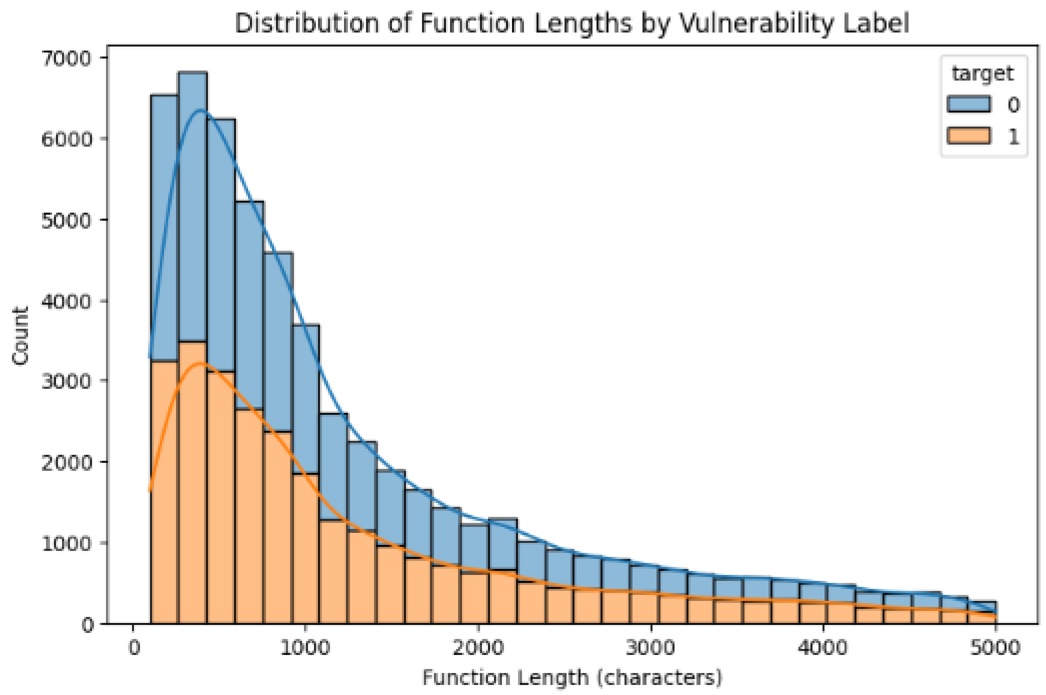


*Figure 5 – Distribution of function length for real-world code dataset*

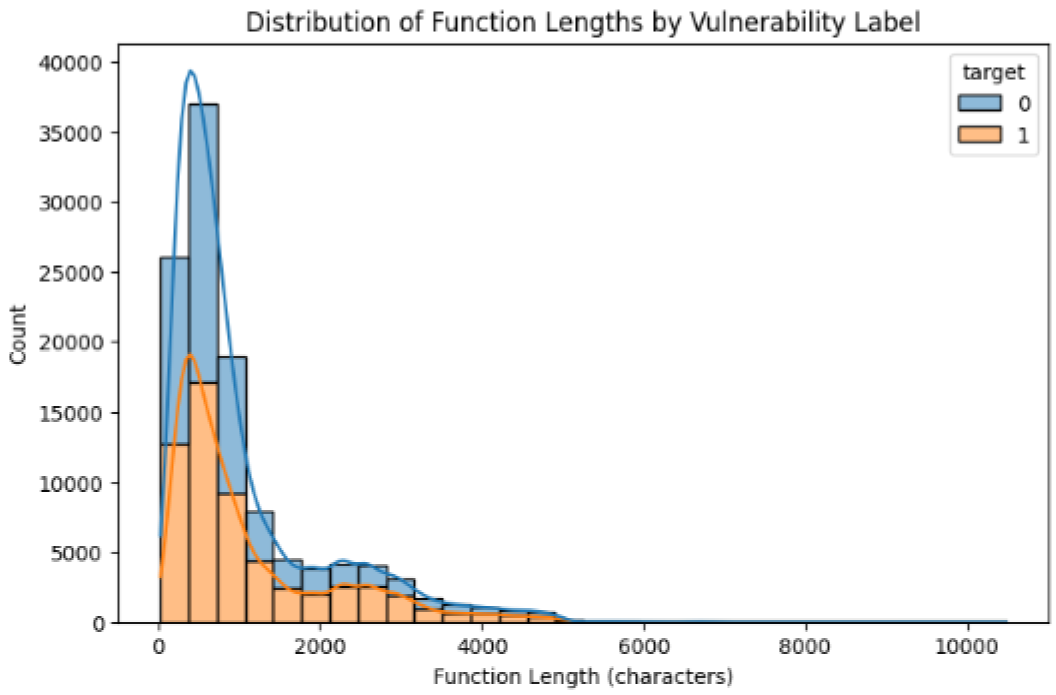


*Figure 6 – Distribution of function length for combined dataset*

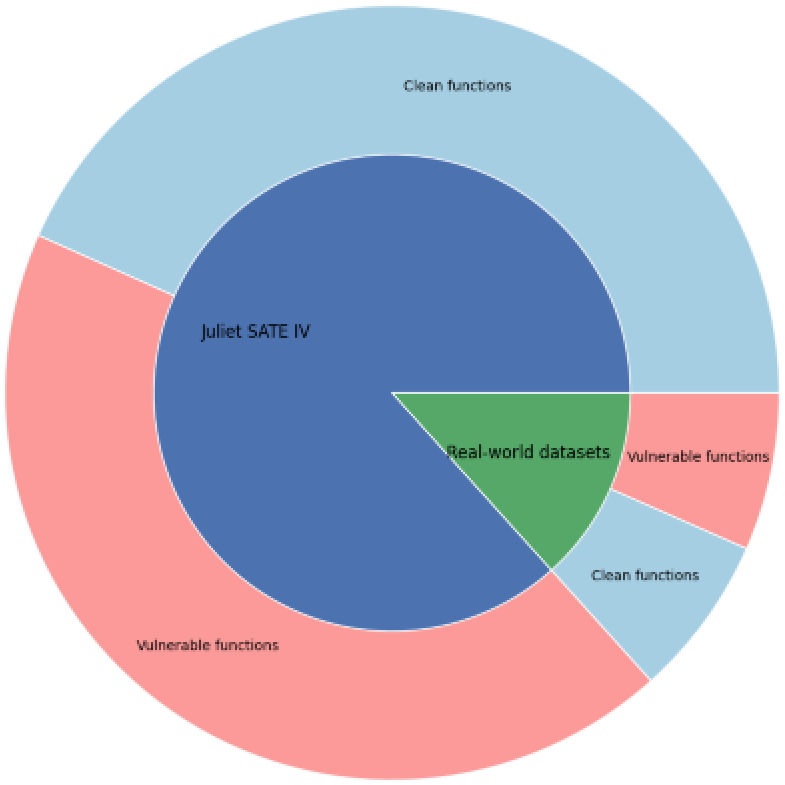


*Figure 7 - Pie chart of final dataset*

# 4. FEATURE ENGINEERING

Feature engineering plays a pivotal role in transforming raw C and C++ source code into structured inputs suitable for machine learning algorithms. It is the process by which relevant attributes are extracted from code to enable models to learn patterns associated with vulnerable or secure programming practices. This chapter outlines three primary categories of features explored in this research: classical code metrics, textual vectorization techniques, and structural features derived from compiler representations such as Abstract Syntax Trees (ASTs) and Control Flow Graphs (CFGs). The final section of the chapter discusses the challenges encountered when representing source code in ways that are both expressive and computationally efficient.

## 4.1. Textual vectorization techniques

In the context of software vulnerability detection, textual vectorization serves as a fundamental preprocessing step that transforms source code from its raw textual form into a structured numerical representation suitable for machine learning models. This transformation enables statistical algorithms to process and learn from code patterns, syntax, and token usage without requiring deep semantic understanding.

Although source code possesses formal syntax and structure, it can be treated analogously to natural language for the purposes of statistical modeling. This approach leverages token-based features to capture meaningful signals related to code complexity, style, and potential vulnerability patterns. Three primary vectorization techniques are employed in this research: Count Vectorization, TF-IDF (Term Frequency–Inverse Document Frequency), and Hashing Vectorization. Each of these methods exhibits unique characteristics with respect to interpretability, computational efficiency, and performance trade-offs.

### 4.2.1. Count Vectorizer

The Count Vectorizer is a foundational technique that represents each document—in this case, each source code file or function—as a vector of token counts. It constructs a vocabulary of all distinct tokens present across the corpus and encodes each sample by counting the frequency of these tokens.

For example, a C/C++ function containing tokens such as if, while, malloc, and { would be represented as a sparse vector, where each dimension corresponds to a specific token and the value denotes its frequency within the document. This bag-of-words approach, while simplistic, effectively captures surface-level statistical information about code.

### 4.2.2 TF-IDF Vectorizer

TF-IDF vectorization enhances the bag-of-words model by assigning weights to tokens based not only on their frequency within individual documents (term frequency) but also on their rarity across the entire dataset (inverse document frequency). The intuition behind this technique is that tokens which appear frequently in a single document but rarely across the dataset may carry more informational value.

In the context of vulnerability detection, TF-IDF helps de-emphasize ubiquitous tokens such as int, return, and {, which are common across all programs, while highlighting more discriminative tokens like strcpy, system, or exec, which are often associated with insecure programming practices.

### 4.2.3 Hashing Vectorizer

The Hashing Vectorizer provides a scalable and memory-efficient alternative to explicit vocabulary-based approaches. Rather than maintaining a growing dictionary of tokens, it uses a hashing function to map each token to a fixed index in a predefined-length vector. This technique is particularly suitable for large-scale datasets where vocabulary size becomes computationally burdensome.

Although hashing may lead to collisions—where multiple tokens map to the same index—the approach has been empirically shown to maintain model performance in many applications, provided the dimensionality is sufficiently large.

## 4.2. Structural features from AST

To capture deeper syntactic and semantic relationships in code, this research also explores structural features derived from program analysis representations. Abstract Syntax Trees (ASTs) model the syntactic structure of source code in a hierarchical tree format, like the structure presented below in *Figure 8*. Each node in an AST corresponds to a syntactic construct, such as a loop, condition, function call, or variable declaration. ASTs preserve the nested and compositional nature of code, allowing machine learning models, particularly deep learning architectures like convolutional neural networks (CNNs) and long short-term memory networks (LSTMs), to learn from patterns of structural similarity and variation. These models can identify sequences or subtrees that commonly appear in vulnerable code segments.

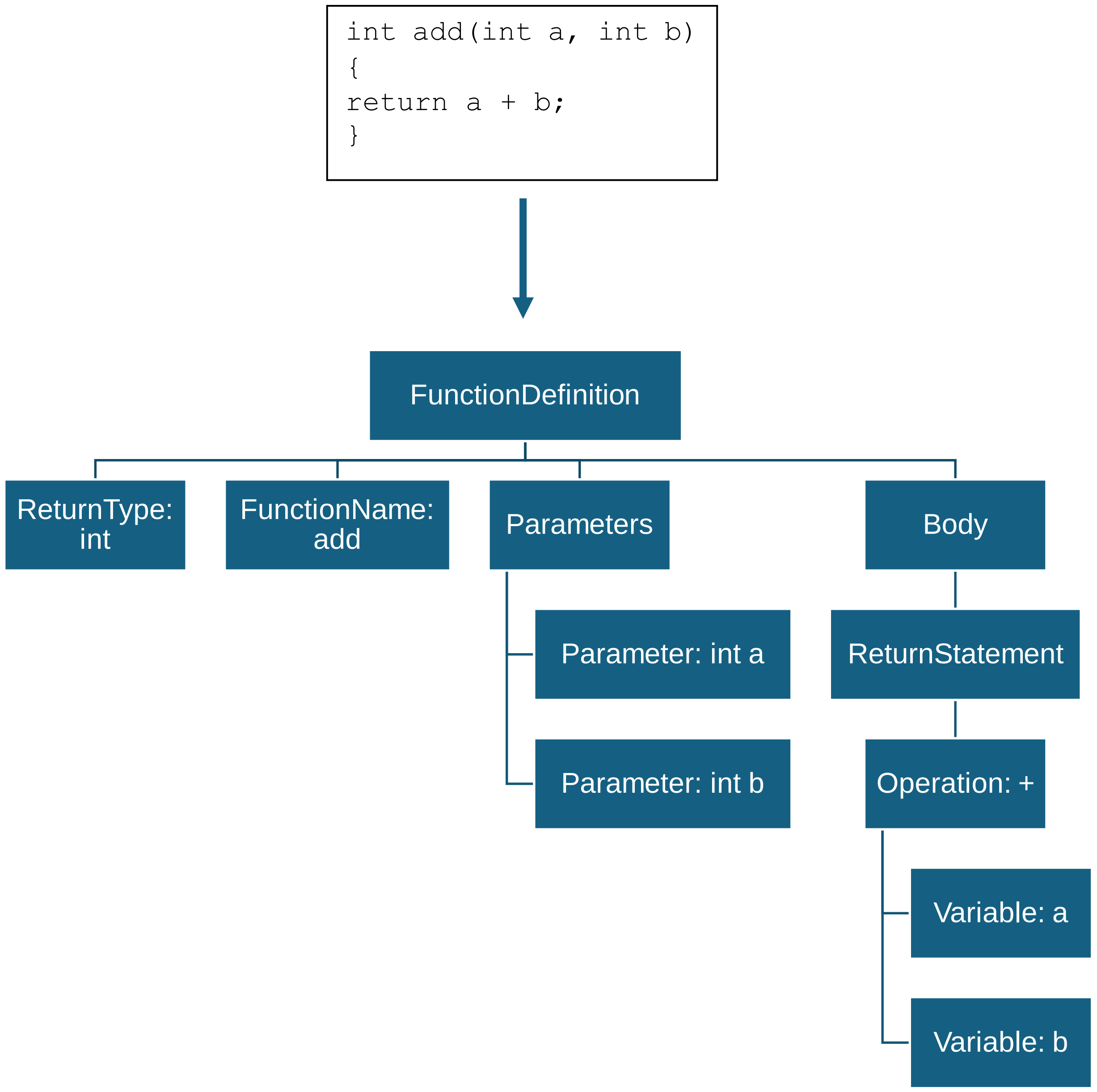
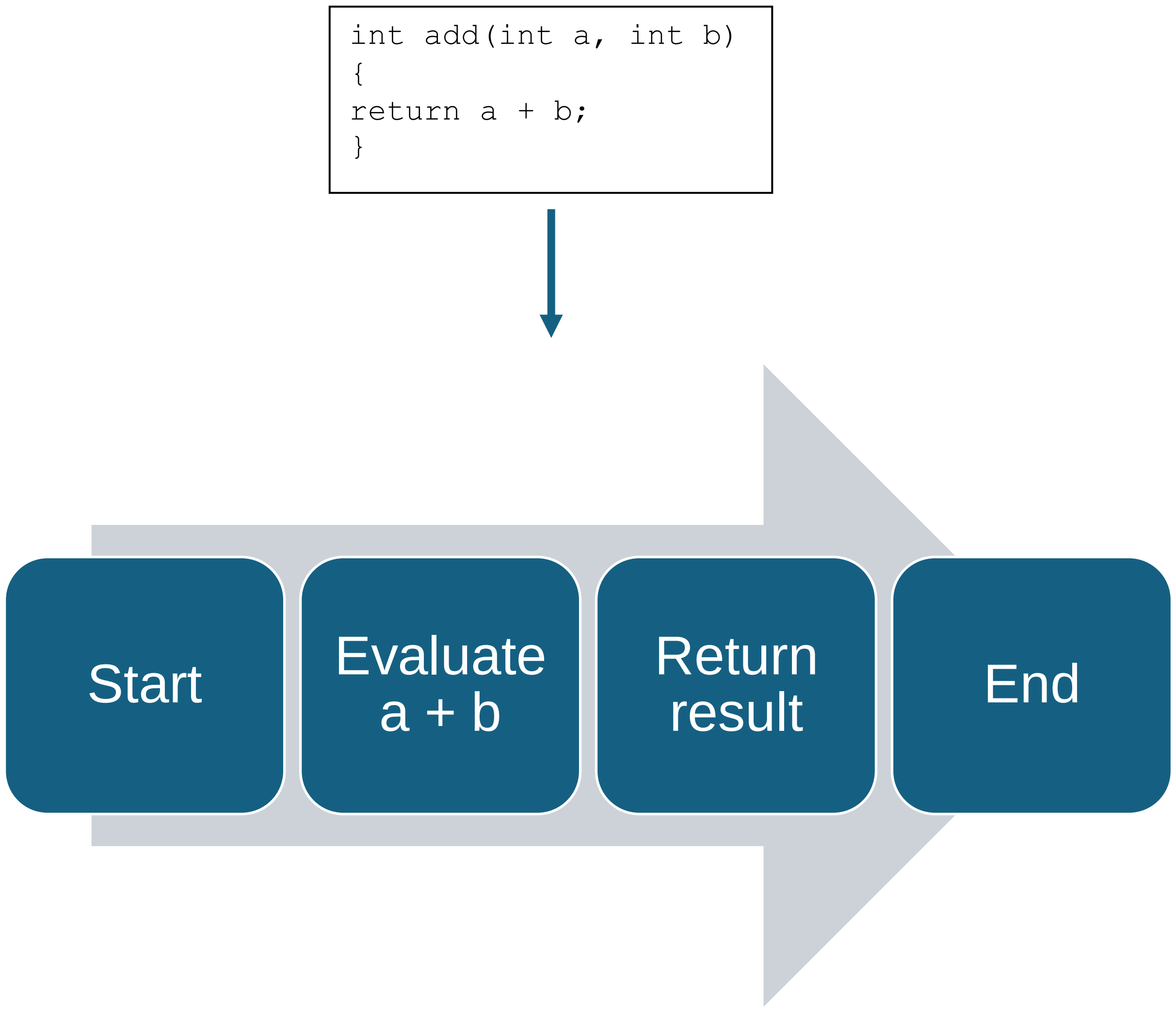

```
int add(int a, int b)
{
return a + b;
}
```

*Figure 8 – Function code to AST structure*

## 4.3. Structural features from CFG

Control Flow Graphs (CFGs), in contrast, emphasize the execution logic of a program, as shown below in *Figure 9*. A CFG represents basic blocks of code as nodes and the flow of execution between them as directed edges. This graphical representation is highly suitable for detecting vulnerabilities that emerge from the interaction of code paths or the improper sequencing of operations. Graph Neural Networks (GNNs) are particularly effective in processing CFGs, as they propagate information between interconnected nodes and capture contextual dependencies. By analyzing the control logic in a program, CFG-based features contribute significantly to detecting complex vulnerabilities such as buffer overflows, race conditions, and use-after-free errors.



```
int add(int a, int b)
{
return a + b;
}
```

*Figure 9 – Function code to CFG structure*

### 4.4. Code representation challenges

Despite the power of these techniques, several challenges arise when engineering features from source code. One of the main difficulties is the diversity of coding styles and implementations. Developers may write functionally equivalent code in many syntactically distinct ways, leading to high variance and potential overfitting in models trained on surface-level features. Additionally, many textual representations fail to capture control dependencies and semantic meaning, which are critical in understanding subtle bugs. Even structural representations like ASTs and CFGs require careful preprocessing to normalize node types, abstract identifiers, and eliminate noise. Another significant limitation is the semantic gap between static code features and runtime behavior. Certain classes of vulnerabilities only manifest during program execution, and purely static analysis may miss these dynamic interactions. Finally, the quality of the feature extraction process is inherently tied to the completeness and accuracy of code labeling. In real-world datasets, mislabeling or inconsistencies in vulnerability annotations can hinder model generalization and lead to misleading evaluations.

# 5. METHODOLOGY

This chapter outlines the methodological framework adopted for the detection of vulnerabilities in C and C++ source code using supervised machine learning approaches. The proposed methodology is structured around three principal families of models: classical machine learning algorithms, deep learning architectures leveraging Abstract Syntax Trees (ASTs), and Graph Neural Networks (GNNs) operating on Control Flow Graphs (CFGs). Each method is evaluated in the context of different feature extraction techniques, code representations, and learning paradigms. The goal is to systematically explore and compare various approaches for their effectiveness, interpretability, and scalability in vulnerability detection tasks.

## 5.1. Classical machine learning models

Classical machine learning models are widely employed in static code analysis due to their interpretability and efficiency. These algorithms are trained on engineered features derived from the source code, including statistical code metrics and vectorized textual data. There are a lot of papers that demonstrate the potential of this kind of models in the context of this problematic.

According to (Chernis and Verma, 2018), classical ML models, such as decision trees and support vector machines can help detecting vulnerabilities in source code. Their paper evaluates various ML models to classify code as vulnerable or not. Their work focuses on analyzing features extracted from source code (like function calls, code patterns, etc.) and shows that machine learning can improve detection accuracy compared to manual or rule-based methods. The study highlights how data-driven approaches can assist in early identification of security flaws during software development.

Another paper that explains the usage of classical ML models in this field is written by (Medeiros et al., 2020). This research focuses on using software metrics in order to build an efficient machine learning model that can classify vulnerabilities from source code. Some of the most important features extracted from the source code are CountOutput, CountInput, CountLine, MaxNesting, AvgLineCode and many others. The problem is to choose only the important features when training the model, so that it is both efficient and accurate. The models used in this scope are Random Forest, KNN, SVM and boosting algorithms. There are four possible outputs for the model: highly critical, critical, low-critical and non-critical. Regarding metrics, different metrics are used in order to achieve a good performance. For example, when it comes to highly-critical cases, the recall is the most important, as we want to correctly classify the vulnerable code. Cross validation techniques are used to avoid overfitting and dimensionality reduction techniques are used in order to avoid the use of irrelevant and redundant software metrics. In general, each model will have it’s own top features when making decisions. Considering this, “giving privilege to a group of metrics for building vulnerability prediction models does not seem to be a promising idea”. In terms of models used, decision trees and SVM tend to make better generalization than other models. That is because they build simpler models that can fit well into unknown data.

### 5.1.1. Decision Trees

Simply explained, a decision tree is a predictive model that works like a flowchart. You start at the top with a question about the data. Depending on the answer (yes or no), you move down the tree to another question, and you continue until you reach a final decision or prediction at a leaf node. Decision trees are powerful because they can capture complex patterns in data, but they also have a big weakness: they tend to overfit. This means they can perform very well on the training data but badly on unseen data because they memorize specific details instead of learning general rules.

With other words, Decision Trees serve as interpretable, rule-based models that recursively partition the feature space based on information gain or Gini impurity. In the context of vulnerability detection, they help identify combinations of code patterns and metrics. Their hierarchical structure allows human analysts to trace decisions leading to the classification of code as vulnerable or benign. Decision Trees model can interpret the AST structured data well, which makes it a great choice for analyzing code functions.

### 5.1.2. Random Forests

Ensemble models can be another solution for detecting vulnerabilities. A random consists of building many decision trees instead of just one. Each tree is trained on a different random subset of data. This introduces more randomness and diversity among the trees. Each tree in the forest makes its own independent prediction. If you are solving a classification problem, the random forest takes a majority vote across all trees. If you are solving a regression problem, it averages the output of all trees.

Random Forests are ensemble models consisting of multiple decision trees, each trained on a random subset of the training data and feature space. The ensemble approach reduces overfitting and improves generalization. This robustness is particularly beneficial when working with heterogeneous code features, where individual trees may be sensitive to noise or class imbalance.

### 5.1.3. Boosting

Gradient Boosting is an ensemble method that constructs a model in a sequential manner, where each new learner attempts to correct the residual errors made by the previous models. The key idea is to minimize a specified loss function using gradient descent techniques. In each iteration, the algorithm fits a new weak learner to the negative gradient of the loss function with respect to the model's output. By iteratively adding learners that target these residuals, Gradient

Boosting gradually refines its predictions, improving both bias and variance. They are suitable for vulnerability detection tasks that involve subtle code-level signals distributed across many features.

AdaBoost, short for Adaptive Boosting, is another sequential ensemble technique that focuses on improving classification performance by assigning higher weights to incorrectly classified samples in each iteration. Initially, all samples are given equal weights. After training each weak learner (usually a shallow decision tree), the algorithm increases the weights of misclassified samples, forcing subsequent learners to pay more attention to those difficult cases.

### 5.1.4. k-Nearest Neighbors

The k-Nearest Neighbors (k-NN) algorithm classifies samples based on the majority class among their k closest neighbors in feature space. It is simple and intuitive, but its performance is highly dependent on the distance metric and feature scaling. In this study, k-NN is evaluated to benchmark its effectiveness on high-dimensional, vectorized representations of source code.

### 5.1.5. Logistic Regression

Logistic regression is a fundamental supervised learning algorithm widely used for binary classification problems, including software vulnerability detection. It models the probability that a given input belongs to a particular class—in this case, whether a segment of code is vulnerable or not—using the logistic (sigmoid) function. This function maps the output of a linear combination of features to a value between 0 and 1, making it ideal for probabilistic interpretation.

In the context of high-dimensional data, such as feature vectors derived from source code through vectorization techniques or structural analysis, logistic regression can be prone to overfitting. To mitigate this, regularization techniques are employed to constrain the model's complexity and enhance generalization performance. Two of the most used forms of regularization are L1 (Lasso) and L2 (Ridge) regularization.

L1 regularization, also known as Lasso, adds a penalty equal to the absolute value of the coefficients to the loss function. This approach encourages sparsity in the model by pushing less important features toward zero. As a result, it serves a dual purpose: it reduces overfitting and performs feature selection. In software vulnerability detection, L1 regularization can help identify which code features—such as specific tokens, syntactic structures, or metrics—are most indicative of vulnerabilities, effectively filtering out irrelevant or redundant attributes.

On the other hand, L2 regularization, or Ridge regression, penalizes the squared magnitude of the coefficients. Unlike L1, it does not force coefficients to zero but rather shrinks them uniformly. This produces more stable models, especially when multicollinearity exists among the features. In the domain of code analysis, L2 regularization helps maintain a more nuanced representation of multiple correlated patterns that may jointly influence vulnerability likelihood.

The choice between L1 and L2 regularization depends on the specific characteristics of the dataset. L1 is preferable when interpretability and feature sparsity are priorities, while L2 is more suitable when the model needs to capture complex inter-feature relationships without eliminating variables entirely. Many modern implementations of logistic regression, such as those found in scikit-learn, also support elastic net regularization, which combines both L1 and L2 to balance their respective advantages. In the context of textual feature vectors, logistic regression can serve as a useful benchmark for assessing the discriminative power of different representations.

## 5.2. Deep learning with AST

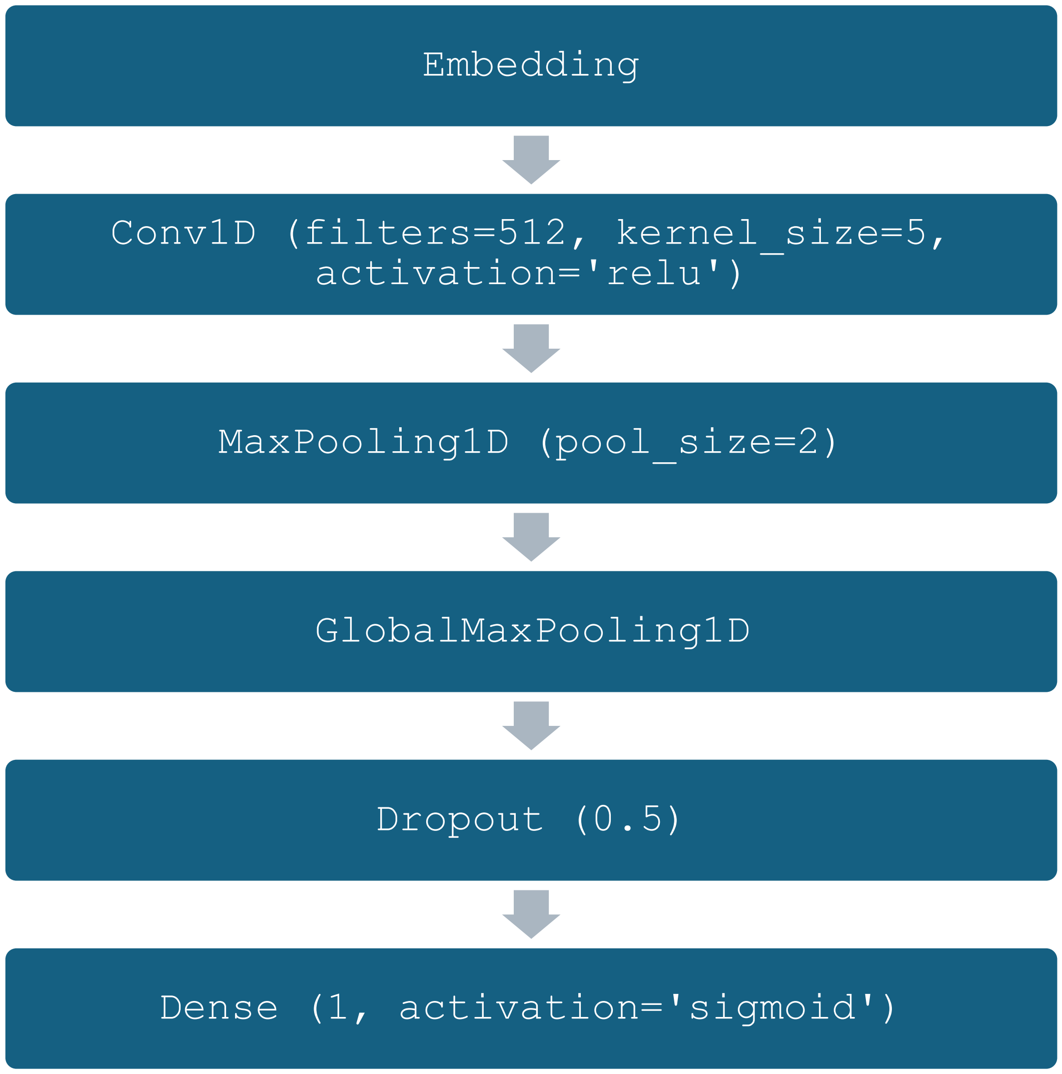
Deep learning models are particularly adept at capturing non-linear and hierarchical patterns in data. When applied to source code, they can be trained on structural representations such as Abstract Syntax Trees (ASTs), which encode the syntactic structure of code statements and expressions, or Control Flow Graphs (CFGs), which encode the process of code execution. As shown in *Chapter 2: Literature Review*, there is a lot of research done in this field and still a lot to explore.

### 5.2.1 CNN-Based Models

Convolutional Neural Networks (CNNs), originally developed for image recognition, have been adapted to process token sequences extracted from AST traversals or flattened code snippets. As shown in (Chen, 2015), a simple CNN architecture can achieve very strong results on a variety of sentence classification tasks, like sentiment analysis (positive/negative) or question classification. There were applied 1D convolutions over word embeddings rather than over images, and it turned out to work great. Convolutional layers capture local dependencies between tokens, allowing the model to learn patterns associated with common vulnerability signatures.

Max pooling is a layer used in convolutional neural networks to reduce the size of feature maps. It works by sliding a small window, across the feature map and keeping only the maximum value within each window. This way, it scales down the data while preserving the most important signals. The result is a smaller feature map that maintains strong features while discarding less relevant details. This not only speeds up computation but also helps prevent overfitting.

On the other hand, global max pooling is a special case of max pooling where, instead of applying a small window, the operation takes the maximum value across the entire feature map. In other words, it collapses a whole feature map into just a single value. Global max pooling is often used just before the output layer of a network, especially for classification tasks, because it reduces the spatial dimensions completely and leaves only the strongest activation per channel. In *Figure 10,* we can observe the architecture of the CNN model that was used in this research. The results can be seen in the next chapter, *Chapter 6: Experiments and results*.



Embedding
Conv1D (filters=512, kernel_size=5, activation='relu')
MaxPooling1D (pool_size=2)
GlobalMaxPooling1D
Dropout (0.5)
Dense (1, activation='sigmoid')

*Figure 10 – CNN model architecture*

### 5.2.2 CNN + LSTM-Based Models

Long Short-Term Memory (LSTM) networks are a type of Recurrent Neural Network (RNN) capable of modeling sequential dependencies in tokenized code. Their ability to capture long-range context makes them well-suited for code structures where vulnerabilities arise due to interactions across distant lines or blocks. LSTMs are applied to sequences derived from preprocessed ASTs or token streams, providing temporal modeling of code semantics. In *Figure 11*, we can see the model architecture.

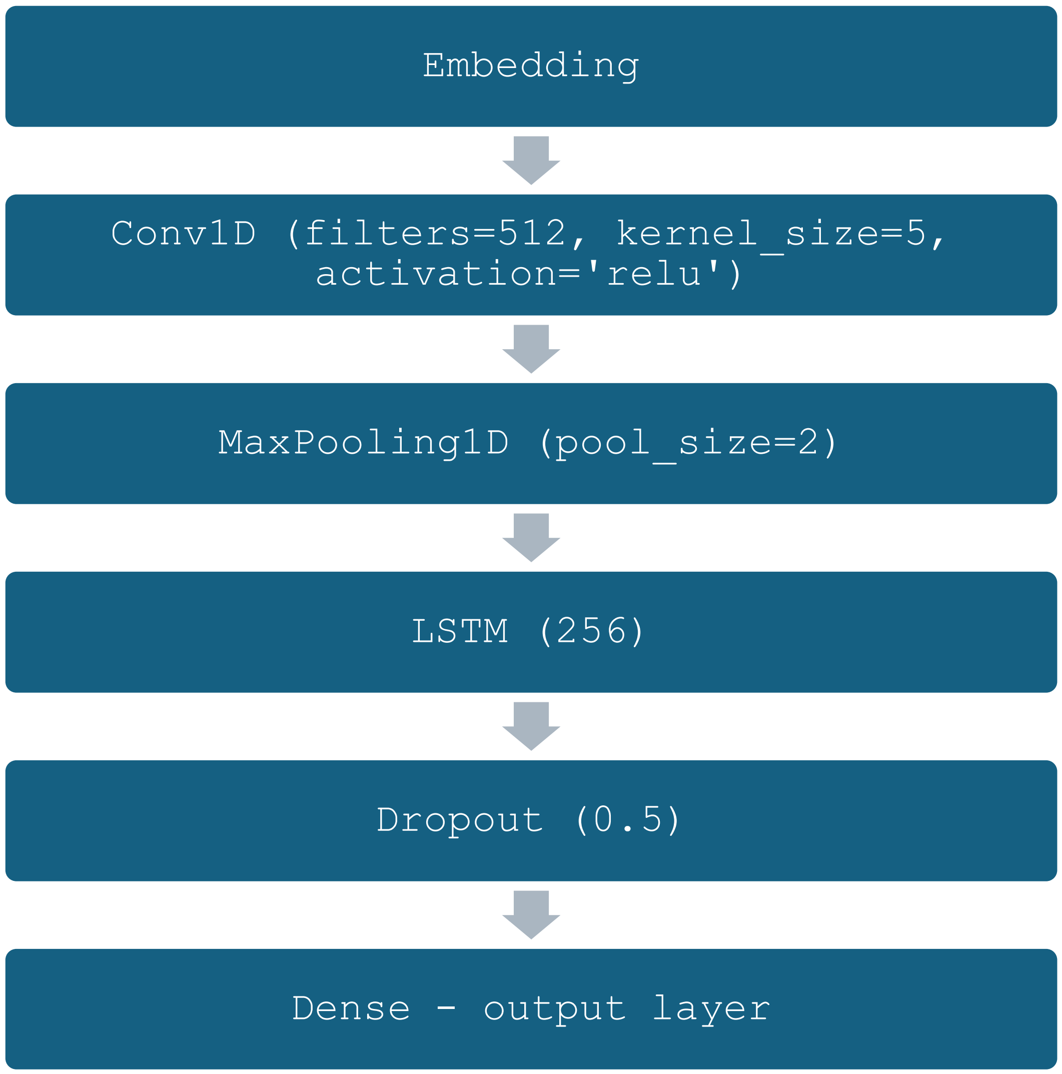

Embedding
Conv1D (filters=512, kernel_size=5, activation='relu')
MaxPooling1D (pool_size=2)
LSTM (256)
Dropout (0.5)
Dense - output layer

*Figure 11 – CNN + LSTM model architecture*

## 5.3. GNN for control flow graphs

Graph Neural Networks (GNNs) represent a recent advancement in learning from non-Euclidean data structures such as graphs. In the realm of program analysis, Control Flow Graphs (CFGs) provide a rich representation of execution paths within a program. Nodes represent basic blocks or instructions, and edges capture control dependencies such as conditional branches or loops.

GNNs operate by propagating and aggregating information across neighboring nodes, enabling the model to learn localized and global graph-level features. This capacity is particularly advantageous for capturing control logic that might reveal complex, path-dependent vulnerabilities. The use of GNNs in this dissertation emphasizes structural reasoning about control flows, going beyond lexical or syntactic token analysis.

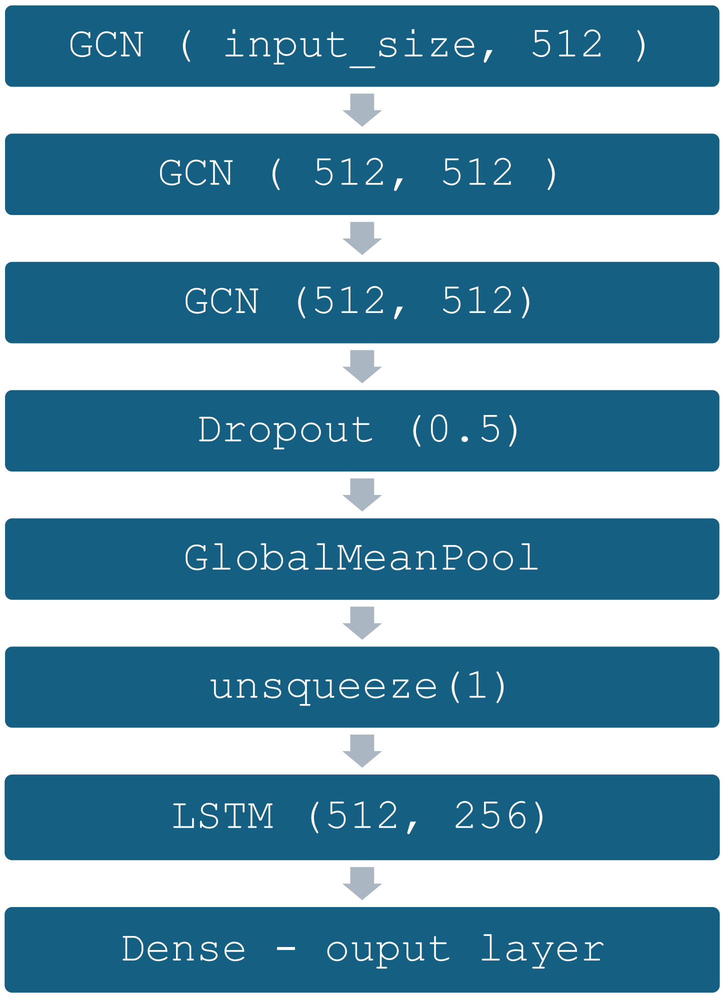


*Figure 12 – GNN model architecture*

The architecture of the model can be seen in *Figure 12*. This model processes graph data by first using 3 stacked GCN layers to learn node representations based on the graph structure. Each GCN layer allows nodes to update their features by gathering information from their neighbors, and stacking multiple layers enables nodes to capture information from farther away in the graph. After the GCNs, dropout is applied to the node features to prevent overfitting and improve generalization.

Next, the model uses global mean pooling to combine all node features into a single fixed-size vector for each graph. This step is necessary because graphs can have different numbers of nodes, and the model needs a consistent representation size for the next stages.

The pooled graph representations are then passed into an LSTM. Even though each graph in this case is treated as a single timestep, the LSTM still provides an additional layer of processing that can model complex dependencies within the graph features or, if extended, capture relationships across sequences of graphs.

After the LSTM processes the input, the final hidden state of the LSTM is used as the summary of the graph's learned representation. This hidden state is fed into a fully connected linear layer to produce the final output, typically a prediction for graph classification. Before this final layer, another dropout is applied to further reduce the risk of overfitting and encourage the model to learn more robust features.

# 6. EXPERIMENTS AND RESULTS

## 6.1. Classical models with text vectorization

The approach used in this research was to use 7 models: Random Forests, Gradient Boosting, Ada Boost, Logistic Regression with L1 regularization, Logistic Regression with L2 regularization, k-Nearest Neighbours and Decision Trees.

| *Dataset* | *Model / Vectorizer* | *Random Forests* | *Gradient Boosting* | *Ada Boost* | *Logistic Regression L1* | *Logistic Regression L2* | *kNN* | *Decision Trees* |
|---|---|---|---|---|---|---|---|---|
| SATE IV Juliet (60.000 samples) | Count Vectorizer | 92% | 85% | 79% | 87% | 88% | 86% | **95%** |
| | TFIDF Vectorizer | 92% | 84% | 78% | 86% | 82% | 84% | 94% |
| | Hashing Vectorizer | 92% | 85% | 79% | 82% | 75% | 86% | 94% |
| DiverseVul + BigVul + MegaVul (~10.000 samples) | Count Vectorizer | **72%** | 62% | 54% | 71% | **72%** | 58% | 64% |
| | TFIDF Vectorizer | **72%** | 61% | 53% | 62% | 67% | 60% | 63% |
| | Hashing Vectorizer | 70% | 61% | 53% | 56% | 58% | 60% | 65% |
| All (~ 70.000 samples) | Count Vectorizer | 89% | 80% | 73% | 84% | 83% | 83% | **91%** |
| | TFIDF Vectorizer | 89% | 80% | 72% | 82% | 80% | 81% | 90% |
| | Hashing Vectorizer | 89% | 80% | 73% | 77% | 71% | 83% | 89% |

*Table 2 – F1 Score of all ML models*

In *Table 2*, we can see the results from training the models. The data was split using train_test_split function, with 80% data for training and 20% for testing. There are 3 datasets. First, the Juliet dataset with 30.000 vulnerable function and 30.000 benign functions. Then, a combination of DiverseVul, BigVul and MegaVul datasets, with ~10.000 function, from which half were vulnerable and half were not vulnerable. Finally, I combined those 2 datasets into a dataset containing both artificial and real-world code functions, which contains nearly 70.000 functions.

The F1 score was considered as the best metric to analyze, and we can talk about the results for each dataset, in order to observe some patterns. For the Juliet dataset, Decision Trees gave the best results, with a F1 score of 95%, followed by Random Forests and kNN. For the second dataset, the best results were achieved by Random Forests and Logistic Regression models, with a F1 score of 72%. The final dataset came close to the Juliet dataset, but had a sligthly lower best F1 score of 91% for the Decision Trees, followed by Random Forests with 89%. The vectorizers did not show a lot of changes in terms of score, but Count Vectorizer was slightly better, followed by TFIDF and lastly by Hashing Vectorizer.

## 6.2. AST-based neural models

The experiments of AST-based neural models were also tested on 3 datasets, as well as the classical ML models. The results can be observed in *Table 3*. The best model for Juliet dataset was CNN + LSTM, with an accuracy of 98.35%, but for the other 2 datasets, CNN model gave better results, with 65.76% and respectively 86.39% accuracy.

| *Dataset* | *CNN* | *CNN + LSTM* |
|---|---|---|
| SATE IV Juliet | 97.35% | **98.35%** |
| DiverseVul + BigVul + MegaVul | **65.76%** | 51.15% |
| All | **86.39%** | 85.30% |

*Table 3 – Accuracy for CNN and CNN + LSTM models*

In the images below, *Figures 13 – 18*, we can see the training process for the CNN and CNN+LSTM models, for each dataset. We can see that the process went smoothly for the Juliet dataset, but it was different for the second dataset, where we can see a lot of spikes in terms of validation loss and accuracy. Finally, for the third dataset, the training worked quite well, with not loss decresing and accuracy increasing up to 85-86%. The problem for the functions in the second dataset was that it was hard for the model to process them, so I could not train the models with a larger number of epochs, because of computation power. The time of training for the Juliet dataset for the ~30 minutes. For the other datasets, the necessary time was about 3-4 hours. The development of the models was done in Google Collaboratory.

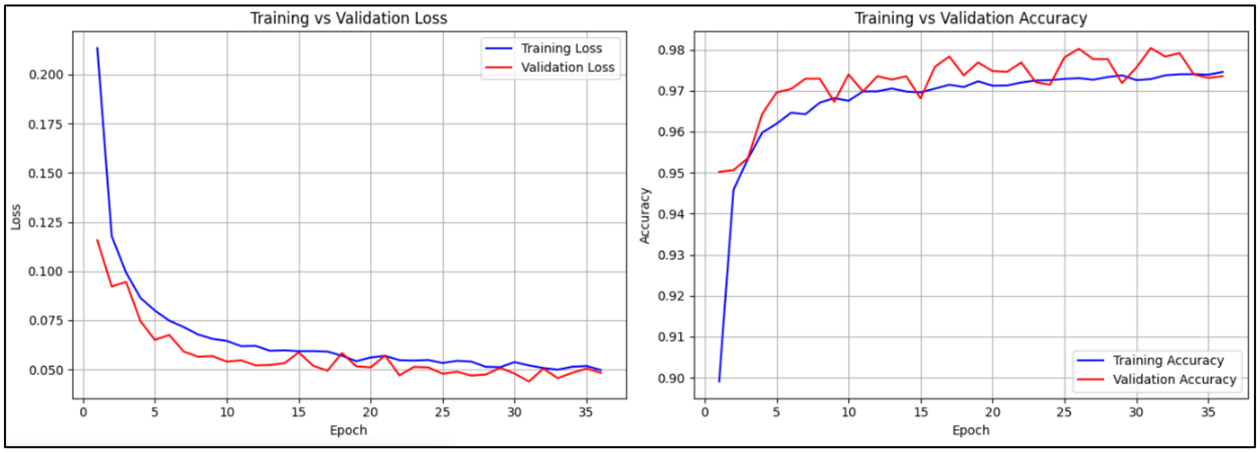


*Figure 13 – Loss and accuracy for CNN on Juliet dataset*

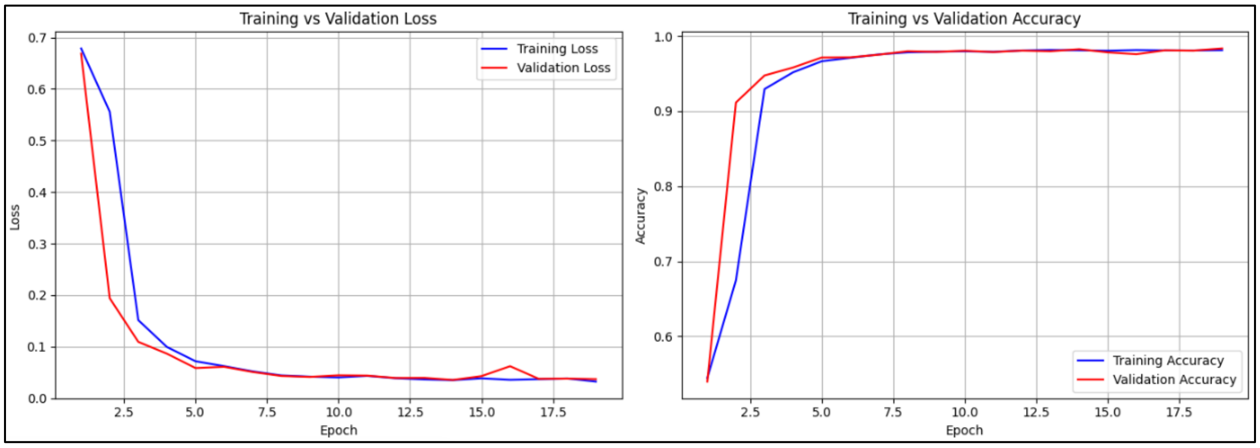


*Figure 14 – Loss and accuracy for CNN + LSTM on Juliet dataset*

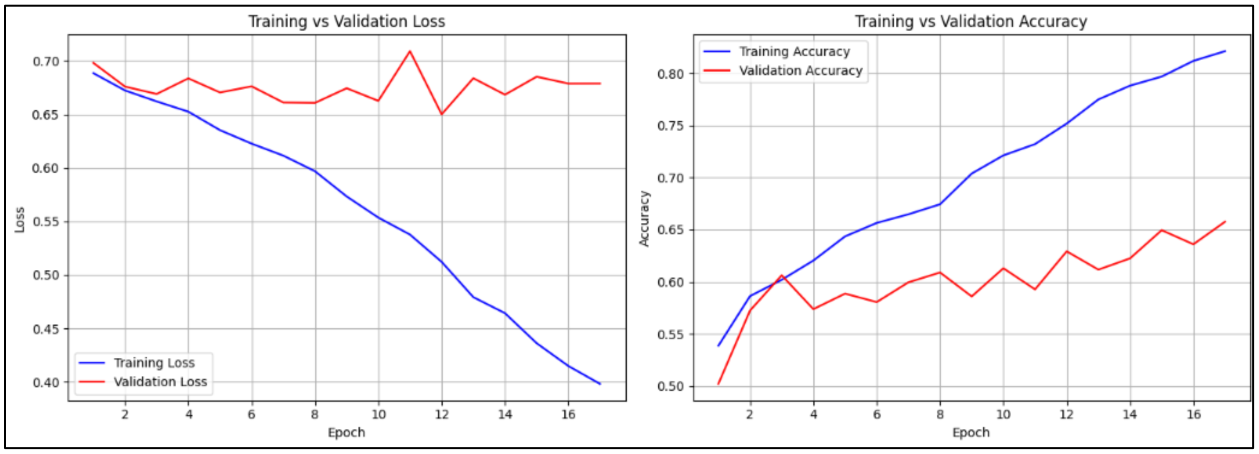


*Figure 15 – Loss and accuracy for CNN on DiverseVul + BigVul + MegaVul dataset*

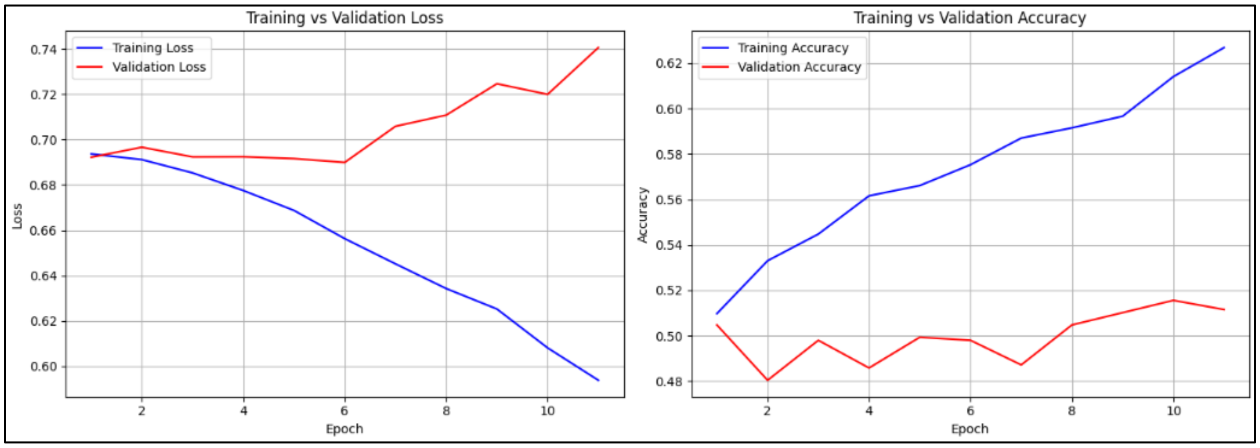


*Figure 16 – Loss and accuracy for CNN + LSTM on DiverseVul + BigVul + MegaVul dataset*

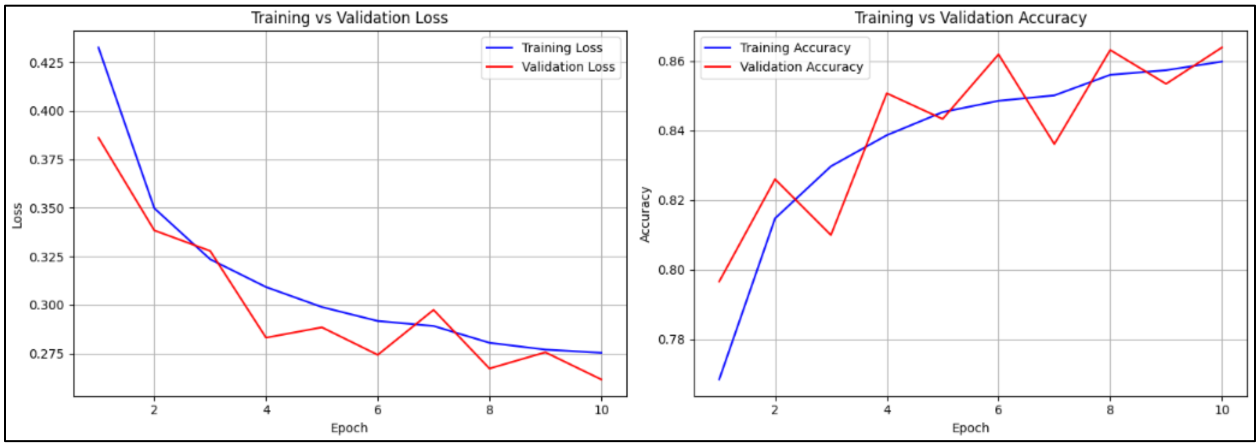


*Figure 17 – Loss and accuracy for CNN on combined dataset*

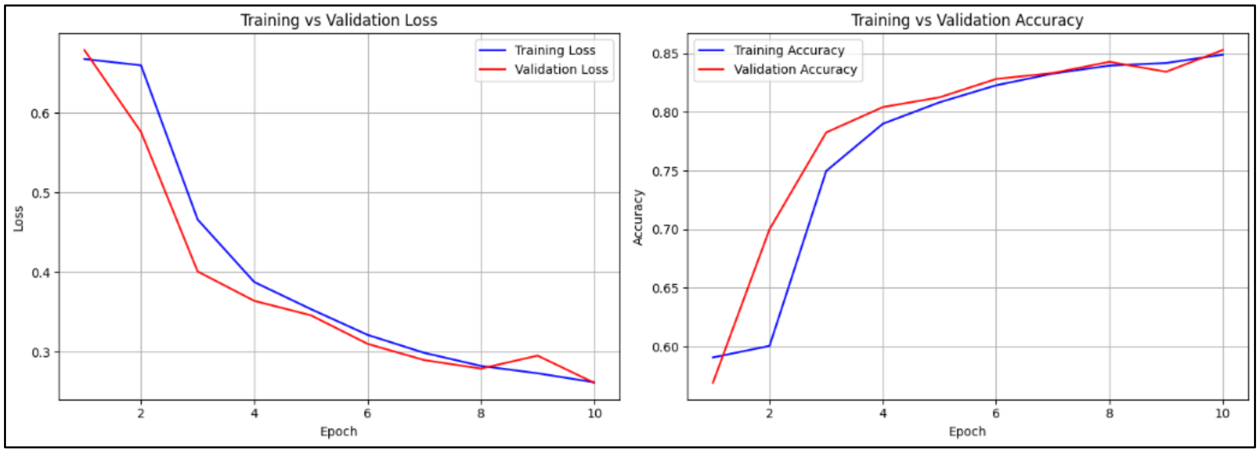


*Figure 18 – Loss and accuracy for CNN + LSTM on combined dataset*

## 6.3. GNN performance on CFGs

The GNN + LSTM models were tested for all 3 datasets. The accuracies can be seen in *Table 4*, below. The best accuracy was on the Juliet dataset, 87.5%. There was a big difference between the real-world and artificial code functions in this training process. In terms of accuracy, the real-world functions from the second dataset had only 55.5% accuracy when used in GNN + LSTM model. When tested on the third dataset, containing all functions, the model had 82.5% accuracy, showing that using an artificial code dataset can improve the performance of a model, but it does not mean that it can really be used in real-world scenarios.

| *Dataset* | *GNN + LSTM* |
|---|---|
| SATE IV Juliet | 87.50% |
| DiverseVul + BigVul + MegaVul | 55.50% |
| All | 82.50% |

*Table 4 – Accuracy for GNN models*

In *Figures 19-21*, we can see the training process for every dataset. The accuracy for the second dataset was also lower because of the number of epochs, which was decresed in order to reduce training time, because the lack of computation power.

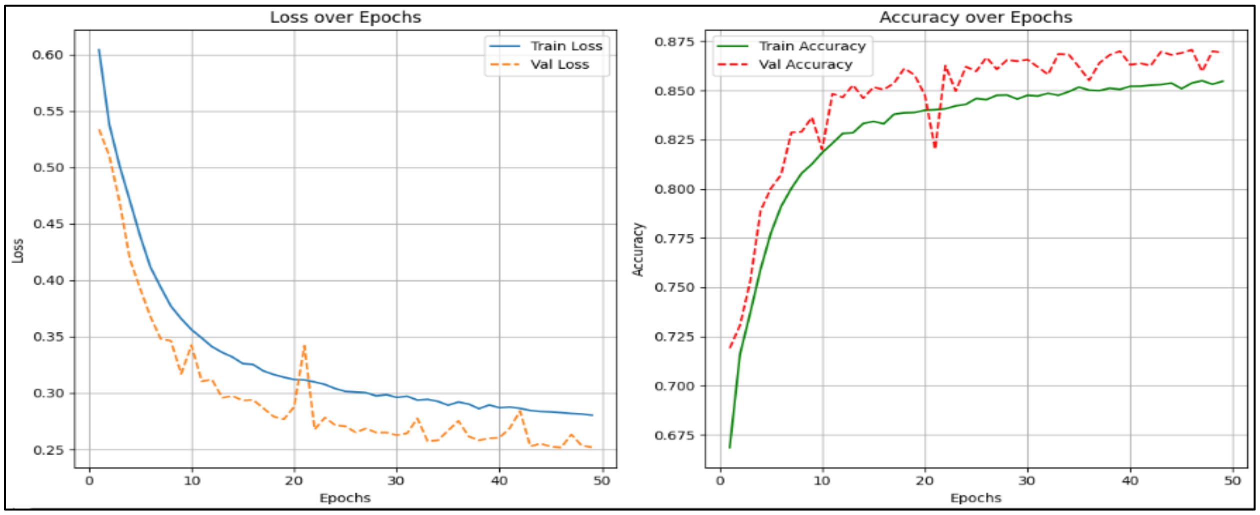


*Figure 19 – Loss and accuracy for GNN + LSTM on Juliet dataset*

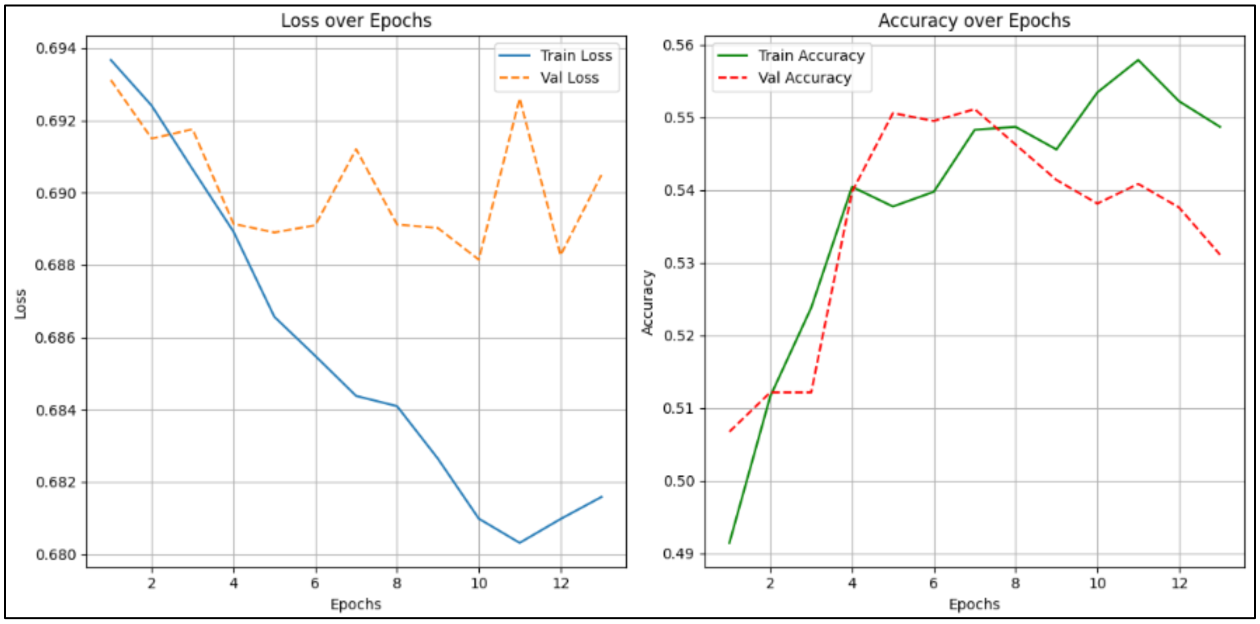


*Figure 20 – Loss and accuracy for GNN + LSTM on DiverseVul + MegaVul + BigVul dataset*

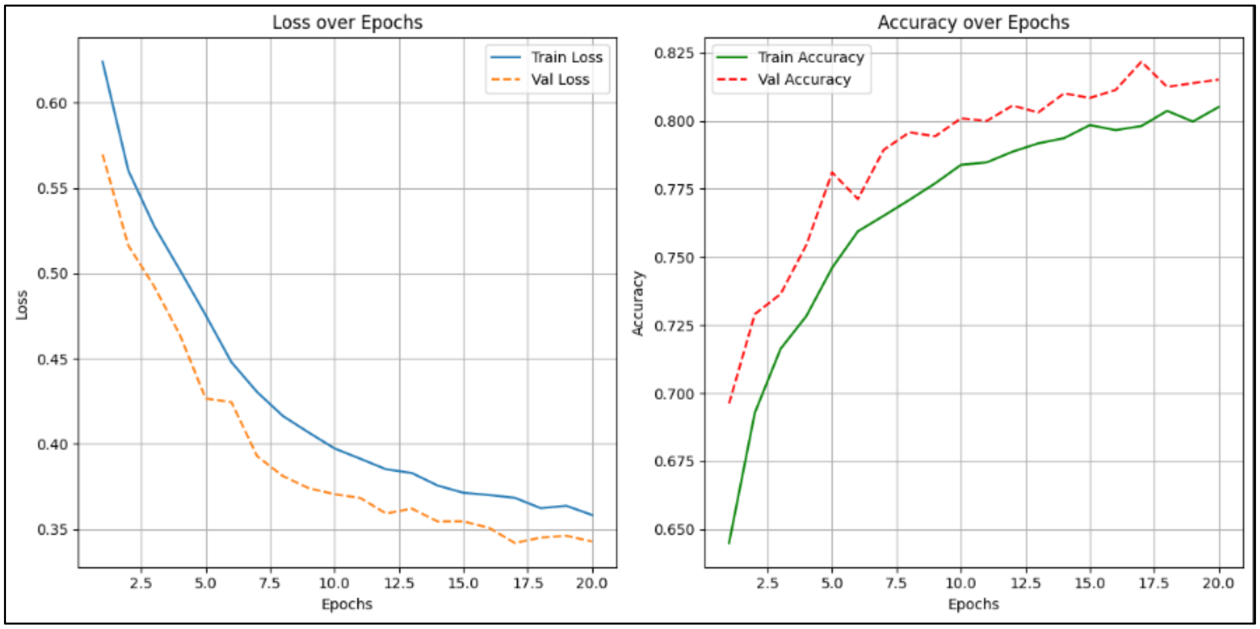


*Figure 21 – Loss and accuracy for GNN + LSTM on combined dataset*

# 7. DISCUSSION

This chapter critically reflects on the findings of the experimental study, analyzing the strengths, weaknesses, interpretability, and practical implications of different machine learning approaches for detecting vulnerabilities in C/C++ source code. The experiments shown in *Chapter 6* can be seen in different ways, and there are some questions to be asked. Machine learning and deep learning models can be interpreted in various ways, and in order to improve the real-world vulnerability detection from source code, we need to ask ourself questions and think about answers. The main questions for this study are explained below.

1. Are classical machine learning models better for vulnerability detection than deep learning techniques?

The experimental results suggest that classical machine learning models, such as Decision Trees, Random Forests, and Gradient Boosting, offer several advantages when applied to vulnerability detection tasks, particularly in scenarios where interpretability, lower computational cost, and limited data are important factors. Models utilizing TF-IDF or Hashing vectorizers achieved competitive precision and recall, and were relatively straightforward to train and deploy. This finding is consistent with the work of Chernis and Verma (2018) and Shen (2018), who argue that simpler models can yield practical, if not optimal, solutions for security applications.

However, deep learning models, particularly CNN and LSTM-based architectures operating on Abstract Syntax Trees, demonstrated superior performance in learning complex patterns associated with vulnerabilities. These models excelled at capturing intricate syntactic and semantic relationships that classical methods could not fully exploit. Chakraborty et al. (2021) and Mazuera-Rozo et al. (2021) similarly observed that deep learning approaches, given sufficient training data and computational resources, can detect subtle code weaknesses that traditional models may overlook. Nevertheless, deep learning requires larger labeled datasets, incurs greater training and inference times, and presents challenges in terms of model interpretability. Thus, the superiority of one approach over the other is context-dependent: for lightweight, interpretable analysis, classical models are preferable; for exhaustive, pattern-rich detection, deep learning models provide a substantial advantage.

2. Are AST based models better than CFG based models for understanding the code functions?

When comparing Abstract Syntax Trees (ASTs) and Control Flow Graphs (CFGs) as input representations, the experiments reveal nuanced insights. AST-based models, particularly those employing CNNs and LSTMs, performed remarkably well in learning the structural and syntactic features of source code. This result is coherent with the intuition that ASTs naturally encode the grammatical structure of programs, thereby allowing models to learn semantic patterns linked to vulnerabilities, as also indicated in the studies by Fan et al. (2020) and Russell et al. (2018).

On the other hand, models based on CFGs, primarily utilizing Graph Neural Networks (GNNs), offered advantages in reasoning over the dynamic execution flow of programs. CFG-based models were effective in capturing control dependencies and execution paths that AST-based models might miss. The research by Grieco et al. (2016) and Hanif et al. (2021) similarly emphasized that CFG representations, though more complex to construct and process, can be crucial for identifying vulnerabilities related to logic errors, such as incorrect conditionals or unhandled exceptions.

Thus, ASTs are generally more effective for capturing structural code properties, while CFGs are better suited for understanding operational behavior. A hybrid approach that combines insights from both representations could yield even better detection capabilities, an avenue recommended for future research.

3. Can we use machine learning in order to simplify the process of vulnerability detection?

The findings of this study strongly suggest that machine learning can indeed simplify parts of the vulnerability detection process. By automating the identification of risky patterns, dangerous code constructs, and anomalous behavior in source code, machine learning models can significantly reduce the manual effort required for static analysis and code auditing.

However, the notion of "simplification" must be nuanced. Machine learning models, particularly deep learning architectures and GNNs, require considerable expertise to develop, train, and maintain. Moreover, the black-box nature of many powerful models complicates their use in critical security workflows where explainability is required. As highlighted by Allamanis et al. (2018) and Vándor et al. (2022), practical deployments of ML-based vulnerability detection often combine traditional rule-based static analysis tools with machine learning models to strike a balance between coverage and interpretability.

Therefore, while machine learning does simplify the detection phase from an end-user perspective by surfacing vulnerabilities faster and with reasonable accuracy, it also introduces new layers of complexity in terms of model governance, updating, and integration with existing security processes. The adoption of these models must be accompanied by robust engineering practices, transparency mechanisms, and human oversight to fully realize their benefits without introducing unacceptable risks.

4. How does the quality and labeling of datasets impact vulnerability detection models?

Since machine learning depends heavily on training data, the way vulnerabilities are labeled, and how "realistic" the examples are (synthetic vs. real-world code), can massively influence model performance. Fan et al. (2020) and Chen et al. (2023) discuss this — synthetic datasets like Juliet are useful but can lead to overfitting, while real-world datasets are messier and harder to learn from.

5. What is the role of explainability in machine learning-based vulnerability detection?

In security, it is not enough to predict that code is vulnerable — security engineers need to understand *why* it is flagged. Classical models (like Decision Trees) are more explainable, but deep learning (CNNs, LSTMs, GNNs) are often black boxes. This tension is discussed in papers like Chakraborty et al. (2021) and is critical for adoption in professional software engineering workflows.

6. How well do machine learning models generalize across different C/C++ projects?

A model trained on one type of code (e.g., open-source libraries) might perform poorly when applied to another (e.g., embedded systems or proprietary code). Generalization is a major challenge, discussed in Akter et al. (2022) and Harer et al. (2018).

7. Can adversarial techniques compromise vulnerability detection models?

Attackers could intentionally "camouflage" vulnerabilities so that ML models fail to detect them (e.g., via code obfuscation or adversarial examples). Although your dissertation focuses on normal detection, mentioning this shows awareness of a very active and important area (Russell et al., 2018 briefly hint at this).

# 8. CONCLUSIONS

This dissertation explored the application of machine learning approaches for detecting vulnerabilities in C/C++ source code, combining classical machine learning models, deep learning techniques based on Abstract Syntax Trees (ASTs), and Graph Neural Networks (GNNs) leveraging Control Flow Graphs (CFGs). Through a detailed experimental study using both synthetic datasets, such as Juliet, and complex real-world datasets, the research aimed to assess the strengths, limitations, and practical implications of each method in the domain of software vulnerability detection.

The results reveal that no single approach uniformly outperforms others across all criteria. Classical machine learning models, such as Decision Trees and Random Forests, demonstrated solid performance with relatively high interpretability, particularly when using textual feature extraction methods like TF-IDF or Hashing vectorizers. These models benefit from faster training times and simpler deployment pipelines, making them suitable for certain industry applications where explainability is crucial.

In contrast, deep learning models that utilize AST representations, including CNNs and LSTMs, showed stronger capabilities in capturing complex syntactic patterns and structural nuances of source code. They achieved higher accuracy in many cases, particularly on real-world datasets, but at the cost of interpretability and significantly higher computational requirements. Meanwhile, GNNs operating on CFGs demonstrated promising results for representing the dynamic aspects of program execution paths, although their effectiveness highly depended on the quality and completeness of the generated graphs.

The study highlights several key insights. First, feature engineering plays a critical role; richer and more structured code representations typically lead to better detection performance. Second, the quality and realism of datasets significantly influence model generalization ability; models trained purely on synthetic data may underperform on real-world examples. Third, explainability remains an open challenge, especially for deep learning models, suggesting the need for more research into interpretable machine learning in security contexts.

Several limitations of the current work must also be acknowledged. The focus on C/C++ source code means that findings may not directly generalize to other programming languages. Additionally, the assumption of clean and labeled data does not always reflect the messy realities of industrial software development environments. Finally, the models were evaluated without considering adversarial examples or obfuscated code, which represent important directions for future research.

In conclusion, machine learning approaches offer significant potential for advancing automated vulnerability detection, but their adoption must be guided by careful consideration of model interpretability, computational cost, and robustness to real-world complexities. Future work should explore cross-language generalization, integrate dynamic analysis features, enhance adversarial robustness, and investigate hybrid models that combine classical and deep learning paradigms to further improve the practicality and trustworthiness of automated vulnerability detection systems.

# REFERENCES

1. Akter, M. S., Shahriar, H., & Bhuiya, Z. A. (2022, December). Automated vulnerability detection in source code using quantum natural language processing. In International Conference on Ubiquitous Security (pp. 83–102). Singapore: Springer Nature Singapore.
2. Allamanis, M., Barr, E. T., Devanbu, P., & Sutton, C. (2018). A survey of machine learning for big code and naturalness. ACM Computing Surveys (CSUR), 51(4), 1–37.
3. Bilgin, Z., Ersoy, M.A., Soykan, E.U., Tomur, E., Çomak, P. and Karaçay, L., 2020. Vulnerability prediction from source code using machine learning. IEEE Access, 8, pp.150672-150684
4. Campbell, G. A., & Papapetrou, P. P. (2013). *SonarQube in action*. Manning Publications Co..
5. Chakraborty, S., Krishna, R., Ding, Y., & Ray, B. (2021). Deep learning based vulnerability detection: Are we there yet? IEEE Transactions on Software Engineering, 48(9), 3280–3296.
6. Chen, Y. (2015). *Convolutional neural network for sentence classification* (Master's thesis, University of Waterloo).
7. Chen, Y., Ding, Z., Alowain, L., Chen, X., & Wagner, D. (2023, October). Diversevul: A new vulnerable source code dataset for deep learning based vulnerability detection. In Proceedings of the 26th International Symposium on Research in Attacks, Intrusions and Defenses (pp. 654–668).
8. Chernis, B., & Verma, R. (2018, March). Machine learning methods for software vulnerability detection. In Proceedings of the fourth ACM international workshop on security and privacy analytics (pp. 31–39).
9. Fan, J., Li, Y., Wang, S., & Nguyen, T. N. (2020, June). A C/C++ code vulnerability dataset with code changes and CVE summaries. In Proceedings of the 17th international conference on mining software repositories (pp. 508–512).
10. Hanif, H., Nasir, M. H. N. M., Ab Razak, M. F., Firdaus, A., & Anuar, N. B. (2021). The rise of software vulnerability: Taxonomy of software vulnerabilities detection and machine learning approaches. Journal of Network and Computer Applications, 179, 103009.
11. Harer, J. A., Kim, L. Y., Russell, R. L., Ozdemir, O., Kosta, L. R., Rangamani, A., ... & Lazovich, T. (2018). Automated software vulnerability detection with machine learning. arXiv preprint arXiv:1803.04497.

12. Knutsen, M., & Lervik, E. H. (2022). Detection of Vulnerabilities in Source Code Using Machine Learning and Natural Language Processing (Master's thesis, NTNU).
13. Kremenek, T. (2008). Finding software bugs with the clang static analyzer. *Apple Inc*, *8*, 2008.
14. Li, Z., Zou, D., Xu, S., Ou, X., Jin, H., Wang, S., Deng, Z. and Zhong, Y., 2018. Vuldeepecker: A deep learning-based system for vulnerability detection. arXiv preprint arXiv:1801.01681.
15. Mazuera-Rozo, A., Mojica-Hanke, A., Linares-Vásquez, M., & Bavota, G. (2021, May). Shallow or deep? an empirical study on detecting vulnerabilities using deep learning. In 2021 IEEE/ACM 29th International Conference on Program Comprehension (ICPC) (pp. 276–287). IEEE.
16. Medeiros, N., Ivaki, N., Costa, P. and Vieira, M., 2020. Vulnerable code detection using software metrics and machine learning. IEEE Access, 8, pp.219174-219198.
17. Nethercote, N., & Seward, J. (2007). Valgrind: a framework for heavyweight dynamic binary instrumentation. *ACM Sigplan notices*, *42*(6), 89-100.
18. Ni, C., Shen, L., Yang, X., Zhu, Y., & Wang, S. (2024, April). MegaVul: AC/C++ vulnerability dataset with comprehensive code representations. In *Proceedings of the 21st International Conference on Mining Software Repositories* (pp. 738-742).
19. Okun, V., Delaitre, A., & Black, P. E. (2013). Report on the static analysis tool exposition (sate) iv. *NIST Special Publication*, *500*, 297.
20. Rajapaksha, S., Senanayake, J., Kalutarage, H., & Al-Kadri, M. O. (2022, December). AI-powered vulnerability detection for secure source code development. In International Conference on Information Technology and Communications Security (pp. 275–288). Cham: Springer Nature Switzerland.
21. Russell, R., Kim, L., Hamilton, L., Lazovich, T., Harer, J., Ozdemir, O., ... & McConley, M. (2018, December). 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.
22. Serebryany, K., Bruening, D., Potapenko, A., & Vyukov, D. (2012). {AddressSanitizer}: A fast address sanity checker. In *2012 USENIX annual technical conference (USENIX ATC 12)* (pp. 309-318).
23. Shen, X. (2018). Predicting vulnerable files by using machine learning method (Doctoral dissertation, MS thesis, Dept. Elect. Eng., Math. Comput. Sci. (EWI), Delft Univ. Technol., Delft, The Netherlands).
24. Vándor, N., Mosolygó, B., & Hegelűs, P. (2022, July). Comparing ML-Based Predictions and Static Analyzer Tools for Vulnerability Detection. In International Conference on Computational Science and Its Applications (pp. 92–105). Cham: Springer International Publishing.
25. Yamaguchi, F., & Rieck, K. (2011). Vulnerability extrapolation: Assisted discovery of vulnerabilities using machine learning. In 5th USENIX workshop on offensive technologies (WOOT 11).

# RESEARCH CODE REPOSITORY

The complete source code developed for this dissertation is available online at: https://github.com/TaviFurdui/detecting-vulnerabilities-using-ml

The repository contains all scripts, datasets, and instructions necessary to reproduce the results discussed in this dissertation.