AI-powered vulnerability detection for secure source code development.

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

2023

AI-Powered Vulnerability Detection for Secure Source Code Development

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
https://www.rgu.ac.uk/
[2] School of Computing and Digital Technology, Birmingham City University, Birmingham B5 5JU, UK
omar.alkadri@bcu.ac.uk

Abstract. Vulnerable source code in software applications is causing paramount reliability and security issues. Software security principles should be integrated to reduce these issues at the early stages of the development lifecycle. Artificial Intelligence (AI) could be applied to detect vulnerabilities in source code. In this research, a Machine Learning (ML) based method is proposed to detect source code vulnerabilities in C/C++ applications. Furthermore, Explainable AI (XAI) was applied to support developers in identifying vulnerable source code tokens and understanding their causes. The proposed model can detect whether the code is vulnerable or not in binary classification with 0.96 F1-Score. In case of vulnerability type detection, a multi-class classification based on CWE-ID, the model achieved 0.85 F1-Score. Several ML classifiers were tested, and the Random Forest (RF) and Extreme Gradient Boosting (XGB) performed well in binary and multi-class approaches respectively. Since the model is trained on a dataset containing actual source codes, the model is highly generalizable.

Keywords: source code vulnerability · machine learning · software security · vulnerability scanners

1 Introduction

Security threats evolve rapidly, forcing developers to be up to date with the latest security vulnerabilities to minimize the risk of software attacks. Education of security for developers is an ongoing process. To date, many software developers have overlooked security issues throughout the software development lifecycle [22, 24]. One of the main reasons for this could be a possible lack of understanding about how common errors in software development result in exploitable vulnerabilities in software systems [15] and possible pressure towards

fast deployment. Also, the communication disconnection between developers and cyber security experts has led to widespread software vulnerabilities [26].

Traditional security tools and penetration-testing techniques are considered very complicated, time-consuming and expensive processes in dynamically changing cyber attacks [14]. For example, one of the challenges businesses face today is that the mandate to be agile and release software faster while ensuring that their product is secure against cyber threats. Possible other solutions include static code analysis tools, which can have low detection capability (high false negative rate) due to the lack of up-to-date cyber attack data [9, 14]. Therefore, the software development industry is in definite need of automating vulnerability detection with the growing impact of cyber attacks on businesses due to downtime, reputation damage, loss of customers and asset sabotage.

Due to the advancement in computational power, new algorithms and availability of data, AI and ML can be successfully used to address problems in various domains. Many applications in the computer security and privacy domain, have been addressed using AI/ML techniques [28]. Software vulnerabilities are such area in which AI/ML algorithms can be used to detect vulnerabilities in source codes [1, 2, 19]. In the context of vulnerability detection, use of AI/ML algorithms help to reduce the need of human expertise [29] and automate the process. Programming languages can be considered as languages with words, numbers and different symbols. Hence previous works have used Natural Language Processing (NLP) techniques to detect vulnerabilities in source code, treating code as a form of texts [5]. Extracted features through NLP techniques are used to train AI/ML algorithms to model this problem as a classification model.

A requirement of having a high accuracy source code vulnerability detection method is fulfilled in this work which used AI/ML techniques. In summary, the following contributions are made:

- *Improved data pre-processing approach to identify important features:* Presenting a method using a Concrete Syntax Trees (CST) to identify the most important features of source codes to train a ML model.
- *Generalized vulnerability detection models:* Source code vulnerability detection using binary and multi-class classification 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.
- *Model explainability:* 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 for optimising the data pre-processing approach to improving the model accuracy.

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 directions are discussed in Section 5.

2 Background and Related Work

This section sets the base for the study by providing a sound knowledge of source code vulnerabilities and weaknesses, various pasrers and scanners, and various vulnerability detection methods while discussing the related studies.

2.1 Source Code Vulnerabilities and Weaknesses

There is a wide scope of human error within the software development process, especially if an extensive testing and validation process is not followed from the initial stage of the software development lifecycle [7]. Due to these potential human errors, several vulnerabilities in the code can occur. Reducing vulnerabilities in source code is identified as a good practice in secure software development [23].

Source code weaknesses are flaws, bugs, faults, or other errors that, if left unaddressed, could result in the software being vulnerable to attack. Software source code weaknesses are identified in Common Weakness Enumeration (CWE) [3] and the known vulnerabilities are identified in Common Vulnerabilities and Exposures (CVE) [4]. Identifying weaknesses in source code at early stages, make the software less vulnerable. Some weaknesses have relationships with other weaknesses (parent-child relationship in CWE category). Therefore, there can be overlaps of codes related to more than one CWE ID (i.e. CWE-120 and CWE-126 are related to buffer sizes).

2.2 Parsers and Scanners

Software developers require supportive tools which can be integrated with their coding to minimize developer errors by detecting vulnerabilities at an initial step to mitigate them after performing the source code analysis [22]. The source code needs to be initially formatted into a generalized form with CST or Abstract Syntax Trees (AST) [25]. Static analysis can be used [9] to create these syntax trees. The rate of false alarms on vulnerabilities depends on the accuracy of formulating the CST/AST and its generalisation mechanism. Tree-sitter[3] is an open-source parser generator tool which can create a CST for a source file. It also can efficiently update the tree when there is a change in the source code.

Using the parsed code, scanners can be used to perform analysis. Few scanners are available which can perform analysis in C/C++ source code with relatively good accuracy [17]. Cppcheck[4] is one of the open source static analysis tools to detect bugs, undefined behaviour and dangerous coding constructs in C/C++ code. It can provide the following data for each alert: filename, line, severity, alert identifier, and CWE. This also can be integrated with other development tools. Flawfinder[5] is another open source tool that can examine C/C++

[3] https://tree-sitter.github.io/tree-sitter
[4] https://cppcheck.sourceforge.io
[5] https://github.com/david-a-wheeler/flawfinder

source code and report possible security weaknesses. It works by using 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 metacharacter dangers (exec, system, popen), poor random number acquisition (random), and race conditions (access, chown, chgrp, chmod, tmpfile, tmpnam, tempnam, and mktemp).

2.3 Vulnerability Detection Methods

Metric-based and pattern-based techniques have been used in previous works [6] for vulnerability detection. Metric-based techniques use supervised or unsupervised machine learning algorithms using features such as complexity metrics, code churn metrics, token frequency metrics, dependency metrics, developer activity metrics or execution complexity metrics [6]. Pattern-based techniques use static analysis to identify vulnerable codes using known vulnerable codes. However, the technique used in this, limited to function level codes and considered as a pre-step for vulnerability assessment as the proposed solution did not identify the vulnerability type or the possible location of the vulnerability. Additionally, usage of metric based features in compared ML algorithms showed a low detection capability.

Authors in [10] have used text features in source code to predict software defects. They have considered everything as texts separated by space or tab except comments. Naive Bayes (NB) and Logistic Regression (LR) were used as the classification algorithms in this study. This concept was adapted by [20] and used for software vulnerability prediction tasks using the same algorithms with Bag of Words (BoW) as features. Everything except for comment words separated by space or tab have been treated as features for this model. Experimental results showed a lower F1-Score for all selected test cases. This might be due to the poor feature selection without focusing on the proper data pre-processing approach. In [8] n-gram (1-gram, 2-gram and 3-gram) and word2vec were used as the features to predict if a test case contains vulnerability or not. As a solution to the class imbalance problem, the authors used the random oversampling technique. However, both of the above-mentioned models [8, 16] are limited to binary classification models to detect the vulnerability states.

Minimum intermediate representation learning was used to source code vulnerability detection in [19]. Unsupervised learning was used in the pre-training stage to solve the lack of vulnerability samples. Convolutions Neural Networks (CNN) were used to generate high-level features. Finally, these features are used in classifiers such as LR, NB, Support Vector Machine (SVM), Multi-Layer Perceptron (MLP), Gradient Boosting (GB), Decision Tree (DT) and RF for vulnerability detection. Only two CWE-IDs of a synthetic dataset were selected as the training dataset and therefore, this model has a low generalization capability for other CWE-IDs and real datasets.

Authors in [27] proposed a method to guide manual source code analysis using vulnerability extrapolation. To this end, the authors generated AST using a parser. This work is limited to vulnerabilities present in a few source code

functions. The vulnerability detection method proposed by [1] is also based on the AST representation of source code. Pycparser[6] library was used to generate AST for the C language. It was modelled as a binary classification task using MLP and CNN algorithms. The proposed model used four CWE classes and achieved between 0.09 to 0.59 F1-Score.

Though the trend toward applying ML for vulnerability detection is high, as discussed above, many studies do not provide a high accuracy/F1-Score when detecting source code vulnerabilities. Many of them were not trained on a dataset that includes a real-world dataset, following enhanced preprocessing techniques. Furthermore, they were only limited to binary classification or a limited number of CWE classes. Therefore, our study addresses these problems by using a real-world dataset to achieve an F1-Score of 0.96 in the binary class model and 0.85 in the multi-class classification model for twenty CWE classes.

3 Methodology

3.1 Dataset

Lack of vulnerability dataset is one of the major challenges for developing vulnerability prediction model [11, 21]. Authors in [12] showed the importance of using sufficient and accurately labelled data to achieve good accuracy of the vulnerability prediction task. Previous works used different datasets to train proposed algorithms. The proposed method in [20] has used data of 182 releases of 20 apps. It used a source code analyser to identify vulnerabilities without using a vulnerability database. Datasets published by Software Assurance Reference (SARD) and the National Vulnerability Database (NVD)[7] used in [19]. To identify the ground truth of mined open-source code, the authors used static analysis, dynamic analysis with commit-message and bug-report tagging. SARD is a dataset produced by the National Institute of Standards and Technology (NIST) as a result of the Software Assurance Metrics And Tool Evaluation (SAMATE) project[8]. SATE IV juilet test suit[9] of SAMATE project, debian linux distribution and data on public git repositories on GitHub were used in [1].

Since this research focuses on predicting both vulnerable and non-vulnerable codes and detecting the CWE-IDs, both positive and negative classes data is needed. As the vulnerable dataset, synthetic test cases (C and C++ languages) of SATE IV juilet test suit was selected. This dataset was developed to encourage the improvement of static code analysers. The selected dataset includes 52,185 source code samples. Since this dataset is limited to vulnerable codes and CWE-IDs distribution is highly imbalanced, a web crawler was developed to retrieve more C and C++ source codes from public GitHub repositories. The entire

[6] https://github.com/eliben/pycparser
[7] https://cve.mitre.org/
[8] https://samate.nist.gov/SARD/
[9] https://www.nist.gov/itl/ssd/software-quality-group/static-analysis-tool-exposition-sate-iv

source code was considered as a sample. Existing static analysis tools were used to identify the ground truth of the retrieved source codes. In general, signature-based detection methods have lower false positives. Since they might suffer from higher false negatives, they were used in an ensemble way to obtain the ground truth. The main objective here is to learn the capabilities of these analysers and obtain a lower false negative and positive rate from the ML-based models. To this end, the sample was considered as malicious if one of the analysers identifies the sample as malicious. If all analysers identify the sample as benign, then it was considered as benign. Based on the combined dataset of SATE IV Juilet test suit and GitHub data, twenty highest frequent CWE-IDs were selected as vulnerable code samples, including over 0.3 million source codes. Vulnerability class distribution is depicted in Figure 1. Similar size of C and C++ source codes were selected as the benign dataset, making it approximately 1:1 positive and negative class distribution.

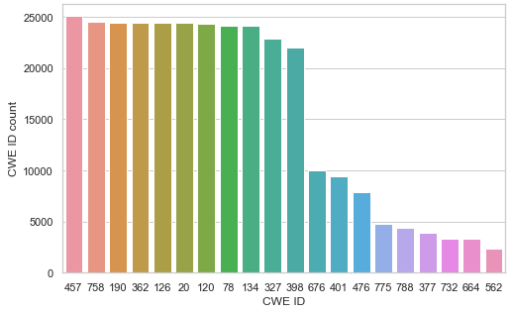

Fig. 1. CWE-ID distribution

3.2 Model Architecture

The proposed model includes two machine learning models for binary and multi-class classifications. The binary classification model is trained to detect the source code as benign or vulnerable code. Multi-class classification model uses the identified vulnerable code to detect the CWE-IDs associated with it. The XAI is used on the multi-class classification results to explain the model prediction and hence to identify vulnerable code segments. This process is depicted in Figure 2.

Data Pre-Processing The selected dataset contains C and C++ language source codes. Previous works [1, 13, 27] used AST and CST representations of codes to identify features. In this research, CST is used to identify the tokens of source code to retain more details in the code using a parser generator tool. Following pre-processing steps were applied to source codes.

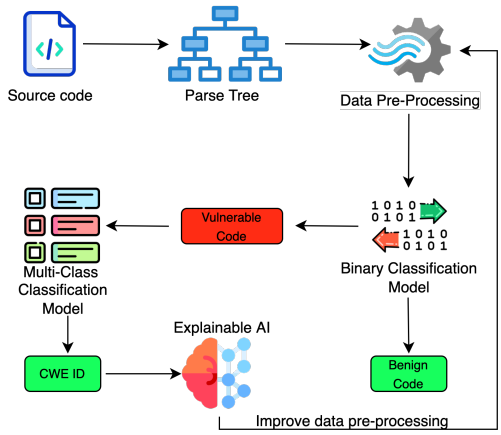

Fig. 2. Model Architecture

1. Use a parser generator to generate CSTs (parse tree) of source codes.
2. Clean CST outputs to generate tokens.
3. Create numerical vectors for ML models.

All source codes were passed through the parser generator to generate CSTs. CSTs contain much information, such as comments, symbols, hexadecimal numbers and user-defined function names, which cannot use as generalized features for machine learning models. Hence, comments and selected symbols were removed and user-defined function names were replaced with common names such as 'UserDef'. Symbols to remove from the codes were identified with the support of a set of domain experts to avoid important symbol removal. Pre-processed CST outputs were used to generate features for ML models. To this end, Python library CountVectorizer and TfidfVectorizer were used to generate features of Bag-of-words (BoW), n-gram (n=2,3) and term frequency-inverse document frequency (TF-IDF). Grid search was used to identify optimal hyperparameters including maximum (max df) and minimum (min df) document frequencies.

Algorithms: Data pre-processing produced three feature vectors from CST tokens: BoW, n-gram and TF-IDF. The complete dataset of 0.6 million source code samples was used to train the binary classification model, whereas 0.3 million source code samples which included 20 CWE-IDs were used to train the multi-class classification model. Due to class imbalance, stratified random sampling was used to split the dataset into 80:20 ratios for the multi-class classification model. 80% of the data was used to train binary and multi-class algorithms and the rest of the 20% was used to evaluate their performance. NB, RF, LR and XGB algorithms were used with BOW, n-gram (n=1,2) and TF-IDF features. 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. Experiments were conducted on a MacBook Pro 2.2 GHz Intel Core i7 with 16 GB RAM.

Vulnerability explanation: Identifying the vulnerabilities and relevant CWE IDs are not sufficient to convert the code into benign code. Identifying the specific code segments (tokens) is helpful in evaluating the validity of model predictions and making the necessary changes to the vulnerable code to make it a benign code. This helps the developer to use the domain knowledge to make an informed decision. Hence, model interpretability is an important factor in source code vulnerability detection. To this end, Local Interpretable Model-agnostic Explanations (LIME) [18] was used. LIME provides an explanation which is a local linear approximation of the trained model's behaviour [18]. LIME learns a sparse linear model by sampling instances around specific instances, approximating the trained model locally. LIME supports text classifiers and provides visual and textual artefacts that developers can understand. These explanations were used to further fine-tune data pre-processing by removing non-related tokens and keeping the important tokens. In addition, LIME provides the explanation for top K predictions, which helps to identify multiple vulnerabilities of a code.

4 Performance Evaluation

This section presents the results for different classifiers and features described in the previous section. All the results discussed in this section were based on the test set. As mentioned earlier, F1-Score was selected as the evaluation metric as it is the harmonic mean of precision and recall. Labels 0 and 1 represent benign and vulnerable classes of the binary classification model, whereas twenty CWE-IDs represent vulnerability classes of the multi-class classification model.

4.1 Machine Learning Models

As mentioned in the previous section, four machine learning algorithms were used to predict vulnerabilities using three features. N-gram includes 2-gram and 3-gram. Table 1 summarize the F1-Score for binary classification models for BoW, 2-gram, 3-gram and TF-IDF features. The best model performance was obtained with the default parameters. The BoW feature achieved a higher or similar F1-Score than n-gram for all algorithms except the XGB algorithm. XGB algorithm showed a very low detection capability for benign class for all features, even with different hyper-parameters. This might be due to the large number of available hyperparameters of XGB, and the selected grid search values were out of the optimum values for the binary classification. The RF algorithm achieved a significantly higher F1-Score than other algorithms for TF-IDF. RF algorithm with the feature BoW outperformed all other algorithms and features and achieved 0.96 F1-Score.

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 achieved by multi-class algorithms with respective features. NB algorithm achieved the lowest F1-Score for all features. For both NB and LR algorithms, increasing the n-gram caused to achieve the same F1-Score as BoW or slight detection improvement. In contrast, the opposite was observed for RF and XGB algorithms. F1-Score was reduced when increasing the n-gram. RF and XGB algorithms for BoW and TF-IDF showed nearly similar detection capabilities.

According to the results, the best overall F1-Score was obtained as 0.85 for XGB algorithm with BoW features. Overall, BoW features performed better than the n-gram features. Generally, higher n-gram models contain more information about the word (token) contexts. However, this increases the data sparsity with the n. This might be one possible reason for the lower F1-Score for n-gram based models compared to BoW based models. Another possible reason would be the association of key terms with the vulnerabilities than the term combinations. Combining these key terms with the nearby terms might reduce the vulnerability detection capability. CWE-IDs which had over 20,000 source code samples, achieved over 0.80 F1-Scores, whereas other classes showed comparatively low detection capability. However, CWE-ID 676 detection rate is higher for all algorithms regardless of the dataset size. Usage of potentially dangerous functions such as strcat(), strcpy() and sprintf() introduce the CWE-ID 676 vulnerability. The frequent appearance of these vulnerable terms could be a reason for the higher detection rate.

Multi-class classification results indicate the detection rate likely to be associated with the dataset size for each class. To verify this, all classes over 20000 samples were considered and the remaining classes were categorized as 'other' category. This produced 12 unique classes compared to 20 classes used in the previous model. The best performing XGB with BoW feature was used to evaluate the performance. Table 3 presents the performance achieved for increased sample size. As expected, this improved the overall F1-Score by 4%.

Detection latency is a critical criterion in a production environment for real-time predictions. This highly depends on the number of features used and model complexity. Since BoW provided the best detection rate, BoW was used to evaluate the detection latency of four ML algorithms. Table 4 presents the average detection latency (ms) for one source code. NB provides the prediction in a very

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

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

short time with lower detection rates. In contrast, RF takes much time despite a higher detection rate. Overall, among the selected algorithms, XGB provides the best detection latency and detection rate tradeoff.

Table 4. Average detection latency

ML Algorithm	Detection latency (ms)
NB	0.005
LR	8.378
RF	175.968
XGB	**14.378**

4.2 Explainable AI

Even though ML algorithms with BoW showed a higher detection rate, this is not much useful unless the reasons behind these predictions are known. Hence, LIME was used to identify the vulnerable code segments of each source code and potential other CWE-IDs which were not available as a ground truth. This is highly important as multiple CWE-IDs might be there due to parent-child relationships. The selected example presented in Figure 3 includes CWE-ID 401 as the vulnerability. This is relevant to the missing release of memory after an effective Lifetime. Developers should sufficiently track and release allocated memory after it has been used [3]. XGB accurately predicts the CWE-ID 401 as the vulnerability of this source code. LIME provides the prediction probabilities for the top 4 predictions and respective features (tokens) that caused the vulnerability. Further, LIME provides the visualization of highlighted code. Since the original codes were pre-processed, this shows the pre-processed code. In this example, it identified that 'realloc', 'malloc', 'sizeof' and 'unistd' positively affect towards CWE-ID 401. These tokens are highlighted with brown colour in the code. Even though the ground truth was 401, as expected, this identified other possible vulnerabilities as well. CWE-ID 190 is another vulnerability that lies in this code due to inappropriate usage of function 'atoi'. Additionally, inappropriate usage of 'strlen' leads to CWE-ID 126, also identified by the algorithm as the 3rd possible vulnerability.

Based on these features, the developer can examine the code regardless of its number of code lines and convert the vulnerable code into benign code by changing the respective feature usage. These explanations also can be used to optimise the feature pre-processing. There might be some features that are not useful to predict the vulnerability and still, the algorithm identifies them as valid features due to dataset bias. These features can be identified by analysing the LIME output, which helps to perform the required pre-processing to remove such features continuously.

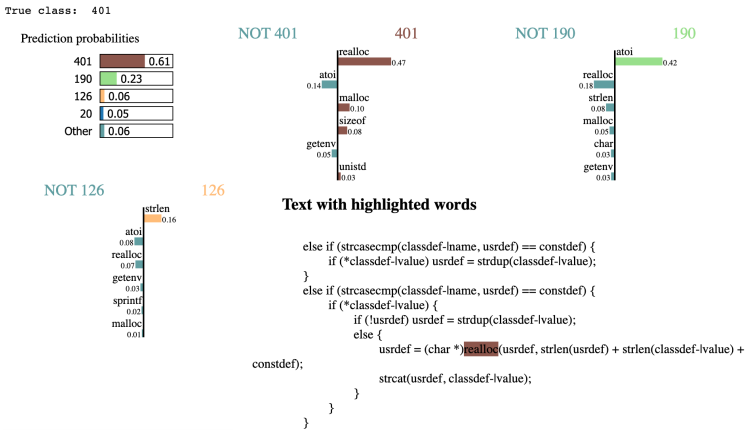

Fig. 3. Explainability of the predictions

5 Conclusion and Future Works

Vulnerable source code sometime can cause critical security flaws. Therefore, the weaknesses of the source code must be reduced to a great extent. Though a few methods are available to detect source code vulnerabilities, their accuracies and generalization capabilities are low. Existing methods do not provide reasons for the vulnerabilities, which is very important to the developers. The proposed method in this work can detect source code vulnerabilities in C/C++ using an ML-based approach with an F1-Score of 0.96 in binary classification (with RF classifier) and an F1-Score of 0.85 in CWE-ID-based multi-class classification (with XGB classifier). Furthermore, XAI was also applied in this work to explain the causes of particular vulnerabilities. The F1-Score can be further increased by improving the data pre-processing techniques and extending the dataset with more source code examples. Currently, the CWE-ID based multi-class classification model can detect twenty types of weaknesses, and by increasing the sample source code, it can detect more classes with higher accuracy and improve the detection capability for the extremely broad vulnerability categories. An automated solution to perform that is also integrated with a live web portal. Once the dataset contains a high volume of data, it can also be explored as a future improvement since there can be vulnerable source code associated with more than one CWE-ID. Once the vulnerabilities are detected, mitigation methods can also be proposed by integrating more features in XAI for future improvement. Finally, the model will be deployed with a live web portal to validate under real-world settings.

Acknowledgment

This work has been funded by The Scottish Funding Council, we are thankful to the funder for their support.

Appendix: Common Weaknesses in C/C++ Source Code

CWE-ID	**CWE-Name**	**Sample Vulnerable C/C++ Code**
CWE-20	Improper Input Validation	board = (board_square_t*) malloc(m * n * sizeof(board_square_t));
CWE-78	Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection')	system(NULL)
CWE-120	Buffer Copy without Checking Size of Input ('Classic Buffer Overflow')	strcpy(buf, string);
CWE-126	Buffer Over-read	strncpy(Filename, argv[1], sizeof(Filename));
CWE-134	Use of Externally-Controlled Format String	snprintf(buf,128,argv[1]);
CWE-190	Integer Overflow or Wraparound response	xmalloc(nresp*sizeof(char*));
CWE-327	Use of a Broken or Risky Cryptographic Algorithm	EVP_des_ecb();
CWE-362	Concurrent Execution using Shared Resource with Improper Synchronization ('Race Condition')	pthread_mutex_lock(mutex);
CWE-401	Missing Release of Memory after Effective Lifetime	char buf = (char) malloc(BLOCK_SIZE); read(fd, buf, BLOCK_SIZE) != BLOCK_SIZE;
CWE-457	Use of Uninitialized Variable	char *test_string; if (i != err_val) test_string = "Hello World!"; printf("%s", test_string);
CWE-676	Use of Potentially Dangerous Function	char buf[24]; strcpy(buf, string);

References

1. 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)
2. Chakraborty, S., Krishna, R., Ding, Y., Ray, B.: Deep learning based vulnerability detection: Are we there yet? IEEE Transactions on Software Engineering **48**(9), 3280–3296 (2022). https://doi.org/10.1109/TSE.2021.3087402
3. Corporation, M.: Common weakness enumeration (cwe) (2022), https://cwe.mitre.org/, accessed: 2022-02-01
4. Corporation, M.: Cve details (2022), https://www.cvedetails.com/, accessed: 2022-02-01
5. 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)
6. 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)
7. 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
8. 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)
9. 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)
10. Hata, H., Mizuno, O., Kikuno, T.: Fault-prone module detection using large-scale text features based on spam filtering. Empirical Software Engineering **15**(2), 147–165 (2010)
11. Jimenez, M.: Evaluating vulnerability prediction models. Ph.D. thesis, University of Luxembourg, Luxembourg (2018)
12. Jimenez, M., Rwemalika, R., Papadakis, M., Sarro, F., Le Traon, Y., Harman, M.: The importance of accounting for real-world labelling when predicting software vulnerabilities. In: Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. pp. 695–705 (2019)
13. Jin, Z., Yu, Y.: Current and future research of machine learning based vulnerability detection. In: 2018 Eighth International Conference on Instrumentation & Measurement, Computer, Communication and Control (IMCCC). pp. 1562–1566 (2018). https://doi.org/10.1109/IMCCC.2018.00322
14. Li, Z., Zou, D., Xu, S., Ou, X., Jin, H., Wang, S., Deng, Z., Zhong, Y.: Vuldeepecker: A deep learning-based system for vulnerability detection. arXiv preprint arXiv:1801.01681 (2018)
15. Morgan, S.: Is poor software development the biggest cyber threat. htt ps://www. csoonline. com/article/297885 **8** (2015)

16. Pang, Y., Xue, X., Namin, A.S.: Predicting vulnerable software components through n-gram analysis and statistical feature selection. In: 2015 IEEE 14th International Conference on Machine Learning and Applications (ICMLA). pp. 543–548 (2015). https://doi.org/10.1109/ICMLA.2015.99
17. 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
18. Ribeiro, M.T., Singh, S., Guestrin, C.: " why should i trust you?" explaining the predictions of any classifier. In: Proceedings of the 22nd ACM SIGKDD international conference on knowledge discovery and data mining. pp. 1135–1144 (2016)
19. 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)
20. 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)
21. Senanayake, J., Kalutarage, H., Al-Kadri, M.O.: Android mobile malware detection using machine learning: A systematic review. Electronics **10**(13) (2021). https://doi.org/10.3390/electronics10131606, https://www.mdpi.com/2079-9292/10/13/1606
22. Senanayake, J., Kalutarage, H., Al-Kadri, M.O., Petrovski, A., Piras, L.: Android source code vulnerability detection: A systematic literature review. ACM Comput. Surv. (aug 2022). https://doi.org/10.1145/3556974, https://doi.org/10.1145/3556974, just Accepted
23. 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, https://doi.org/10.1145/3488932.3527290
24. Tahaei, M., Vaniea, K.: A survey on developer-centred security. In: 2019 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW). pp. 129–138 (2019). https://doi.org/10.1109/EuroSPW.2019.00021
25. Wile, D.S.: Abstract syntax from concrete syntax. In: Proceedings of the 19th international conference on Software engineering. pp. 472–480 (1997)
26. Xie, J., Lipford, H.R., Chu, B.: Why do programmers make security errors? In: 2011 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC). pp. 161–164 (2011). https://doi.org/10.1109/VLHCC.2011.6070393
27. Yamaguchi, F., Lottmann, M., Rieck, K.: Generalized vulnerability extrapolation using abstract syntax trees. In: Proceedings of the 28th Annual Computer Security Applications Conference. pp. 359–368 (2012)
28. Zeng, P., Lin, G., Pan, L., Tai, Y., Zhang, J.: Software vulnerability analysis and discovery using deep learning techniques: A survey. IEEE Access (2020)
29. 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)