Detecting Vulnerabilities In C/C++ Source Code Using Machine Learning Approaches

Sorin-Octavian Furdui
Master of Machine Learning
Politehnica University of Timişoara
Timişoara, Timiş, Romania
sorin.furdui@student.upt.ro

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 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, using multiple vectorization techniques. 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.

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

I. INTRODUCTION

A. Problem statement and objectives

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 largescale or real-world codebases efficiently. Considering the advances of machine learning, particularly deep learning, new methods that can automatically learn from large code repositories have emerged. However, there are still questions regarding the performance, reliability and generalization capabilities of these models when applied to diverse programming languages 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 on well-established datasets.

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 and deep learning models—such as Decision Trees, Random Forests, kNN, Logistic Regression, Boosting, Convolutional Neural Networks (CNNs), Long Short-Term Memory networks (LSTMs) and Graph Neural Networks (GNNs).

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, F1 score, interpretability, and computational complexity.

B. Structure of the paper

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

Chapter II - 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.

Chapter III - 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, such as DiverseVul, MegaVul and BigVul.

Chapter IV - 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 V - Experiments: In this chapter, the results of experiments made using the different models and feature sets are presented and compared. Performance metrics such as precision, 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 VI: Conclusion: The final chapter summarizes the key findings and the limitations of the research. 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.

II. LITERATURE REVIEW

A. 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 [1] and SonarQube [2] 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. Tools such as Valgrind [3] and AddressSanitizer [4] 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.

The research presented in [5] 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.

B. Machine Learning in Software Security

Recent advancements in machine learning 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. Studies such as [6] and [7] laid the groundwork by using ML for source code modeling and vulnerability classification. 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 [8].

One of the papers [9] that made some valuable contribution in this field, 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, improving detection capabilities compared to traditional static analysis tools.

Another paper that is well known for contributions in this field is [10], where a model named VulDeePecker is built. The model is 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. 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.

The study presented in [11] provides 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) and OWASP Top 10. 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.

The work shown in [12] utilizes 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 [13]. 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.

Another type of approach, involving quantum computing, could be observed in [14]. 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.

The research written by [15] 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. 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.

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 [16] and efforts by [17] 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 [17], 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.

III. DATASETS

The scope of the research includes the analysis of both synthetic datasets, such as Juliet SATE IV [18] and complex real-world vulnerability datasets, such as DiverseVul [16], BigVul [19] and MegaVul [20]. The study is grounded in supervised learning methodologies, relying on labeled data. The final dataset contains samples from all those 4 datasets. Most of them (nearly 85%) are syntethic functions from Juliet SATE IV. The other 15% are real-world code functions from project such as Linux, Wireshark, TCPDump, from the other 3 datasets. 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.

A. Juliet SATE IV Dataset

The Juliet Test Suite [18] 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.

B. Real-World Functions Dataset

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. 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.

C. Exploratory Data Analysis

In the figures below, there is presented some exploratory data analysis. In Fig. 1, there is a wordcloud with most used words from the final dataset. We can observe words like data, null, return, break, char or int. These were the words that were mostly used across the final dataset.

In Fig. 2, there is a distribution of function length across final dataset, with a hue for vulnerability (0 means clean, 1 means vulnerable). We can observe that there are more clean functions than vulnerable ones, but the distribution in terms of function length is following the same pattern for both vulnerable and benign functions. A more balanced approach could be achieved by adding more vulnerable functions or by reducing the number of clean functions. I decided to reduce the number of clean functions, in order to obtain a more balanced final dataset.

In addition, we can observe the final dataset structure in the pie chart from Fig. 3. The number of clean and vulnerable functions is the same. The number of complex real-world code functions was reduced to nearly 10.000 in order to have a smaller training time, because of their high complexity. Also, I think that using more synthetic functions could improve the accuracy of the models for the real-world functions. The models could learn from simple synthetic code and apply the findings in complex code.

Finally, looking to Table 1, it 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. These features are usefull in order to compare the 2 types of datasets. We can observe that real-world code functions are usually larger, with an average number of lines of 43.21 lines, whereas synthetic functions have a mean equal to 26.62. The complexity is also greater for the real-world code, with a maximum of 120, while synthetic functions have a max of 45.

Fig. 1. Wordcloud of most used words for combined dataset

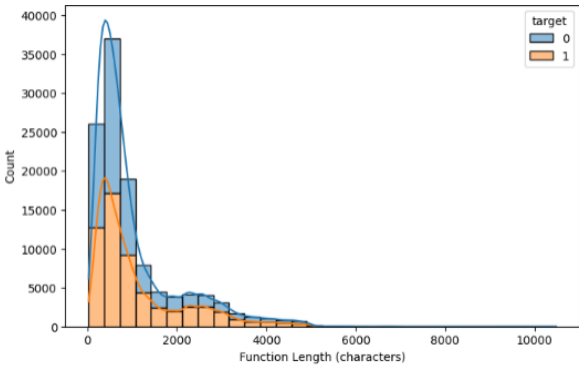

Fig. 2. Distribution of function length for combined dataset

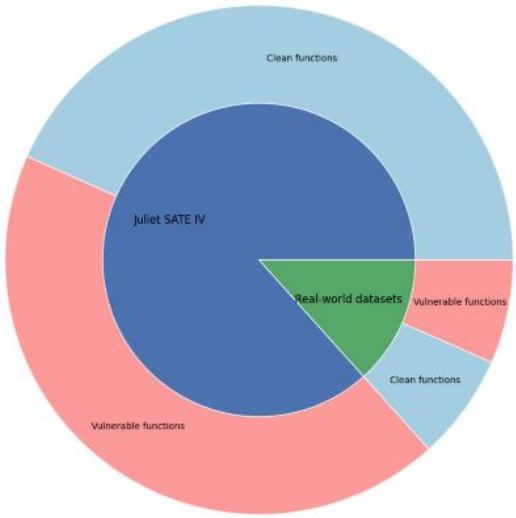

Fig. 3. Pie chart of final dataset

TABLE I. STATISTICAL FEATURES FOR ALL DATASETS

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

IV. METHODOLOGY

A. Feature Engineering

Feature engineering plays a crucial 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 two primary categories of features explored in this research: 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.

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.

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 Fig. 4. 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.

Control Flow Graphs (CFGs), in contrast, emphasize the execution logic of a program, as shown below in Fig. 5. 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 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.

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.

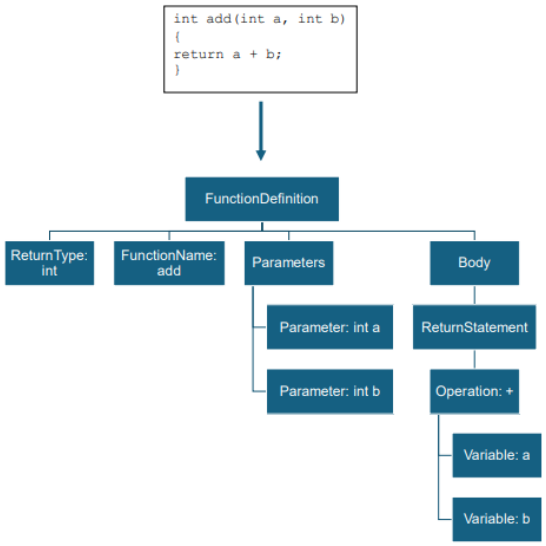

Fig. 4. Function code to AST structure

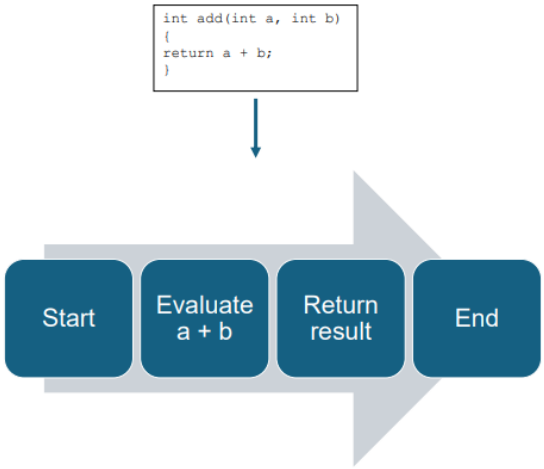

Fig. 5. Function code to CFG structure

B. Classical Machine Learning Techniques

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 [21], 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 and shows that ML 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 [22]. 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. 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.

Another important point of view is that ensemble models such as Decision Trees, Random Forests or boosting models such as Gradient Boosting and AdaBoost could work well with AST-like structures of code. Also, Logistic Regression is a supervised learning algorithm widely used for binary classification problems. 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.

C. Deep Learning Techniques

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.

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 [23], 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 as a great solution. 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 whilediscarding 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.

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 Fig. 6, 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.

Long Short-Term Memory [24] (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. In combination with CNN layers, LSTMs are applied to sequences derived from preprocessed ASTs or token streams, providing temporal modeling of code semantics. In Fig. 7, we can see the model architecture.

Graph Neural Networks [25] (GNNs) represent a recent advancement in learning from nonEuclidean 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 paper emphasizes structural reasoning about control flows, going beyond lexical or syntactic token analysis.

The architecture of the model can be seen in Fig. 8. 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 fixedsize 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.

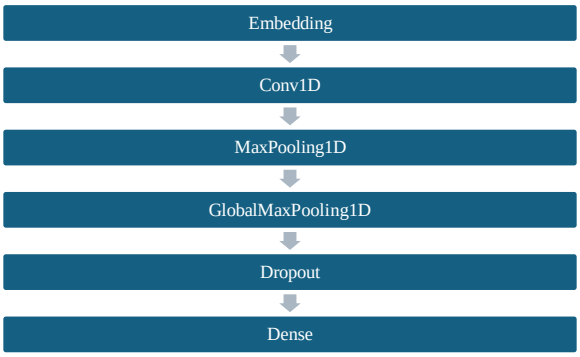

Fig. 6. Architecture of CNN model

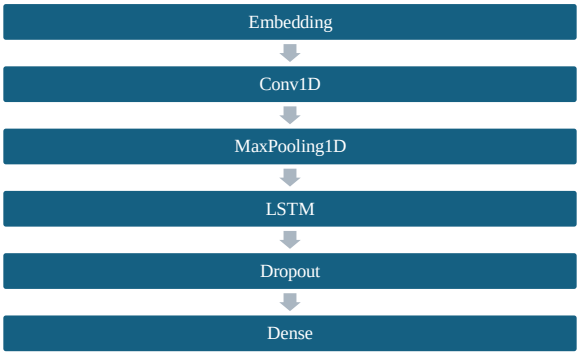

Fig. 7. Architecture of CNN + LSTM model

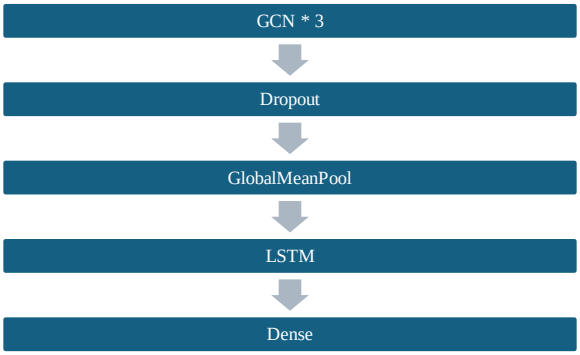

Fig. 8. Architecture of GNN + LSTM model

V. EXPERIMENTS

A. Classical Machine Learning Techniques

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.

In Table II, 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 clean. This can be seen also in Fig. 3. Finally, I combined those two datasets into a dataset containing both syntethic and real-world code 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.

B. CNN + LSTM

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 III. 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.

C. GNN + LSTM

The GNN + LSTM models were tested for all 3 datasets. The accuracies can be seen in Table IV, 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

TABLE II. F1 SCORE FOR ALL CLASSICAL ML MODELS

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 III. ACCURACY FOR CNN AND CNN + LSTM MODELS

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

TABLE IV. ACCURACY FOR GNN + LSTM MODELS

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

VI. CONCLUSIONS

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 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 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.

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.

REFERENCES

[1] Kremenek, T. (2008). Finding software bugs with the clang static analyzer. Apple Inc, 8, 2008.

[2] Campbell, G. A., & Papapetrou, P. P. (2013). SonarQube in action. Manning Publications Co.

[3] Nethercote, N., & Seward, J. (2007). Valgrind: a framework for heavyweight dynamic binary instrumentation. ACM Sigplan notices, 42(6), 89-100

[4] 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).

[5] 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.

[6] 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.

[7] 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. 45

[8] 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.

[9] 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.

[10] 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.

[11] 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.

[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] Yamaguchi, F., & Rieck, K. (2011). Vulnerability extrapolation: Assisted discovery of vulnerabilities using machine learning. In 5th USENIX workshop on offensive technologies (WOOT 11).

[14] 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.

[15] 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

[16] 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).

[17] 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.

[18] Okun, V., Delaitre, A., & Black, P. E. (2013). Report on the static analysis tool exposition (sate) iv. NIST Special Publication, 500, 297.

[19] 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).

[20] 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).

[21] 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).

[22] 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.

[23] Chen, Y. (2015). Convolutional neural network for sentence classification (Master's thesis, University of Waterloo).

[24] Hochreiter, S., & Schmidhuber, J. (1997). Long short-term memory. *Neural computation*, *9*(8), 1735-1780.

[25] Scarselli, F., Gori, M., Tsoi, A. C., Hagenbuchner, M., & Monfardini, G. (2008). The graph neural network model. *IEEE transactions on neural networks*, *20*(1), 61-80.