Enhancing security assurance in software development: AI-based vulnerable code detection with static analysis.

RAJAPAKSHA, S., SENANAYAKE, J., KALUTARAGE, H. and AL-KADRI, M.O.

2024

Enhancing Security Assurance in Software Development: AI-Based Vulnerable Code Detection with Static Analysis

Sampath Rajapaksha$^{1[0000-0001-7772-3774]}$, Janaka Senanayake$^{1[0000-0003-2278-8671]}$, Harsha Kalutarage$^{1[0000-0001-6430-9558]}$, and Mhd Omar Al-Kadri$^{2[0000-0002-1146-1860]}$

[1] School of Computing, Robert Gordon University, Aberdeen AB10 7QB, UK
{s.rajapaksha,j.senanayake,h.kalutarage,}@rgu.ac.uk
[2] University of Doha for Science and Technology, Doha, Qatar
omar.alkadri@udst.edu.qa

Abstract. The presence of vulnerable source code in software applications is causing significant reliability and security issues, which can be mitigated by integrating and assuring software security principles during the early stages of the development lifecycle. One promising approach to identifying vulnerabilities in source code is the use of Artificial Intelligence (AI). This research proposes an AI-based method for detecting source code vulnerabilities and leverages Explainable AI to help developers identify and understand vulnerable source code tokens. To train the model, a web crawler was used to collect a real-world dataset of 600,000 source code samples, which were annotated using static analysers. Several ML classifiers were tested on a feature vector generated using Natural Language Processing techniques. The Random Forest and Extreme Gradient Boosting classifiers were found to perform well in binary and multi-class approaches, respectively. The proposed model achieved a 0.96 F1-Score in binary classification and a 0.85 F1-Score in multi-class classification based on Common Weakness Enumeration (CWE) IDs. The model, trained on a dataset of actual source codes, is highly generalisable and has been integrated into a live web portal to validate its performance on real-world code vulnerabilities.

Keywords: source code vulnerability · artificial intelligence · software security · vulnerability scanners

1 Introduction

In software development, it is common for developers to overlook the thorough validation of source code for security and vulnerability during coding and prior to releasing the product to the customer [6]. This oversight can cause security threats to rapidly evolve, which forces developers to keep up to date with the latest security vulnerabilities to minimize the risk of software attacks. Education on security for developers is an ongoing process, but currently, many software developers neglect security issues throughout the software development lifecycle. This may be due to a lack of understanding about how common errors in software development can lead to exploitable vulnerabilities in software systems [13], as well as the pressure for fast deployment. The communication gap between developers and cybersecurity experts has also contributed to widespread software vulnerabilities [20].

However, with the availability of data, new algorithms, and advances in computational power, AI and ML techniques can be effectively used to address various problems in different domains, including computer security and privacy [21]. Hence, AI/ML algorithms can be utilized to detect vulnerabilities in source codes [2, 14]. By using AI/ML algorithms for vulnerability detection, the need for human expertise can be reduced [22], and the process can be automated. Programming languages consist of words, numbers, and various symbols, similar to natural languages. Therefore, previous research has used Natural Language Processing (NLP) techniques to detect vulnerabilities in source code by treating the code as a form of text [3]. To train AI/ML algorithms, extracted features have been generated through NLP techniques, considering this as a classification problem. However, existing methods have used limited information for feature generation, resulting in high false negatives.

This study addresses the need for a highly accurate source code vulnerability detection method using AI/ML techniques and proposes a new approach to feature generation, and demonstrates its effectiveness through experiments. The study makes the following contributions to the field of source code vulnerability detection and offers a promising solution for automated security testing in software development.

- *Data pre-processing approach to identify important features:* Presenting a novel method using Concrete Syntax Trees (CST) to identify the most important features of source codes to train an ML model.
- *Generalized vulnerability detection models:* The generalization capability of the proposed method is high since the models are trained on a carefully generated dataset that includes real-world source codes and a subset of a synthetic dataset.
- *Explainability of the model:* Visually representing the identified vulnerable source code segments to help make the necessary changes to convert the code from vulnerable to benign. Furthermore, this supports optimising the pre-processing data approach to improve the model accuracy.
- *Integration with a web portal:* Once the developer enters the source code, the vulnerability of the code is displayed on a web portal with an explanation of the vulnerabilities associated with it.

The rest of the paper is organised as follows: Section 2 contains background and related work. Section 3 explains the methodology of this work. Section 4 discusses the performance evaluation. Finally, the conclusions and future work are discussed in Section 5.

2 Background and Related Work

By reviewing relevant studies, this section lays the groundwork for the research by providing a thorough understanding of source code vulnerabilities, different parsers and scanners, as well as a range of vulnerability detection techniques.

2.1 Vulnerabilities in Source Code

Human error can lead to numerous vulnerabilities in software code, particularly when an extensive testing and validation process is not implemented from the beginning of the software development lifecycle [6]. To promote secure software development practices, reducing vulnerabilities in the source code is essential [19]. Nevertheless, without proper mechanisms in place, some developers may overlook potential vulnerabilities. To address these issues, various repositories of vulnerabilities and weaknesses have been established by organizations and the community, such as Common Weakness Enumeration (CWE)[1] and Common Vulnerabilities and Exposures (CVE)[2]. These repositories contain software and hardware-related vulnerabilities, which developers can reference when identifying patterns and mitigating security loopholes in the source code. Additionally, it is important to note that some vulnerabilities are related to other vulnerabilities and can belong to more than one CWE category. Awareness of these relationships can help developers develop software more securely. By providing automated tool support based on CWE and CVE details, the software development process can be completed more efficiently and vulnerable source code can be minimized.

2.2 Scanners and Parsers

Supportive tools are required by software developers to integrate with their coding and detect vulnerabilities at an early stage to mitigate them through source code analysis [12, 14, 16–18]. The source code must first be formatted into a generalized form, either using CST or Abstract Syntax Tree (AST). Syntax trees can be generated through static analysis [8]. The accuracy of formulating the CST/AST and its generalisation mechanism influences the rate of false alarms on vulnerabilities. Tree-sitter[3], an open-source parser generator tool, can create a CST for a source file and efficiently update the tree when changes occur in the source code.

After parsing the code, analysis can be performed using scanners. Some scanners are available for analyzing C/C++ source code with relatively good accuracy [10]. Cppcheck[4] is an open-source static analysis tool that detects bugs, undefined behavior, and dangerous coding constructs in C/C++ code. It provides essential data for each alert, such as filename, line, severity, alert identifier,

[1] https://cwe.mitre.org
[2] https://www.cvedetails.com
[3] https://tree-sitter.github.io/tree-sitter
[4] https://cppcheck.sourceforge.io

and CWE, and can be integrated with other development tools. Another open-source tool, Flawfinder[5], can examine C/C++ source code and report possible security weaknesses. It has a built-in database of C/C++ functions with well-known vulnerable problems, such as format string problems (printf, snprintf, and syslog), buffer overflow risks (strcpy, strcat, gets, sprintf, and scanf), potential shell meta-character dangers (exec, system, popen), poor random number acquisition (random), and race conditions (access, chown, chgrp, chmod, tmpfile, tmpnam, tempnam, and mktemp).

2.3 Detecting Vulnerabilities

Previous works have proposed two techniques to detect vulnerabilities, namely metric-based techniques and pattern-based techniques. Metric-based techniques utilize features such as complexity metrics, token frequency metrics, code churn metrics, dependency metrics, developer activity metrics, or execution complexity metric to detect vulnerabilities through supervised or unsupervised machine learning methods [4]. On the other hand, pattern-based techniques utilize static analysis to identify vulnerable codes based on known vulnerable patterns. However, this technique is limited to function-level codes and is considered a preliminary step for vulnerability assessment since it does not identify the vulnerability type or possible locations. Moreover, the use of metric-based features in different machine learning algorithms showed low detection capability.

In [15], the authors utilized text features extracted from the source code to predict software defects. They considered everything as text except comments, separated by space or tab. Naive Bayes (NB) and Logistic Regression (LR) were used as classification algorithms. This approach was adapted by [15] for software vulnerability prediction tasks, using the same algorithms with Bag of Words (BoW) as features. However, the experimental results showed a lower F1-Score for all selected test cases, which may be due to poor feature selection and lack of emphasis on proper data pre-processing. In [7], n-gram (1-gram, 2-gram, and 3-gram) and word2vec were used as features to predict the presence of vulnerabilities in test cases. The authors addressed the class imbalance problem by using random oversampling. However, this model [7] is limited to binary classification and cannot identify the type of vulnerability.

In their work [14], the authors utilized minimum intermediate representation learning for detecting vulnerabilities in source code. In order to address the lack of vulnerability samples, unsupervised learning was employed during the pre-training stage. High-level features were generated using Convolutional Neural Networks (CNN), and these features were subsequently fed into classifiers such as Logistic Regression (LR), Naive Bayes (NB), Support Vector Machine (SVM), Multi-Layer Perceptron (MLP), Gradient Boosting (GB), Decision Tree (DT), and Random Forest (RF) for vulnerability detection. However, this model was trained using only two CWE-IDs from a synthetic dataset, which limits its generalization capability to other CWE-IDs and real datasets.

[5] https://github.com/david-a-wheeler/flawfinder

The authors of [5] presented a technique to assist manual source code analysis through vulnerability extrapolation. They achieved this by generating an AST using a parser, but this approach was limited to identifying vulnerabilities in only a few source code functions. Similarly, [2] employed an AST representation of source code to detect vulnerabilities. They utilized the Pycparser[6] library to generate the AST for the C language and modeled it as a binary classification task, using MLP and CNN algorithms. The proposed model targeted four CWE classes, achieving an F1-Score between 0.09 to 0.59. Commercially accessible solutions like Fortify [7] and Coverity[8] are among the tools available for identifying vulnerabilities. However, these tools are not freely accessible and have limitations in effectively identifying vulnerabilities linked to CWE IDs.

Despite the increasing popularity of utilizing machine learning for vulnerability detection, as previously mentioned, several studies have failed to achieve a high accuracy/F1-Score when identifying vulnerabilities in source code. Many of these studies were not trained on a comprehensive dataset that includes real-world data or did not follow improved pre-processing techniques. Additionally, these studies were limited to binary classification or a small number of Common Weakness Enumeration (CWE) classes. As a result, our research addresses these issues by using a real-world dataset and achieving an F1-Score of 0.96 in the binary classification model and 0.85 in the multi-class classification model for twenty CWE classes. These results outperform the state-of-the-art benchmark models.

3 Vulnerability Detection Process

In this section, we will cover the proposed vulnerability detection process, which comprises the model's architecture as well as the dataset used.

3.1 Dataset

Generally, ML-based methods require a large amount of dataset during the training phase. One of the major challenges of source code vulnerability detection is the lack of vulnerability datasets for the majority of common CWE-IDs [9]. National Institute of Standards and Technology (NIST) published a dataset to encourage the improvement of static code analysers and to find security-related defects in source code. This dataset was published as a result of Software Assurance Metrics and Tool Evaluation (SAMATE) project[9] and the dataset is referred as Static Analysis Tool Exposition (SATE IV) Juliet test suite[10]. This

[6] https://github.com/eliben/pycparser

[7] https://www.microfocus.com/en-us/cyberres/application-security/fortify-languages

[8] https://www.synopsys.com/software-integrity/security-testing/static-analysis-sast/coverity-cwe.html

[9] https://samate.nist.gov/SARD/

[10] https://www.nist.gov/itl/ssd/software-quality-group/static-analysis-tool-exposition-sate-iv

dataset includes 52,185 synthetic test cases for C and C++ languages and annotated for CWE-IDs. However, the major limitations of this dataset are highly imbalanced CWE-ID distribution and the lack of availability of benign data to train supervised ML algorithms. To address these issues, a web crawler was developed to retrieve more C and C++ source codes from public GitHub repositories. Since the retrieved GitHub source codes are not annotated for CWE-IDs to train ML algorithms, two static code analysers, Cppcheck and Flawfinder which are based on pre-defined rules were used for the CWE-ID annotation. The annotation was also verified with the expert knowledge of ethical hackers and security testers to ensure the quality of the dataset. In general, rule-based methods have lower false positives and suffer from higher false negatives. Different rule-based static code analysers have different vulnerability detection capabilities based on the defined rules. The objective of this approach, as a proof of concept, is to learn both analysers' capabilities and achieve lower false negative and positive rates from ML-based models. Going forward, a significant number of freely available and commercial-grade analysers could be utilized for annotation, enabling trained ML models to learn the capabilities of various analysers. Ultimately, this could result in the development of a superior AI-based code analyser that outperforms any existing static code analysers. Accordingly, a sample was considered as vulnerable if one of the analysers identifies the code as vulnerable. In contrast, sample was considered as benign if both analysers identify it as benign. Source codes relevant to the twenty highest CWE-IDs were considered as the vulnerable samples whereas, an approximately a similar number of benign samples were selected to make a balanced dataset. This resulted in having 600,000 C and C++ source codes, including SATA IV vulnerable codes. Figure 1 depicts the CWE-ID count distribution for the selected vulnerable source codes.

3.2 Model Architecture

The suggested framework comprises of two ML models designed for both binary and multiclass classification purposes. The binary model's function is to distinguish between vulnerable and benign statuses, whereas the multiclass model is aimed at recognizing CWE-IDs if the source code is found to be vulnerable. Additionally, the utilisation of Explainable Artificial Intelligence (XAI) [1], is incorporated to explain the predictions made by the model. This is achieved by highlighting sections of code (tokens) that are considered vulnerable, based on the outcomes from the multiclass model. The entire procedure is illustrated in Figure 2, encompassing three core steps: data preprocessing, binary and multiclass classification, and prediction explanation facilitated by XAI.

Data Pre-Processing The entire source code available in dataset is referred as the sample. This might include a function, a snippet of code or a large source code file. SATE IV dataset includes large source code files, whereas retrieved GitHub source codes include all levels of codes. Therefore, the proposed solution can take any code as input to the ML models. The developer sends the source

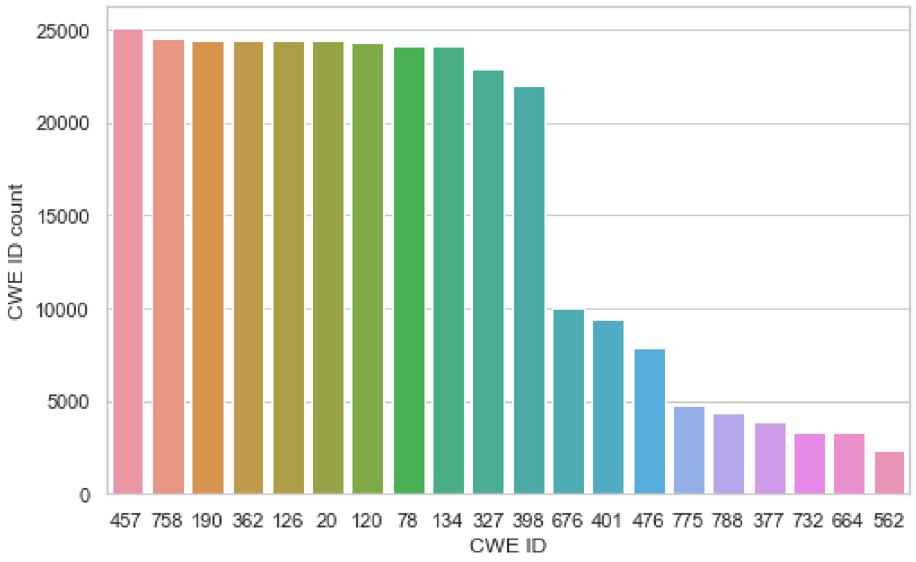
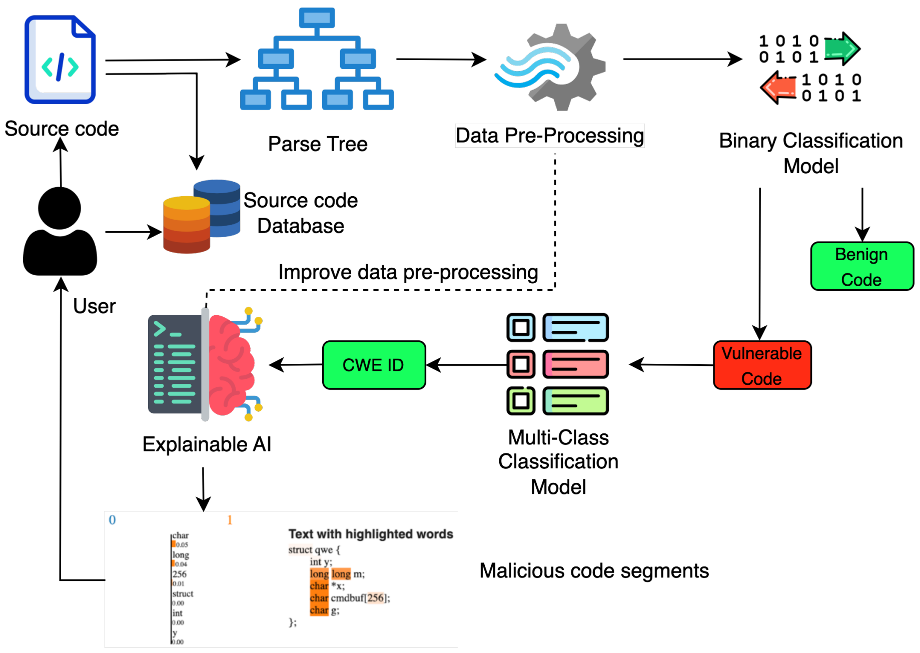


Fig. 1. CWE-ID distribution



```
struct qwe {
        int y;
        long long m;
        char *x;
        char cmdbuf[256];
        char g;
};
```

Fig. 2. Model Architecture

code into the web portal for the prediction. Since these source codes are similar to unstructured texts, first it needs to pre-process the codes to identify the features to train the ML models. To this end, CST is used to identify the tokens of the source codes. This is due to CST can retain more details of the codes than AST.

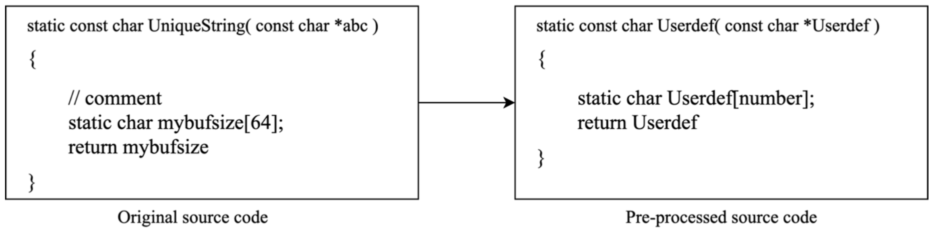
Tree-sitter is used to generate the CST. Following pre-processing steps are applied to the given source code.

1. Use Tree-sitter to generate CSTs of source codes.

2. Clean CST outputs to generate important tokens.

3. Create numerical vectors for the identified tokens to train ML models.

Generated CSTs include various information such as user comments, user-defined functions, different symbols, and hexadecimal numbers which cannot be considered as generalized features for ML models. Therefore, it needs to remove some of these details such as user comments and translate others into generalized formats. However, we identified and translated these information with the support of domain experts who are working as experienced ethical hackers and security testers to avoid the removal of important information. Figure 3 shows a sample of pre-processed source code along with the original code. According to this, we removed the comment and converted the user-defined names into the standard format of ‘Userdef’. Additionally, the number 20 was converted into ‘number’. C and C++ specific names (functions and keywords) kept unchanged. Tokens obtained from the pre-processed source code are [‘static’, ‘const’, ‘char’, ‘Userdef’, ‘number’, ‘return’, ‘(’, ‘)’, ‘*’, ‘{’, ‘}’, ‘[’, ‘]’, ‘;’]. Accordingly, we pre-processed all source codes and converted them into generalized source codes to extract the tokens.



```
static const char UniqueString( const char *abc )
{
    // comment
    static char mybufsize[64];
    return mybufsize
}
```

```
static const char Userdef( const char *Userdef )
{
    static char Userdef[number];
    return Userdef
}
```

Fig. 3. Preprocessing

Pre-processed source codes are used to generate features to train ML models. To this end, we used CountVectorizer and TfidfVectorizer Python libraries treating source codes as texts in natural language [11]. As features, Bag-of-words (BoW), n-gram (n=2,3) and term frequency-inverse document frequency (TF-IDF) were used. Grid search was used to identify the optimum hyperparameters such as maximum (max df) and minimum (min df) document frequencies.

Algorithms: Features generated by the data pre-processing step can be used in binary and multi-class classification models. Accordingly, 600,000 source code samples are used to train the binary classification model. Dataset were split into 80:20 ratio as training and testing samples. Based on our previous work, RF, LR and XGB algorithms were used as the classification algorithms with the features of BoW, n-gram and TF-IDF. The trained binary classifier can identify the given source code as vulnerable or benign. If the code is vulnerable, then it sends into the multi-class classification model to identify the relevant CWE-IDs. Similar to binary model training, 300,000 vulnerable source code samples were split into 80:20 ratio with the stratified sampling to train the above algorithms. Since a vulnerable code might have more than one vulnerability, the top K (K=3) predictions were used as possible vulnerable classes to address the multi-label cases. Python sklearn library was used to implement these algorithms. All experiments run on a MacBook M1 Pro with 16 GB RAM.

We also compare the proposed multi-class classification model with the Multi-Layer Perceptron (MLP) based model proposed in [2]. We consider this as the baseline model. This model converts the source code into the AST representation using Pycparser[11] library. To encode nodes of an AST into numeric values, they identified 48 different essential token types based on the grammar of C language and assigned unique values for each token type. Array representation of this was used to train the MLP model.

Vulnerability explanation: Identifying the code as vulnerable is not much useful if the vulnerable code segments (tokens) are not located. The developer has to go through the complete code and needs to identify these tokens manually and the lack of knowledge about the vulnerabilities might restrict the vulnerable token identification. Therefore, locating the vulnerable tokens is vital in evaluating the model prediction and to make the necessary changes to the vulnerable code to make it a benign code. To this end, we use Local Interpretable Model-agnostic Explanations (LIME)[12] framework to explain the prediction. LIME provides an explanation which is a local linear approximation of the behaviour of the trained model. LIME learns a sparse linear model by sampling instances around specific instances, approximating the trained model locally. LIME supports natural language-based models and provides visual and textual artefacts that developers can understand. We used our trained multi-class model with LIME to provide the model explanation. This provides output by highlighting the vulnerable token in the input source code. The developer confirmed accurate CWE-IDs (ground truth) sends to the source code database for future model retraining. During the model training, outputs of the LIME were used to optimize the data pre-processing with the support of domain experts by removing non-related tokens and keeping the important tokens.

[11] https://github.com/eliben/pycparser
[12] https://github.com/marcotcr/lime

4 Performance Evaluation

This section provides the results for both binary and multiclass models for a different set of features. The test dataset was used to evaluate the performance. F1-Score was selected as the evaluation metric as it provides the harmonic mean of precision and recall. For the binary classification model, label 0 represents benign source codes, whereas label 1 represents vulnerable source codes. In contrast, the multi-class classification model has twenty CWE-IDs as the label.

4.1 Machine Learning Models

Four ML models were used to identify the best models for binary and multi-class classification using selected features. Table 1 summarises the F1-Score for binary classification models for BoW, 2-gram, 3-gram and TF-IDF features. We included the default hyperparameters into the grid search criteria and all algorithms achieved the best performance for the default hyperparameters. The BoW feature achieved a higher F1-Score than the 2-gram or 3-gram. XGB model achieved the lowest detection for all features. The RF algorithm outperformed LR and XGB and showed the best performance with the feature BoW as highlighted in the green colour cell in Table 1.

Table 1. Performance of binary classification ML algorithms with BoW, n-gram, and TF-IDF features (F1-Score)

Class	**NB**				**LR**				**RF**				**XGB**			
	BoW	**2-gram**	**3-gram**	**TF-IDF**	**BoW**	**2-gram**	**3-gram**	**TF-IDF**	**BoW**	**2-gram**	**3-gram**	**TF-IDF**	**BoW**	**2-gram**	**3-gram**	**TF-IDF**
0	0.72	0.57	0.63	0.84	0.90	0.88	0.89	0.91	0.95	0.95	0.95	0.95	0	0.02	0.03	0
1	0.81	0.76	0.78	0.85	0.89	0.88	0.89	0.91	0.96	0.95	0.95	0.95	0.68	0.63	0.66	0.68
Overall	0.76	0.66	0.71	0.84	0.89	0.88	0.89	0.91	0.96	0.95	0.95	0.95	0.34	0.33	0.37	0.34

Table 2 presents the performance for multi-class classification. Increasing the n-gram of the LR model resulted to achieve higher F1-Score. However, the opposite can be observable for the RF and XGB models. XGB model with BoW feature outperformed all other algorithms and feature combination with the overall F1-Score of 0.85 as highlighted in the green colour cell in Table 2. Similar to the binary classification features, BoW performed better than n-gram for multi-class classification. In general, higher n-gram includes the context of tokens and are expected to perform better than the BoW feature. However, increasing the n-gram causes to increase the sparsity of feature vectors and this might be a possible reason for the weak performance of n-gram feature compared to BoW feature. Another possible reason would be the association of key terms with the vulnerabilities than the term combination. The baseline model (MLP) only outperformed the LR model for BoW, 2-gram and 3-gram features. This is likely due to restricted token types used during the data pre-processing. In contrast, we considered higher number of token types and CST preserve more details than the AST outputs.

Table 2. Performance of multi-class classification ML algorithms with BoW, n-gram, and TF-IDF features (F1-Score)

	NB				**LR**				**RF**				**XGB**			
CWE ID	**BoW**	**2-gram**	**3-gram**	**TF-IDF**	**BoW**	**2-gram**	**3-gram**	**TF-IDF**	**BoW**	**2-gram**	**3-gram**	**TF-IDF**	**BoW**	**2-gram**	**3-gram**	**TF-IDF**
20	0.39	0.39	0.34	0.56	0.63	0.63	0.63	0.70	0.82	0.79	0.74	0.82	0.87	0.83	0.76	0.87
78	0.57	0.57	0.56	0.66	0.78	0.75	0.73	0.83	0.91	0.88	0.84	0.9	0.95	0.91	0.85	0.95
120	0.06	0.34	0.35	0.55	0.59	0.60	0.59	0.62	0.80	0.78	0.75	0.79	0.83	0.82	0.78	0.82
126	0.30	0.32	0.32	0.53	0.58	0.60	0.61	0.66	0.83	0.80	0.75	0.83	0.87	0.84	0.80	0.87
134	0.40	0.43	0.45	0.54	0.65	0.68	0.69	0.69	0.85	0.82	0.80	0.85	0.86	0.84	0.79	0.86
190	0.35	0.29	0.28	0.57	0.70	0.71	0.68	0.73	0.88	0.87	0.83	0.88	0.91	0.89	0.83	0.90
327	0.57	0.53	0.51	0.69	0.87	0.80	0.75	0.84	0.94	0.90	0.85	0.94	0.96	0.91	0.83	0.96
362	0.49	0.50	0.49	0.58	0.71	0.69	0.67	0.71	0.84	0.82	0.79	0.83	0.87	0.84	0.81	0.87
377	0.26	0.23	0.24	0.32	0.36	0.41	0.48	0.62	0.74	0.67	0.62	0.73	0.86	0.72	0.65	0.85
398	0.70	0.73	0.74	0.74	0.86	0.87	0.87	0.86	0.93	0.92	0.91	0.93	0.94	0.94	0.92	0.93
401	0.39	0.42	0.43	0.43	0.42	0.54	0.59	0.62	0.78	0.76	0.73	0.77	0.79	0.80	0.77	0.79
457	0.39	0.40	0.44	0.57	0.65	0.67	0.68	0.69	0.84	0.83	0.81	0.84	0.84	0.82	0.78	0.83
476	0.30	0.32	0.33	0.23	0.40	0.47	0.54	0.47	0.77	0.76	0.75	0.78	0.72	0.72	0.69	0.71
562	0.30	0.31	0.29	0.17	0.47	0.50	0.56	0.38	0.77	0.77	0.76	0.76	0.70	0.71	0.70	0.69
664	0.26	0.26	0.27	0.21	0.34	0.38	0.51	0.48	0.77	0.76	0.74	0.77	0.81	0.82	0.79	0.82
676	0.50	0.48	0.45	0.49	0.79	0.73	0.68	0.80	0.92	0.88	0.80	0.92	0.97	0.91	0.83	0.96
732	0.36	0.40	0.40	0.48	0.66	0.61	0.64	0.70	0.85	0.81	0.75	0.85	0.91	0.89	0.80	0.91
758	0.52	0.53	0.52	0.63	0.70	0.73	0.78	0.76	0.92	0.92	0.91	0.92	0.89	0.87	0.83	0.89
775	0.27	0.27	0.30	0.44	0.38	0.44	0.52	0.52	0.68	0.66	0.64	0.66	0.72	0.73	0.71	0.70
788	0.10	0.29	0.33	0.23	0.16	0.21	0.30	0.43	0.66	0.67	0.65	0.65	0.64	0.67	0.64	0.63
Overall	0.37	0.40	0.40	0.48	0.59	0.60	0.62	0.66	0.82	0.80	0.77	0.82	0.85	0.82	0.78	0.84

CWE-IDs which has over 20,000 training samples achieved over 0.8 F1-Score. However, some CWE-IDs such as 676 achieved higher F1-Score regardless of the small sample size. Usage of vulnerable tokens such as strcat(), strcpy() and sprintf() lead to the CWE-ID 676 vulnerability. Therefore, it is possible to learn these types of patterns well even with a smaller dataset due to the frequent appearance of vulnerable tokens. To evaluate the impotence of dataset size for vulnerability detection, we trained a separate XGB model with the BoW feature by only considering the CWE-IDs which had over 20,000 samples. Table 3 summarizes the performance achieved for this model. As expected, this increased the overall F1-Score by 4%. Therefore, it is possible to improve the detection performance by increasing the dataset size.

Table 3. Performance of XGB algorithm with BoW for 12 classes (F1-Score)

CWE ID	120	126	134	190	208	327	362	398	457	758	780	Other	**Overall**
F1-Score	0.8	0.88	0.86	0.9	0.87	0.96	0.87	0.94	0.83	0.88	0.96	0.89	0.89

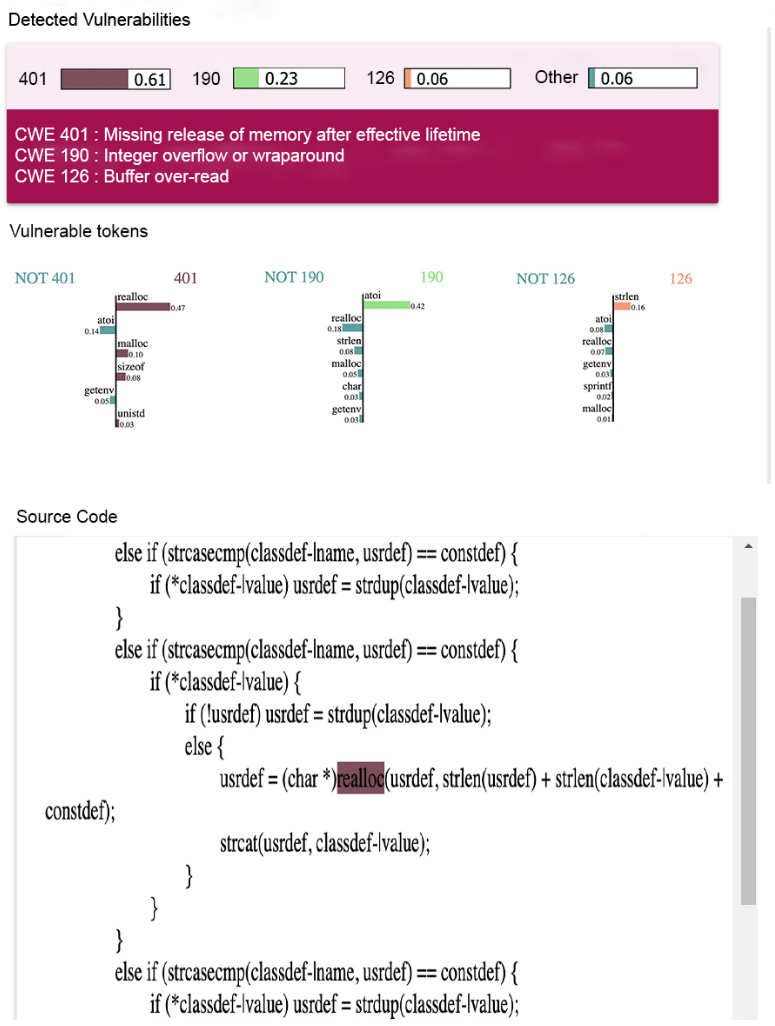
In the deployed web portal, developers are expected to get the prediction with the minimum time and highest detection rate. Therefore, detection latency is another important aspect of the source code vulnerability detection model in a real-world environment. This was estimated for the BoW feature due to its higher vulnerability detection capability. Table 4 summarizes the average detection latency (ms) per source code sample for the three ML models and the baseline model. LR takes the minimum time for the prediction. RF takes higher time which is not suitable for deployment. In contrast, XGB provides the best detection and latency trade-off by outperforming the baseline model.

Table 4. Average detection latency

ML Algorithm	**Detection latency (ms)**
MLP	36.41
LR	8.378
RF	175.968
XGB	**14.37**

4.2 Explainable AI and Web Portal Output

Based on the achieved F1-Score and detection latency, we deployed the RF as a binary classifier and XGB as the multi-class clarifier in the web portal backend. Therefore, LIME used the RF as the classifier to give the prediction explanation. In the deployed web portal, the developer gets the highlighted code as the output for the given input source code. Even though the CWE-ID annotation was done at the multi-class level by assigning one CWE-ID for one source code sample, in practice, multiple CWE-IDs can appear in the same source code due to the parent-child relationship of CWE-IDs. Since LIME provides the explanation for top K prediction, LIME has the capability to visualize multiple CWE-IDs based on their probability. Therefore, the developer can identify multiple CWE-IDs in the output if the input source code has multiple vulnerabilities. Figure 4 presents a part of XAI output which display on the web portal for a given source code.



CWE 401 : Missing release of memory after effective lifetime
CWE 190 : Integer overflow or wraparound
CWE 126 : Buffer over-read

```
        else if (strcasecmp(classdef-lname, usrdef) == constdef) {
            if (*classdef-lvalue) usrdef = strdup(classdef-lvalue);
        }
        else if (strcasecmp(classdef-lname, usrdef) == constdef) {
            if (*classdef-lvalue) {
                if (!usrdef) usrdef = strdup(classdef-lvalue);
                else {
                    usrdef = (char *)realloc(usrdef, strlen(usrdef) + strlen(classdef-lvalue) +
constdef);
                    strcat(usrdef, classdef-lvalue);
                }
            }
        }
        else if (strcasecmp(classdef-lname, usrdef) == constdef) {
            if (*classdef-lvalue) usrdef = strdup(classdef-lvalue);
```

Fig. 4. Web Portal Output

This code has the CWE-ID 401 vulnerability, which is missing release of memory after an effective lifetime (known as the memory leak). It is expected that developers to track and release allocated memory after it has been used. XGB accurately predicts the CWE-ID 401 as the most probable vulnerability with a 0.61 probability for this code. This is shown in brown colour and tokens which causes this vulnerability also highlighted in the same colour in the pre-processed source code. These tokens are 'realloc', 'malloc', 'sizeof' and 'unistd'. Most probable token is 'realloc' with a 0.47 probability. Annotated ground truth for this code was CWE-ID 401. However, as the second most probable vulnerability, this predicts the CWE-ID 190 with a 0.23 probability. This is related to the use of the function 'atoi' inappropriately. The domain experts analysed the code and also confirmed that CWE-ID 190 also lies in this code even though it has not been annotated as a ground truth. Inappropriate usage of token 'strlen' also highlights that vulnerability CWE-ID 126 is presented in this code. Even if the input is a large source code file with a large number of lines, developers can quickly identify the vulnerable code segments using the provided colour codes. Additionally, this shows a brief description for detected CWE-IDs so that developers can quickly identify the reason and make it benign.

The web portal has the facility to confirm the predicticed CWE-IDs using the developer's domain knowledge. This change reflects in the source code database by annotating the correct ground truth CWE-IDs for the input source code (as shown in the Figure 2). This human-in-the-loop process allows for incremental improvement of the model's accuracy over time through model retraining, as well as adaptation to changes in the properties of the data (concept drift). Additionally, this approach produces a human-annotated dataset which helps to overcome the limitations of static code analysers. Therefore, the proposed model has higher capability to outperform static code analysers which used for data annotation.

5 Conclusion and Future Works

The vulnerabilities of source code need to be minimized to avoid the critical security flaws which lead to numerous impactful consequences. However, existing solutions for source code vulnerability detections suffer from high false negatives and low generalization capability. Additionally, these solutions do not provide the reasons for vulnerabilities which is an important aspect of source code vulnerability detection. As a solution, this paper proposed AI-based vulnerability detection method for C and C++ languages which achieved 0.96 F1-Score for the binary classification (with RF classifier) and 0.85 F1-Score for the multi-class classification (with XGB classifier) to detect vulnerable CWE-IDs. Additionally, XAI which is based on LIME provides visual explanations for detected vulnerabilities. The effectiveness of vulnerability detection relies significantly on the size of the dataset. Hence, it becomes imperative to employ a well-chosen representative sample from the dataset that encompasses many source codes characterising diverse representations of the same vulnerability. Moreover, the selection of datasets should span across various code repositories to mitigate any inclinations towards particular CWE-IDs, thereby reducing bias. BoW features showed the effectiveness of the feature regardless of its simplicity. Detection capability for the n-grams might increase with the dataset size as it reduces the data sparsity.

The improvement of the model's detection capability can be achieved by taking into account the XAI outputs. The current version of the model has the capacity to detect up to 20 CWE-IDs. However, our plan is to enhance the model's performance by training it with a larger dataset, including data gathered from developers, which is anticipated to result in a higher vulnerability detection for a greater number of CWE-IDs. Additionally, we intend to offer benign code segments for the identified vulnerable code segments to developers as an extension of the deployed model. In addition, we aim to extend our research to other programming languages, such as Java and Python, to provide a more comprehensive developer support system that aligns with the realities of secure software development practices.

References

1. Barredo Arrieta, A., Díaz-Rodríguez, N., Del Ser, J., Bennetot, A., Tabik, S., Barbado, A., Garcia, S., Gil-Lopez, S., Molina, D., Benjamins, R., Chatila, R., Herrera, F.: Explainable artificial intelligence (xai): Concepts, taxonomies, opportunities and challenges toward responsible ai. Information Fusion **58**, 82–115 (2020). https://doi.org/https://doi.org/10.1016/j.inffus.2019.12.012, https://www.sciencedirect.com/science/article/pii/S1566253519308103
2. Bilgin, Z., Ersoy, M.A., Soykan, E.U., Tomur, E., Çomak, P., Karaçay, L.: Vulnerability prediction from source code using machine learning. IEEE Access **8**, 150672–150684 (2020)
3. Dam, H.K., Tran, T., Pham, T., Ng, S.W., Grundy, J., Ghose, A.: Automatic feature learning for vulnerability prediction. arXiv preprint arXiv:1708.02368 (2017)
4. Du, X., Chen, B., Li, Y., Guo, J., Zhou, Y., Liu, Y., Jiang, Y.: Leopard: Identifying vulnerable code for vulnerability assessment through program metrics. In: 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE). pp. 60–71. IEEE (2019)
5. Feng, H., Fu, X., Sun, H., Wang, H., Zhang, Y.: Efficient vulnerability detection based on abstract syntax tree and deep learning. In: IEEE INFOCOM 2020 - IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS). pp. 722–727 (2020). https://doi.org/10.1109/INFOCOMWKSHPS50562.2020.9163061
6. Fujdiak, R., Mlynek, P., Mrnustik, P., Barabas, M., Blazek, P., Borcik, F., Misurec, J.: Managing the secure software development. In: 2019 10th IFIP International Conference on New Technologies, Mobility and Security (NTMS). pp. 1–4 (2019). https://doi.org/10.1109/NTMS.2019.8763845
7. Grieco, G., Grinblat, G.L., Uzal, L., Rawat, S., Feist, J., Mounier, L.: Toward large-scale vulnerability discovery using machine learning. In: Proceedings of the Sixth ACM Conference on Data and Application Security and Privacy. pp. 85–96 (2016)
8. Harer, J.A., Kim, L.Y., Russell, R.L., Ozdemir, O., Kosta, L.R., Rangamani, A., Hamilton, L.H., Centeno, G.I., Key, J.R., Ellingwood, P.M., et al.: Automated software vulnerability detection with machine learning. arXiv preprint arXiv:1803.04497 (2018)

9. Jimenez, M.: Evaluating vulnerability prediction models (2018), https://orbilu.uni.lu/handle/10993/36869
10. Pereira, J.D., Vieira, M.: On the use of open-source c/c++ static analysis tools in large projects. In: 2020 16th European Dependable Computing Conference (EDCC). pp. 97–102. IEEE (2020). https://doi.org/10.1109/EDCC51268.2020.00025
11. Pimpalkar, A.P., Retna Raj, R.J.: Influence of pre-processing strategies on the performance of ml classifiers exploiting tf-idf and bow features. ADCAIJ: Advances in Distributed Computing and Artificial Intelligence Journal **9**(2), 49–68 (Jun 2020). https://doi.org/10.14201/ADCAIJ2020924968
12. Rajapaksha, S., Senanayake, J., Kalutarage, H., Al-Kadri, M.O.: Ai-powered vulnerability detection for secure source code development. In: Innovative Security Solutions for Information Technology and Communications. pp. 275–288. Springer Nature Switzerland, Cham (2023). https://doi.org/10.1007/978-3-031-32636-3˙16
13. Renaud, K.: Human-centred cyber secure software engineering. Zeitschrift für Arbeitswissenschaft pp. 1–11 (2022)
14. Russell, R., Kim, L., Hamilton, L., Lazovich, T., Harer, J., Ozdemir, O., Ellingwood, P., McConley, M.: Automated vulnerability detection in source code using deep representation learning. In: 2018 17th IEEE international conference on machine learning and applications (ICMLA). pp. 757–762. IEEE (2018)
15. Scandariato, R., Walden, J., Hovsepyan, A., Joosen, W.: Predicting vulnerable software components via text mining. IEEE Transactions on Software Engineering **40**(10), 993–1006 (2014)
16. Senanayake, J., Kalutarage, H., Al-Kadri, M.O., Petrovski, A., Piras, L.: Developing secured android applications by mitigating code vulnerabilities with machine learning. In: Proceedings of the 2022 ACM on Asia Conference on Computer and Communications Security. p. 1255–1257. ASIA CCS '22, Association for Computing Machinery, New York, NY, USA (2022). https://doi.org/10.1145/3488932.3527290
17. Senanayake, J., Kalutarage, H., Al-Kadri, M.O., Petrovski, A., Piras, L.: Android code vulnerabilities early detection using ai-powered acved plugin. In: Atluri, V., Ferrara, A.L. (eds.) Data and Applications Security and Privacy XXXVII. pp. 1–19. Springer International Publishing, Cham (2023). https://doi.org/10.1007/978-3-031-37586-6˙20
18. Senanayake, J., Kalutarage, H., Al-Kadri, M.O., Petrovski, A., Piras, L.: Android source code vulnerability detection: A systematic literature review. ACM Comput. Surv. **55**(9) (jan 2023). https://doi.org/10.1145/3556974
19. de Vicente Mohino, J., Bermejo Higuera, J., Bermejo Higuera, J.R., Sicilia Montalvo, J.A.: The application of a new secure software development life cycle (s-sdlc) with agile methodologies. Electronics **8**(11) (2019). https://doi.org/10.3390/electronics8111218
20. Votipka, D., Fulton, K.R., Parker, J., Hou, M., Mazurek, M.L., Hicks, M.: Understanding security mistakes developers make: Qualitative analysis from build it, break it, fix it. In: 29th USENIX Security Symposium (USENIX Security 20). pp. 109–126. USENIX Association (Aug 2020)
21. Zeng, P., Lin, G., Pan, L., Tai, Y., Zhang, J.: Software vulnerability analysis and discovery using deep learning techniques: A survey. IEEE Access (2020)
22. Zhou, Y., Liu, S., Siow, J., Du, X., Liu, Y.: Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks. In: NeurIPS (2019)