Evaluate Current Machine Learning Algorithms Employed for Static Code Analysis

Ahlaad Yalavarthi
ayalavarthi@wm.edu

Abstract

Static code analysis is essential for detecting software vulnerabilities early in the development lifecycle. Traditional rule-based tools, however, often suffer from limited accuracy and high false-positive rates. In this project, we aim to explore the use of machine learning and deep learning models to classify vulnerable functions in source code, thus attempting to enhance detection performance and reduce the manual effort required.

We evaluate a diverse set of models, including classical models such as Logistic Regression, Support Vector Machines (SVM), Random Forest, as well as neural architectures such as a Multi-Layer Perceptron (MLP), and augment these models with CodeBERT, a transformer-based language model trained on programming languages. We use this to generate semantic embeddings of functions, which are then classified using downstream models.

All models are trained and tested on the open-source "DiverseVul" dataset. This contains over 300,000 labeled C/C++ functions with binary vulnerability labels, marked 0 for safe and 1 for vulnerable. We experiment with increasing sample sizes and compare TF-IDF-based approaches and CodeBERT-based pipelines.

Our experiments show that TF-IDF + SVM classifier achieved the highest performance, with an F1-score of 0.49 for the vulnerable class and 0.80 overall accuracy. These results demonstrate the value of semantic-aware representations for static code analysis and suggest that lightweight classifiers can perform effectively when coupled with pretrained language models. We conclude with recommendations for future work, including further fine-tuning of transformer models and evaluating their robustness against adversarial inputs.

1 Introduction

Detecting software vulnerabilities is crucial to prevent cybercrimes and economic losses, but to date it remains a hard problem. Static code analysis refers to the process of analyzing source code without executing it in order to detect bugs, vulnerabilities, and violations of programming standards. This plays a critical role in secure software development pipelines, especially for identifying vulnerabilities early on in the lifecycle. However, conventional rule-based tools such as static analyzers often exhibit high false positive rates, limited generalizability, and difficulty detecting complex, context-dependent vulnerabilities. Given the tremendous success of machine learning (ML) and deep learning (DL) techniques in image and natural language applications, it is natural to wonder if deep learning can enhance our ability to detect vulnerabilities [7].

These approaches aim to learn from labeled examples of vulnerable code, capturing implicit patterns that traditional tools may miss. However, many ML-based static analysis systems are sensitive to code structure, token imbalance, and data quality, and their performance varies significantly depending on the model and representation used. This project evaluates the effectiveness of various ML and DL models for static vulnerability classification. We compare classical models like Logistic Regression, Support Vector Machines (SVM), and Random Forests against neural architectures such as Multi-Layer Perceptrons (MLPs). We also integrate CodeBERT, a transformer-based pretrained model trained on code corpora [11], to generate semantic representations of functions, which are then classified by downstream models.

Our goal is to empirically assess the strengths and weaknesses of these approaches using the DiverseVul dataset, a large-scale collection of labeled C/C++ functions. We explore how different feature representations (TF-IDF vs. CodeBERT) and model architectures affect classification performance, especially on the vulnerable class. This study aims to inform the design of future ML-based static analysis tools and guide efforts toward more robust, generalizable detection methods.

The remainder of this paper proceeds as follows. Section 2 overviews our sample paper. Section 3 describes the design of our sample paper. Section 4 evaluates our solution. Section 5 discusses additional topics. Section 6 describes related work. Section 7 concludes.

2 Overview

Our approach compares and evaluates classical and deep learning-based machine learning models for static code vulnerability detection. The primary goal is to assess how well different models identify vulnerable C/C++ functions using static analysis, and how more sophisticated input representations influence that performance.

We begin by pre-processing a large dataset of labeled functions (DiverseVul in our case), where all functions are assigned as vulnerable or safe. For our initial baselines, we extract traditional TF-IDF representations from the raw code and train classifiers such as Logistic Regression, Support Vector Machines (SVM), and Random Forest. We then develop a Multi-Layer Perceptron (MLP) trained on these same TF-IDF vectors to see if neural networks are able to improve over traditional classifiers.

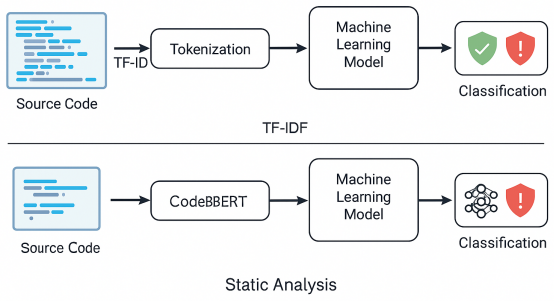

Figure 1: A high-level architecture of our approach

The second half of our pipeline utilizes CodeBERT, a pretrained transformer-based model trained on code corpora. We use CodeBERT to convert each function into a dense semantic embedding by extracting the [CLS] token representation. These embeddings are then used as input to downstream classifiers, including Logistic Regression, MLP, and optionally SVM, to evaluate how well semantic-aware embeddings help in vulnerability detection.

Throughout the pipeline, we experiment with increasing data sizes (from 5K to 50K/100k functions), use stratified train/test splits, and also use class weighting to deal with imbalance. The models are evaluated using the following metrics:

- Precision:
- Recall
- F1-score
- Accuracy

We will give special focus to the performance of the models for the vulnerable class.

Figure 1 provides a high-level architecture of our system pipeline. It captures the data pre-processing stage, the dual representation pathways (TF-IDF and CodeBERT), and the downstream classifiers applied in each case.

3 Design

Our system is designed to evaluate the effectiveness of various machine learning models in detecting software vulnerabilities from static code. The architecture consists of the following components: dataset preparation, feature extraction, model selection, and evaluation.

3.1 Dataset

We use the DiverseVul dataset, a large-scale labeled dataset consisting of 330,492 non-vulnerable and 18,945 vulnerable C/C++ functions. The dataset was built from 7,514 commits across 797 open-source projects, identified using vulnerability references from sources such as NVD, GitHub Security Advisory, and project-specific advisories.

3.1.1 Preparation and Description

Each sample is labeled according to the presence of vulnerabilities and mapped to one or more CWE (Common Weakness Enumeration) categories. Each function is annotated with a binary label: 0 for safe and 1 for vulnerable. To ensure computational feasibility and experimental reproducibility, we sample subsets of increasing sizes (5K, 10K, 20K, 50K, and 100K) and maintain class balance through stratified sampling.

Samples were de-duplicated using MD5 hashes and further cleaned by removing trivial functions (e.g., fewer than three lines of logic). Despite its size, the dataset remains imbalanced, with vulnerable samples representing roughly 5% of the total. Label noise is also present, though the authors of DiverseVul estimate that 60% of vulnerable samples are correctly labeled, which is an improvement over prior datasets like Big-Vul.

3.1.2 CWE Coverage and Hardness

The dataset includes a wide range of CWEs, including the top 25 most critical types. Table 1 below is from the original DiverseVul paper, which shows how different CWEs vary in terms of prevalence and difficulty. For example, CWE-502 (Deserialization of Untrusted Data) has a high true positive rate (TPR) despite low training coverage, while CWE-119 (Buffer Errors) is common but more error-prone.

CWE	Train (%)	Test #	TPR (%)	FPR (%)	Description
CWE-119	**15.16**	313	**39.30**	3.55	Improper Restriction of Operations within the Bounds of a Memory Buffer
CWE-120	2.29	49	**40.82**	3.55	Buffer Copy without Checking Size of Input (Classic Buffer Overflow)
CWE-125	**11.08**	239	27.20	3.55	Out-of-bounds Read
CWE-189	2.97	57	**31.58**	3.55	Numeric Errors
CWE-190	**4.77**	100	21.00	3.55	Integer Overflow or Wraparound
CWE-200	**5.10**	131	**31.30**	3.55	Exposure of Sensitive Information to an Unauthorized Actor
CWE-20	**10.76**	224	**32.59**	3.55	Improper Input Validation
CWE-22	1.13	20	25.00	3.55	Improper Limitation of a Pathname to a Restricted Directory (Path Traversal)
CWE-264	3.55	73	28.77	3.55	Permissions, Privileges, and Access Controls
CWE-269	1.14	23	8.70	3.55	Improper Privilege Management
CWE-276	0.19	3	0	3.55	Incorrect Default Permissions
CWE-284	3.35	77	25.97	3.55	Improper Access Control
CWE-287	0.58	10	10.00	3.55	Improper Authentication
CWE-306	0.00	0	N/A	3.55	Missing Authentication for Critical Function
CWE-310	1.95	44	25.00	3.55	Cryptographic Issues
CWE-352	0.10	1	0	3.55	Cross-Site Request Forgery (CSRF)
CWE-362	2.62	61	16.39	3.55	Race Condition
CWE-369	1.26	31	29.03	3.55	Divide By Zero
CWE-399	**5.29**	110	**41.82**	3.55	Resource Management Errors
CWE-400	2.38	34	5.88	3.55	Uncontrolled Resource Consumption
CWE-401	1.83	33	24.24	3.55	Missing Release of Memory after Effective Lifetime
CWE-415	1.55	30	30.00	3.55	Double Free
CWE-416	**5.46**	112	17.86	3.55	Use After Free
CWE-434	0.07	1	0	3.55	Unrestricted Upload of File with Dangerous Type
CWE-476	**5.00**	106	17.92	3.55	NULL Pointer Dereference
CWE-502	0.05	3	**66.67**	3.55	Deserialization of Untrusted Data
CWE-611	0.09	3	0	3.55	Improper Restriction of XML External Entity Reference
CWE-703	**6.39**	133	10.53	3.55	Improper Check or Handling of Exceptional Conditions
CWE-77	0.18	6	16.67	3.55	Command Injection
CWE-78	0.38	7	0	3.55	OS Command Injection
CWE-787	**15.57**	311	**33.76**	3.55	Out-of-bounds Write
CWE-79	0.47	12	**50.00**	3.55	Cross-site Scripting
CWE-798	0.01	0	N/A	3.55	Use of Hard-coded Credentials
CWE-862	0.26	6	16.67	3.55	Missing Authorization
CWE-89	0.31	9	**33.33**	3.55	SQL Injection
CWE-918	0.02	4	0	3.55	Server-Side Request Forgery (SSRF)
CWE-94	0.69	15	0	3.55	Improper Control of Generation of Code (Code Injection)

Table 1: CWE-wise prediction performance using the CodeT5 Base model (adapted from DiverseVul [7]). Top-10 highest training coverage and TPR scores highlighted in bold.

3.2 Feature Extraction

Our system supports two primary types of feature representations:

- **TF-IDF Vectorization:** Each function is tokenized into a bag-of-words representation, and a TF-IDF matrix is computed. This sparse representation is used as input to traditional machine learning classifiers and MLPs.
- **CodeBERT Embeddings:** CodeBERT, a pretrained transformer model for source code, is used to convert each function into a 768-dimensional embedding by extracting the [CLS] token from the final hidden layer. These dense semantic vectors are input into downstream classifiers.

3.3 Model Selection

We evaluate both classical and deep learning models, each trained using either TF-IDF or CodeBERT embeddings:

- Logistic Regression (with class balancing)
- Support Vector Machine (SVM) with linear kernel
- Random Forest (limited depth, standard settings)
- Multi-Layer Perceptron (2 hidden layers, ReLU activations)

Each model is trained using an 80-20 train-test split, with stratified sampling to preserve class distribution. For neural models, a validation split of 10% is used during training. When using CodeBERT embeddings, training is performed on NumPy tensors rather than sparse matrices.

3.4 Threat Model

Our design assumes the attacker's goal is to inject vulnerable code into a codebase that evades detection by static analysis tools. The adversary may craft code that appears benign or mimics safe patterns. We assume the training data (DiverseVul) is trusted and free of adversarial contamination. Our system's trusted computing base includes the preprocessing pipeline and any pretrained models such as CodeBERT.

3.4.1 Who is the adversary?

The adversary in this scenario is an attacker aiming to introduce vulnerabilities into the codebase or evade detection mechanisms used in static code analysis. This adversary could be an external attacker or a malicious insider seeking to exploit weak points within the application's code. By introducing subtle or obfuscated vulnerabilities, such as those that manipulate memory, bypass access controls, or misuse input validation, the adversary seeks to compromise the application for personal or financial gain, disrupt system functionality, or gain unauthorized access to sensitive information.

3.4.2 Adversary Goal:

The adversary's primary goal is to introduce undetected vulnerabilities or bypass detection mechanisms in static code analysis. Specific objectives may include:

1. Injecting Malicious Code: Embedding code that performs unauthorized actions, such as data exfiltration, system control, or disruption of normal operations, without being flagged by detection systems.
2. Gaining Unauthorized Access: Exploiting weaknesses to access restricted areas of the application or sensitive data, often with the intent to misuse or manipulate information.
3. System Disruption: Introducing vulnerabilities that could lead to crashes, unexpected behavior, or denial of service, either to harm the organization or benefit a competitor.
4. Evading Detection Mechanisms: Crafting vulnerabilities that remain hidden from machine learning-based detection models, potentially through obfuscation, polymorphism, or subtle manipulations that bypass standard detection patterns.
5. Financial or Competitive Gain: Achieving financial rewards through activities like data theft, system ransom, or intellectual property manipulation, as well as gaining competitive insights or creating disruptions in competitor systems.

3.4.3 Adversary Capabilities:

Adversaries are assumed to be capable of:

1. Code Manipulation and Obfuscation Skills: The adversary is skilled in manipulating code to introduce subtle vulnerabilities that may not be easily detectable.
2. Possessing Knowledge of Detection Patterns: The adversary is familiar with common vulnerability detection methods and this knowledge enables them to craft exploits designed specifically to evade known detection signatures or bypass certain analysis features.
3. Access to Source Code: Depending on the scenario, the adversary may have access to the source code

during the development process. This could include insiders or external contributors with authorized access, allowing them to directly introduce vulnerabilities into the codebase.

4. Proficiency in Exploiting Vulnerabilities: The adversary has expertise in exploiting a range of vulnerability types, such as injection attacks, buffer overflows, or input validation issues. This allows them to strategically target weak spots that the detection model may struggle with.

5. Limited Resources or Privileges: Although skilled, the adversary may lack physical access to systems or administrative control over all components. They rely primarily on software-based tactics and avoid scenarios requiring privileged system access or physical proximity.

3.4.4 Trusted Computing Base (TCB):

We assume the following components to be part of the Trusted Computing Base (TCB) for our system:

- **DiverseVul Dataset:** The dataset used for training is assumed to be free from adversarial poisoning, although it may include some natural label noise.
- **Pretrained CodeBERT Model:** The CodeBERT transformer is assumed to be downloaded from a trusted source and not backdoored or altered maliciously.
- **Training Pipeline:** The preprocessing, feature extraction, and classifier training scripts are assumed to be executed in a secure, non-compromised environment.
- **Runtime Environment:** Model inference and evaluation occur in trusted environments (e.g., local machine or secured cloud instance) with no malicious software interference.

3.5 Research Questions

This project is guided by the following research questions:

- **RQ1:** How accurately do classical and deep learning models detect vulnerabilities in source code, and how do they compare in terms of precision, recall, and F1-score?
- **RQ2:** How does the choice of feature representation — traditional TF-IDF versus pretrained semantic embeddings (CodeBERT) — affect model performance?
- **RQ3:** How does training dataset size influence the model's ability to generalize and reliably detect vulnerabilities, especially for the minority (vulnerable) class?
- **RQ4:** Are lightweight models (e.g., Logistic Regression, shallow MLPs) sufficient when combined with powerful embeddings, or are more complex architectures necessary?
- **RQ5:** What are the limitations of current models in terms of detecting harder vulnerability types, and how can evaluation insights inform improvements to reduce false positives and improve robustness?

4 Evaluation

We evaluate multiple combinations of feature representations and classifiers to determine their effectiveness in binary vulnerability detection. Each model is trained and tested on subsets of the DiverseVul dataset, with consistent 80-20 train-test splits and stratified sampling to preserve class distribution.

We use four primary metrics to evaluate performance: precision, recall, F1-score, and accuracy. In particular, we emphasize performance on the vulnerable class (labeled 1), as these instances are of greater security interest and inherently more difficult to detect.

Given the significant class imbalance in our dataset, we particularly emphasize the F1-score of the vulnerable class (label = 1), as it balances both false positives and false negatives. All experiments are run on balanced training sets of increasing size (5K, 10K, 20K, 50K, and 100K samples), with an 80-20 train-test split and stratification to preserve class ratios.

4.1 Experimental Setup

All models were implemented using Python (scikit-learn, TensorFlow, and HuggingFace Transformers). TF-IDF-based models were trained and evaluated locally on a Windows 10 machine with 16 GB RAM and an NVIDIA RTX 4070 GPU. CodeBERT experiments—due to GPU memory constraints and long embedding times—were executed on Google Colab with access to T4 and A100 GPUs. Random seeds were fixed where applicable (e.g., `random_state=42`) to ensure reproducibility.

4.2 Model Comparisons

We begin with traditional TF-IDF feature extraction, followed by classifiers including Logistic Regression, SVM,

Model	Samples	Accuracy	Precision (1)	Recall (1)	F1-score (1)
TF-IDF + SVM	100K	0.80	0.40	0.64	**0.49**
TF-IDF + LR	100K	0.79	0.38	0.64	0.48
TF-IDF + RF	20K	0.73	0.21	0.42	0.28
TF-IDF + RF	100K	0.85	0.48	0.19	0.27
TF-IDF + MLP	20K	0.74	0.30	0.57	0.39
TF-IDF + MLP	50K	0.82	0.38	0.33	0.35
CodeBERT + LR	20K	0.68	0.27	0.67	0.38
CodeBERT + MLP	50K	0.75	0.32	0.54	0.40

Table 2: Performance metrics for all models evaluated on the vulnerable class (label = 1). F1-score highlights detection balance.

Random Forest, and MLP. We also experiment with CodeBERT embeddings combined with downstream classifiers. Table 2 summarizes the results for each model tested.

4.3 Interpretation of Results

Surprisingly, the best performance was achieved by a traditional pipeline: TF-IDF + Support Vector Machine (SVM), trained on **100K samples**. TF-IDF + SVM model configuration achieved an **F1-score of 0.49** on the vulnerable class, **recall of 0.64**, and an overall accuracy of **0.80**. These results outperformed CodeBERT + MLP, which peaked at an F1-score of 0.40 and accuracy of 0.75 with 50K samples. Figure 3 helps us better visualize the results.

Figure 2 is the confusion matrix for this TF-IDF + SVM model. The model demonstrates strong discriminatory power, correctly identifying 14,152 safe functions and 1,917 vulnerable ones. However, it also misclassifies 2,848 safe functions as vulnerable (false positives) and misses 1,083 vulnerable functions (false negatives). While the vulnerable recall (64%) is higher than most models in our experiments, the trade-off in precision reflects the inherent challenge of identifying subtle or obfuscated vulnerabilities. These results further emphasize the value of scalable and interpretable models for high-precision static code analysis.

This result underscores two key observations:

- **Sample size matters:** The richer the training set, the better the model learns to generalize rare vulnerable patterns, even with sparse TF-IDF vectors.
- **Simplicity scales:** Lightweight models like SVM can outperform more complex neural pipelines when provided sufficient, balanced data.

We note that CodeBERT remains a powerful representation model, especially for semantic understanding. However, when computational resources or dataset sizes allow, TF-IDF-based models combined with SVM are highly competitive in practice.

In future work, we aim to combine the strengths of both approaches, such as integrating CodeBERT with traditional models or fine-tuning transformer layers end-to-end on vulnerability-specific labels.

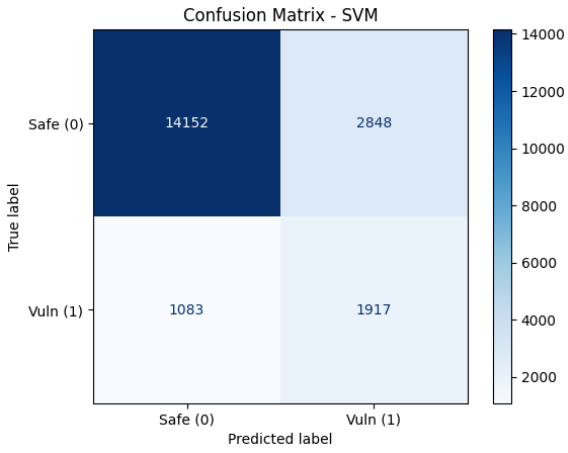

Figure 2: Confusion Matrix of TF-IDF + SVM

5 Discussion

Our evaluation puts forward significant observations and trade-offs in applying machine learning models to detect static code vulnerabilities.

First, traditional feature representations like TF-IDF, when paired with lightweight models such as SVM, can outperform deep learning pipelines under the right conditions. Notably, TF-IDF + SVM achieved the best F1-score for the vulnerable class (0.49), demonstrating that simpler

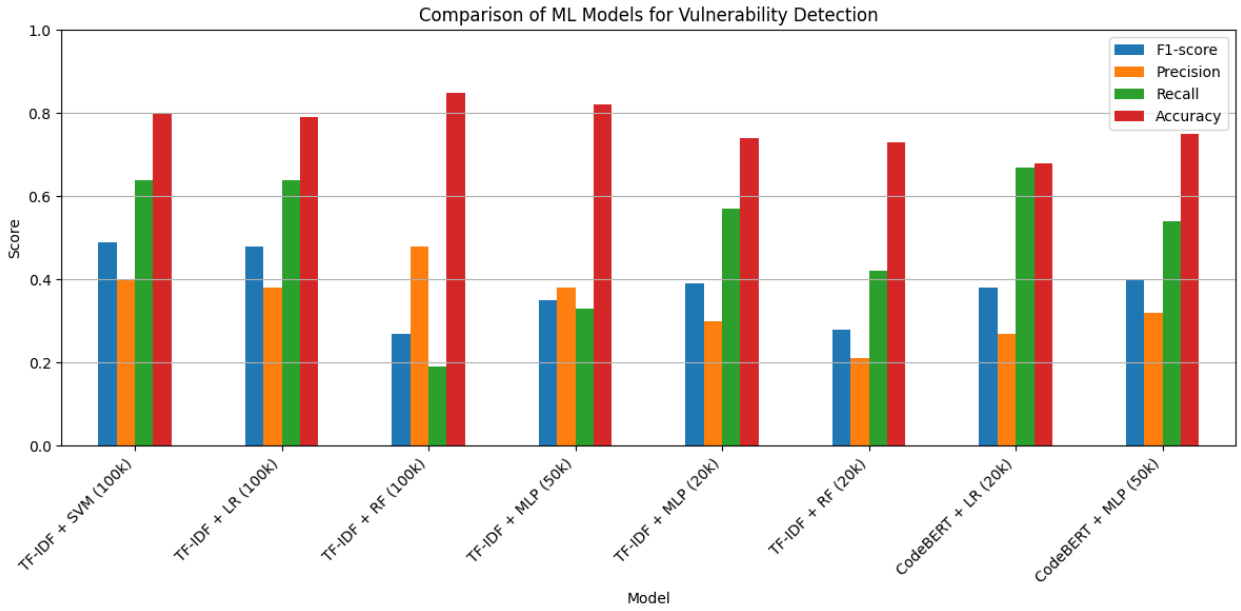

Figure 3: Comparison of ML Models for Vulnerability Detection

models can generalize effectively with sufficient training data.

Second, while CodeBERT provides semantically rich embeddings and captures deeper code semantics, its full potential depends on computational resources and careful downstream modeling. Although CodeBERT + MLP performed competitively at smaller sample sizes, it was eventually surpassed by TF-IDF + SVM as training data increased.

Third, model performance improved steadily as training dataset size grew. Such a pattern supports the importance of large-scale, diverse, and well-tagged datasets like DiverseVul, especially in low-resource settings where vulnerability samples are scarce.

Despite these gains, class imbalance remains a persistent challenge. Even with class weighting and stratified sampling, precision for the vulnerable class lagged behind. Future work could explore more robust remedies such as SMOTE, data augmentation, or adversarial vulnerability generation.

Overall, our findings suggest that simplicity, scalability, and thoughtful data preprocessing can match or even exceed the performance of transformer-based methods—especially when computational efficiency is a priority.

6 Related Work

Static code analysis is a fundamental tool in software engineering, widely used to detect potential vulnerabilities and improve code quality without executing the program. Nachtigall [21] provided a perspective on explaining static analysis, highlighting its potential and challenges. Static code analysis tools like VulDeePecker have been developed to leverage deep learning for vulnerability detection [18]. Alikhashashneh explored the usefulness of machine learning techniques in enhancing the effectiveness of static code analysis tools, improving their applicability in real-world environments [1].

The application of machine learning to analyze large codebases, often referred to as ”Big Code,” has seen significant research interest. Allamanis [2] provided a survey on machine learning for Big Code, emphasizing the naturalness of code and how machine learning models can be adapted for code understanding and analysis. Their earlier work focused on representing programs with graphs, improving the representation of code in machine learning models [3]. The code2vec model, proposed by Alon [4], introduced a method for learning distributed representations of code, further advancing this area of research.

In the field of machine learning for program analysis, the Ariadne project aimed to develop techniques for analyzing machine learning programs themselves [9]. The detection of vulnerabilities has also benefited from machine learning approaches, such as automated vulnerability detection systems proposed by Harer [13] and the au-

tomatic feature learning methods for vulnerability prediction by Dam [8]. Li [17] introduced VulPecker, an automated system for detecting vulnerabilities using code similarity analysis.

Another interesting development in the field is the work by Rogatch, who introduced hierarchical clustering for software architecture inference from source code [25]. Additionally, the generation of comments for source code, as discussed by Wang and Zhang [34], offers insights into improving the understandability and documentation of software systems.

Li [16] explored the applications of deep learning in software engineering, showcasing how machine learning models can enhance code analysis and vulnerability detection. The analysis of static code analysis tools continues to evolve, with Nikolić [22] conducting an evaluation of these tools for their effectiveness in security applications. A survey by Sharma [26] provided a comprehensive overview of machine learning techniques used in source code analysis, underlining their growing importance.

Wang [33] further explored the static analysis of source code vulnerabilities using machine learning techniques, providing a detailed survey on the subject. Kaur and Nayyar [14] conducted a comparative study of static code analysis tools for C/C++ and Java, highlighting their effectiveness in detecting vulnerabilities. Kulenovic and Donko [15] surveyed static code analysis methods for security vulnerability detection, contributing to a deeper understanding of this field.

Machine learning-based static code analysis for software quality assurance has also gained attention, with Sultanow [27] investigating how these techniques can improve the quality and security of software systems. Tribus [29] explored static code features for a machine learning-based inspection, focusing on C code.

The detection of code smells has also been explored using machine learning techniques. Arcelli Fontana [5] compared various machine learning methods for code smell detection, showing how these models can improve the maintainability of software. Fan [10] investigated the role of static code analysis in the AI era, discussing how intelligent code analysis agents can enhance software security.

Deep learning has been applied to the processing of static analysis alarms, with Tan and Tian [28] proposing a method for refining these results to reduce false positives. Lujan [19] conducted a preliminary study on the adequacy of static analysis warnings in predicting code smells, contributing to the ongoing debate on the effectiveness of these tools.

In the domain of data science software, Urban [32] explored the static analysis of such software, addressing the unique challenges faced in this field. Tribus [30] investigated the use of data mining for static code analysis in C, further showcasing the potential of these techniques.

Automation in code review activities is another area where machine learning has been applied. Tufano [31] proposed a machine learning-based approach to automate code reviews, improving the efficiency of the software development lifecycle. Moshkin [20] contributed to this field by exploring the automation of program code analysis using machine learning methods.

Zhou [35] introduced Devign, a system that uses graph neural networks for effective vulnerability identification, highlighting the growing importance of neural networks in this domain. Ghaffarian and Shahriari [12] conducted a survey on machine learning and data mining techniques for vulnerability discovery, offering a comprehensive overview of the current landscape.

Finally, the use of machine learning-based fuzzing techniques for vulnerability detection has been systematically reviewed by Chafjiri [6], contributing to a deeper understanding of the effectiveness of these methods. Online resources, such as articles from PVS-Studio [24] and the Philips Technology Blog [23], further highlight the practical applications of machine learning in static analysis, providing real-world insights into the challenges and opportunities in this field.

In conclusion, the integration of machine learning into static code analysis represents a significant advancement in software vulnerability detection. The studies discussed here demonstrate the potential for improved accuracy and efficiency in identifying security vulnerabilities, paving the way for more secure and reliable software systems.

7 Conclusion

This project demonstrates that effective vulnerability detection in static code analysis does not always require the most complex models. In practice, traditional approaches like TF-IDF combined with scalable classifiers such as SVM can outperform deep neural pipelines, provided there is sufficient data and meticulous class balancing. This goes against the current trend of defaulting to deep models and helps highlight the lasting value of simpler approaches when applied at scale.

Also, while CodeBERT provides semantic advantages, its advantages are best utilized when coupled with the appropriate classifiers and supported by sufficient computation resources. The trade-off between computational cost and marginal gain is worth examining further, especially in computationally constrained environments.

This work also underscores the critical role of high-quality, diverse datasets. As vulnerability detection is inherently imbalanced, future efforts must not only focus on better models but also on better data—through improved labeling, synthetic sample generation, and active learning.

Ultimately, the path forward may lie in hybrid strategies that fuse semantic embeddings with symbolic features or combine static and dynamic analysis. As adversarial patterns grow more subtle and codebases scale, the synergy of simplicity, scale, and semantic insight will be essential for advancing secure software development.

Many of the models referred to in the related work section are complex models which have had varying degrees of success. As Artificial Intelligence (AI) and ML keeps growing at a rapid pace, more and more techniques become available to experiment with and we could possibly combine them to hopefully achieve an ideal model that detects most vulnerabilities, with as few false positives as possible

References

[1] E. A. Alikhashashneh. USING MACHINE LEARNING TECHNIQUES TO IMPROVE STATIC CODE ANALYSIS TOOLS USEFULNESS. 10 2019.

[2] M. Allamanis, E. T. Barr, P. Devanbu, and C. Sutton. A survey of machine learning for big code and naturalness. *ACM Comput. Surv.*, 51(4), July 2018.

[3] M. Allamanis, M. Brockschmidt, and M. Khademi. Learning to represent programs with graphs. *arXiv preprint arXiv:1711.00740*, 2017.

[4] U. Alon, M. Zilberstein, O. Levy, and E. Yahav. code2vec: learning distributed representations of code. *Proc. ACM Program. Lang.*, 3(POPL), Jan. 2019.

[5] F. Arcelli Fontana, M. V. Mäntylä, M. Zanoni, and A. Marino. Comparing and experimenting machine learning techniques for code smell detection. *Empirical Software Engineering*, 21:1143–1191, 2016.

[6] S. B. Chafjiri, P. Legg, J. Hong, and M.-A. Tsompanas. Vulnerability detection through machine learning-based fuzzing: A systematic review. *Computers & Security*, page 103903, 2024.

[7] Y. Chen, Z. Ding, L. Alowain, X. Chen, and D. Wagner. Diversevul: A new vulnerable source code dataset for deep learning based vulnerability detection, 2023.

[8] H. K. Dam, T. Tran, T. Pham, S. W. Ng, J. Grundy, and A. Ghose. Automatic feature learning for vulnerability prediction. *arXiv preprint arXiv:1708.02368*, 2017.

[9] J. Dolby, A. Shinnar, A. Allain, and J. Reinen. Ariadne: analysis for machine learning programs. In *Proceedings of the 2nd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*, MAPL 2018, page 1–10, New York, NY, USA, 2018. Association for Computing Machinery.

[10] G. Fan, X. Xie, X. Zheng, Y. Liang, and P. Di. Static code analysis in the ai era: An in-depth exploration of the concept, function, and potential of intelligent code analysis agents. *arXiv preprint arXiv:2310.08837*, 2023.

[11] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang, and M. Zhou. Codebert: A pre-trained model for programming and natural languages, 2020.

[12] S. M. Ghaffarian and H. R. Shahriari. Software vulnerability analysis and discovery using machine-learning and data-mining techniques: A survey. *ACM computing surveys (CSUR)*, 50(4):1–36, 2017.

[13] J. A. Harer, L. Y. Kim, R. L. Russell, O. Ozdemir, L. R. Kosta, A. Rangamani, L. H. Hamilton, G. I. Centeno, J. R. Key, P. M. Ellingwood, et al. Automated software vulnerability detection with machine learning. *arXiv preprint arXiv:1803.04497*, 2018.

[14] A. Kaur and R. Nayyar. A comparative study of static code analysis tools for vulnerability detection in c/c++ and java source code. *Procedia Computer Science*, 171:2023–2029, 2020. Third International Conference on Computing and Network Communications (CoCoNet'19).

[15] M. Kulenovic and D. Donko. A survey of static code analysis methods for security vulnerabilities detection. In *2014 37th International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*, pages 1381–1386, 2014.

[16] X. Li, H. Jiang, Z. Ren, G. Li, and J. Zhang. Deep learning in software engineering. *arXiv preprint arXiv:1805.04825*, 2018.

[17] Z. Li, D. Zou, S. Xu, H. Jin, H. Qi, and J. Hu. Vulpecker: an automated vulnerability detection system based on code similarity analysis. In *Proceedings of the 32nd Annual Conference on Computer Security Applications*, ACSAC '16, page 201–213, New York, NY, USA, 2016. Association for Computing Machinery.

[18] Z. Li, D. Zou, S. Xu, X. Ou, H. Jin, S. Wang, Z. Deng, and Y. Zhong. Vuldeepecker: A deep learning-based system for vulnerability detection. In

Proceedings of the 2018 Network and Distributed System Security Symposium (NDSS 2018). Internet Society, 2018.

[19] S. Lujan, F. Pecorelli, F. Palomba, A. De Lucia, and V. Lenarduzzi. A preliminary study on the adequacy of static analysis warnings with respect to code smell prediction. In *Proceedings of the 4th ACM SIGSOFT International Workshop on Machine-Learning Techniques for Software-Quality Evaluation*, MaLTeSQuE 2020, page 1–6, New York, NY, USA, 2020. Association for Computing Machinery.

[20] V. Moshkin, V. Kalachev, and A. Zarubin. Automation of program code analysis using machine learning methods. In *2022 International Russian Automation Conference (RusAutoCon)*, pages 404–408. IEEE, 2022.

[21] M. Nachtigall, L. Nguyen Quang Do, and E. Bodden. Explaining static analysis - a perspective. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering Workshop (ASEW)*, pages 29–32, 2019.

[22] D. Nikolić, D. Stefanović, D. Dakić, S. Sladojević, and S. Ristić. Analysis of the tools for static code analysis. In *2021 20th International Symposium INFOTEH-JAHORINA (INFOTEH)*, pages 1–6, 2021.

[23] Philips Technology Blog. Using machine learning powered static analysis to identify logging, privacy, and security issues, 2022. Accessed: 2024-10-15.

[24] PVS-Studio. Machine learning in static analysis of program source code, 2022. Accessed: 2024-10-15.

[25] S. Rogatch. An efficient high-quality hierarchical clustering algorithm for automatic inference of software architecture from the source code of a software system, 2012.

[26] T. Sharma, M. Kechagia, S. Georgiou, R. Tiwari, I. Vats, H. Moazen, and F. Sarro. A survey on machine learning techniques for source code analysis. *arXiv preprint arXiv:2110.09610*, 2021.

[27] E. Sultanow, A. Ullrich, S. Konopik, and G. Vladova. Machine learning based static code analysis for software quality assurance. 09 2018.

[28] Y. Tan and J. Tian. A method for processing static analysis alarms based on deep learning. *Applied Sciences*, 14(13), 2024.

[29] H. Tribus. Static code features for a machine learning based inspection: An approach for c, 2010.

[30] H. Tribus, I. Morrigl, and S. Axelsson. Using data mining for static code analysis of c. In *Advanced Data Mining and Applications: 8th International Conference, ADMA 2012, Nanjing, China, December 15-18, 2012. Proceedings 8*, pages 603–614. Springer, 2012.

[31] R. Tufano, L. Pascarella, M. Tufano, D. Poshyvanyk, and G. Bavota. Towards automating code review activities. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pages 163–174. IEEE, 2021.

[32] C. Urban. Static analysis of data science software. In *Proceedings of the ACM on Programming Languages*, 2022.

[33] J. Wang, M. Huang, Y. Nie, and J. Li. Static analysis of source code vulnerability using machine learning techniques: A survey. In *2021 4th International Conference on Artificial Intelligence and Big Data (ICAIBD)*, pages 76–86, 2021.

[34] X. Wang and B. Zhang. Comment generation for source code: State of the art, challenges and opportunities. *arXiv preprint arXiv:1802.02971*, 2018.

[35] Y. Zhou, S. Liu, J. Siow, X. Du, and Y. Liu. Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks. *Advances in neural information processing systems*, 32, 2019.