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 solving 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. 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. ASTs are processed using convolutional and recurrent neural networks (CNN and RNN) and Control Flow Graphs (CFGs) in combination with Graph Neural Networks (GNNs) are used in order to capture complex relationships within code. To ensure a comprehensive evaluation, I incorporated various datasets that range in complexity. 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, control flow graph, neural networks

I. INTRODUCTION

The detection of software vulnerabilities in C/C++ code remains a complex challenge in software security. Traditional static and dynamic analysis tools are effective in some scenarios, but often generate false positives and require expert interpretation. Considering the advances of machine learning, particularly deep learning, new methods that can automatically learn from large code repositories have emerged. This means there are also questions regarding the performance, generalization capabilities and reliability of these models when applied to real-world function cases.

Additionally, there is a lack of standardization in datasets, feature engineering techniques and evaluation metrics, making it difficult to 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.

II. LITERATURE REVIEW

A. Static and Dynamic Code Analysis Techniques

Firstly, 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 functions and bugs.

Then, dynamic analysis, on the other hand, means executing the code and monitoring its behavior under different kind of inputs. Tools such as Valgrind [3] and AddressSanitizer [4] are well-known in this field. However, dynamic analysis can be time-consuming and might not cover all execution paths, leading to incomplete results.

The research presented in [5] showed a comparative analysis of machine learning-based vulnerability detection methods versus traditional static analysis tools. Their paper highlighted the strengths and limitations of both approaches when applied to C and C++ source code. While analyzers are rule-based, machine learning models can adapt to diverse coding styles and learn complex patterns from labeled data.

B. Machine Learning in Software Security

Machine learning 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 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. 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.

The study presented in [11] provides a comprehensive review of software vulnerability detection techniques with focus on machine learning approaches. The authors present machine learning techniques, analyzing both models like decision trees, support vector machines, and random forests, as well as deep learning models, CNNs and RNNs. Each technique is examined in terms of how it represents code features (like tokens, syntax trees, and semantic information) 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 different algorithms, including ML classifiers and deep learning models like 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 boosts the performance of ML models in vulnerability detection tasks, especially when applied to code datasets annotated with 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 research written by [13] 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. The challenges of this approach are the same as presented in other papers. 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 [14] and [15] 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 [15], 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 [16] and complex real-world vulnerability datasets, such as DiverseVul [14], BigVul [17] and MegaVul [18]. 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. By combining both types of datasets, this study ensures a robust evaluation across the spectrum of code environments.

The Juliet Test Suite [16] 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.

In Fig. 1, 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. I decided to reduce the number of clean functions, in order to obtain a more balanced final dataset. 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.

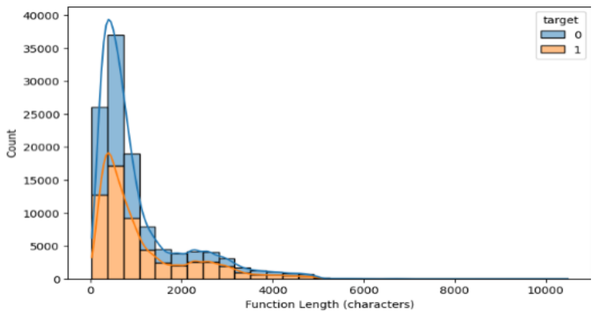

Fig. 1. Distribution of function length for combined dataset

IV. METHODOLOGY

A. Feature Engineering

This chapter describes two primary categories of features used in this research: textual vectorization techniques and structural features derived from compiler representations such as ASTs and CFGs.

In the context of vulnerability detection, textual vectorization can be seen 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 approach uses 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. ASTs model the syntactic structure of source code in a hierarchical tree format, like the structure presented below in Fig. 2. CFGs, in contrast, emphasize the execution logic of a program, as shown below in Fig. 3. 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 becasuse CFGs propagate information between interconnected nodes and capture contextual dependencies. 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.

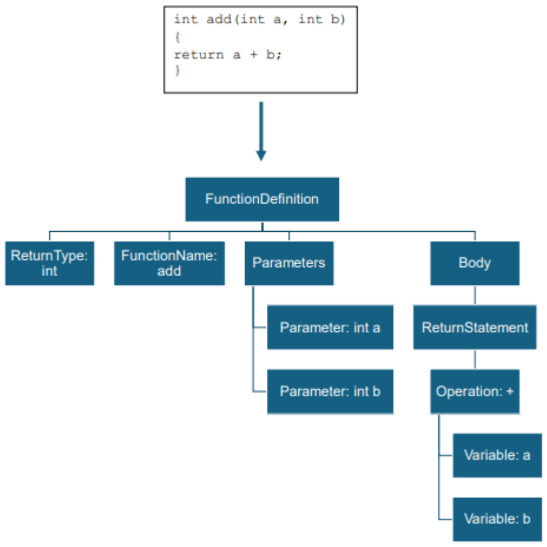
```
int add(int a, int b)
{
return a + b;
}
```

Fig. 2. Function code to AST structure

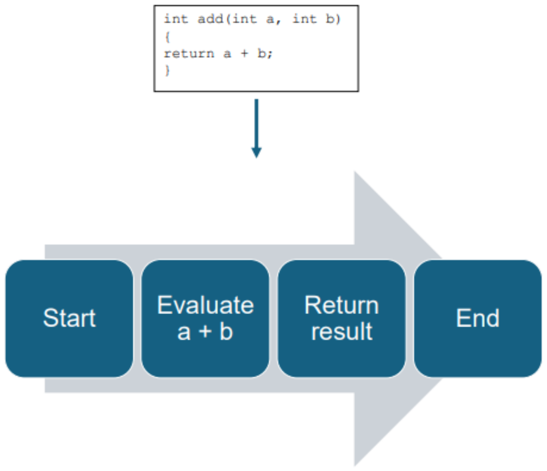
```
int add(int a, int b)
{
return a + b;
}
```

Fig. 3. 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 [19], 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 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.

C. Deep Learning Techniques

Deep learning models are particularly good at capturing non-linear patterns in data. While trained on structural representations such as ASTs, which encode the syntactic structure of code statements and expressions, or CFGs, which encode the process of code execution, they gave good performances, as shown in Chapter 2: Literature Review.

CNNs, originally developed for image recognition, have been adapted to process token sequences extracted from AST traversals or flattened code snippets. As shown in [20], 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 the concept of 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. In Fig. 4, 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.

LSTM [21] networks are a type of RNN capable of modeling sequential dependencies in tokenized code. In combination with CNN layers, LSTMs are applied to sequences derived from preprocessed ASTs or token streams, providing modeling of code semantics. In Fig. 5, we can see the model architecture.

GNNs [22] 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. 6. 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 adding multiple layers enables nodes to capture information from farther away in the graph. The decision to use 3 layers was made in order to have a handshake between time for training and performance. After the GCNs, dropout is applied to the node features to prevent overfitting. Next, the model uses global mean pooling to combine all node features into a single vector of fixed size for each graph. The pooled graph representations are then passed into an LSTM. The LSTM provides an additional layer of processing that can model complex dependencies within the graph features. After the LSTM processes the input, the final hidden state of the LSTM is passed 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, which uses sigmoid activation to generate a label.

For the AST models, the input for the neural networks consisted of words from AST tokenized using the Tokenizer library from Keras. The optimizer used was Adam and the loss function was binary crossentropy. The models were trained for 10 epochs, using Tensorflow Python library. The activation for CNN layers was ReLU. The parameters for each layer can be seen in the Figures.

On the other hand, the GNN model was trained for 20 epochs, using Torch Python library. The input of the network consisted of edges and nodes of the graphs. The optimizer was Adam, and the loss function was binary crossentropy, the same as the CNN models.

All 3 models were trained on Google Collaboratory and the training for each one took ~3 hours. More resources for would have made a difference in training time and accuracy for deep learning models. I believe this is one of the reasons for the fact that deep learning algorithms did not make such a great performance, comparing with machine learning models such as RF or DT, as described in the next chapter. Simply said, my decision was to sacrifice the best performace for the DL models in order to not have a very big training time.

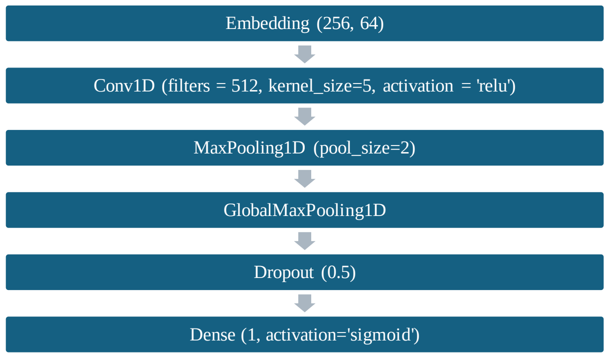
Embedding (256, 64)
Conv1D (filters = 512, kernel_size=5, activation = 'relu')
MaxPooling1D (pool_size=2)
GlobalMaxPooling1D
Dropout (0.5)
Dense (1, activation='sigmoid')

Fig. 4. Architecture of CNN model

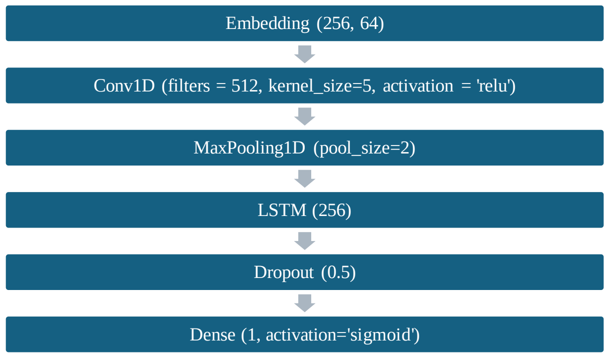
Embedding (256, 64)
Conv1D (filters = 512, kernel_size=5, activation = 'relu')
MaxPooling1D (pool_size=2)
LSTM (256)
Dropout (0.5)
Dense (1, activation='sigmoid')

Fig. 5. Architecture of CNN + LSTM model

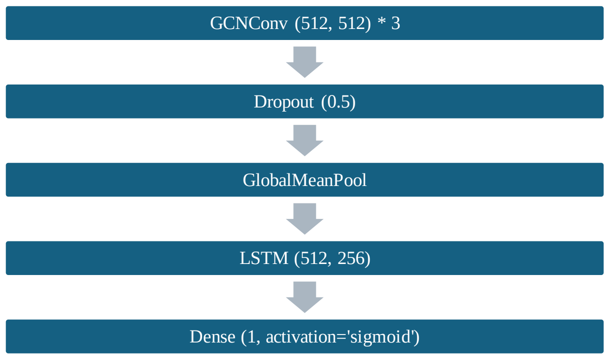

Fig. 6. Architecture of GNN + LSTM model

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. 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 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%**
Real-world datasets	**65.76%**	51.15%
Combined	**86.39%**	85.30%

TABLE IV. ACCURACY FOR GNN + LSTM MODELS

Dataset	*GNN + LSTM*
SATE IV Juliet	87.50%
Real-world datasets	55.50%
Combined	82.50%

VI. CONCLUSIONS

The results show that no single approach uniformly outperforms others across all criteria. Classical machine learning models, such as Decision Trees and Random Forests, demonstrated solid performances. 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 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. Then, GNNs operating on CFGs demonstrated promising results for representing the dynamic aspects of program execution paths.

The best performing model from this paper is the CNN+LSTM on Juliet dataset, with 98.35% accuracy. The best model for the combined dataset was the DT model that uses Count Vectorizer as input, which gave 91% accuracy. The main reason why simpler ML models gave better results in some scenarios is that DT and RF seem to receive more information from AST structures. Another reason can be the training process for the deep learning techniques. Without computation power, deep learning models cannot perform at their full potential.

In conclusion, the study presents some insights regarding vulnerability detection using AI. First, feature engineering has an important role. Structured code representations usually 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. Machine learning approaches offer significant potential for advancing automated vulnerability detection, but their adoption must be guided by careful consideration of model performance and computational cost to real-world code.

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*.

[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., ... & 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] Bilgin, Z., Ersoy, M. A., Soykan, E. U., Tomur, E., Çomak, P., & Karaçay, L. (2020). Vulnerability prediction from source code using machine learning. *IEEE Access*, *8*, 150672-150684.

[14] 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).

[15] 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.

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

[17] Fan, J., Li, Y., Wang, S., & Nguyen, T. N. (2020, June). AC/C++ code vulnerability dataset with code changes and CVE summaries. In *Proceedings of the 17th international conference on mining software repositories* (pp. 508-512).

[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] 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).

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

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

[22] 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.