electronics

MDPI

Article

Reduction of False Positives for Runtime Errors in C/C++ Software: A Comparative Study

Jihyun Park [1], Jaeyoung Shin [2] and Byoungju Choi [2,*]

[1] Department of Artificial Intelligence and Software, Ewha Womans University, Seoul 03760, Republic of Korea; pola0527@ewhain.net
[2] Department of Computer Science & Engineering, Ewha Womans University, Seoul 03760, Republic of Korea; njs05036@ewhain.net
* Correspondence: bjchoi@ewha.ac.kr; Tel.: +82-2-3277-3508

Abstract: In software development, early defect detection using static analysis can be performed without executing the source code. However, defects are detected on a non-execution basis, thus resulting in a higher ratio of false positives. Recently, studies have been conducted to effectively perform static analyses using machine learning (ML) and deep learning (DL) technologies. This study examines the techniques for detecting runtime errors used in existing static analysis tools and the causes and rates of false positives. It analyzes the latest static analysis technologies that apply machine learning/deep learning to decrease false positives and compares them with existing technologies in terms of effectiveness and performance. In addition, machine-learning/deep-learning-based defect detection techniques were implemented in experimental environments and real-world software to determine their effectiveness in real-world software.

Keywords: static analysis; early defect detection; machine learning; deep learning; false positive rate

Citation: Park, J.; Shin, J.; Choi, B. Reduction of False Positives for Runtime Errors in C/C++ Software: A Comparative Study. *Electronics* **2023**, *12*, 3518. https://doi.org/10.3390/electronics12163518

Academic Editor: Martin Reisslein

Received: 7 July 2023
Revised: 10 August 2023
Accepted: 17 August 2023
Published: 20 August 2023

1. Introduction

Testing is a critical activity for assuring software quality and can cost greater than 50% of the total cost of a software development project [1]. Among software testing methods, static analysis is a technique of examining coding rules, code complexities, runtime errors, or security vulnerabilities by analyzing the source code without executing software. Various automated tools such as Coverity, CodeSonar, Infer, and Splint have been used for static analyses [2–5]. Runtime error analysis utilizing static analysis tools examine errors in syntax, type (variable or function argument), unused code, memory, synchronization, and security vulnerabilities. Thus, developers can detect and fix defects in the early stages. However, static analysis tools are underutilized in practice [6–8].

The high rate of false positives when using these tools is among the primary factors for this underutilization [8,9]. Considering the static analysis of runtime errors, errors are primarily detected based on rules and not executed results. Because errors are detected based on source code patterns and coding rules, false positives, in which the tool reports errors when none exist, occur often [8,9]. Accordingly, developers or testers must review the static analysis results and determine whether they are false positives; this requires considerable investment in time and effort.

Various studies have been conducted over the past 20 years to reduce false positives. Researchers have introduced new rules to diversify the types of detectable errors. Additionally, advancements have been made in improving accuracy through semantic analysis of source code rather than just pattern matching. In particular, recent studies have applied machine learning and deep learning techniques to conduct effective static analysis. By training on existing error types and extracting features from target source code, these approaches enhance accuracy. They also leverage machine learning/deep learning to eliminate false positives in error types identified by static analysis tools.

This study analyzes the latest static analysis techniques that apply ML/DL to decrease false positives. In particular, it examines the techniques for detecting runtime errors used in existing static analysis tools and the causes and rates of false positives. This study compares advanced static analysis techniques that apply ML/DL to decrease false positives with existing techniques in terms of effectiveness regarding false positive rate and performance and analyzes whether the ML/DL applied method is effective in the actual software execution environment.

Various static analysis tools are available for open-source or commercial use; however, this study was performed with the most extensively used open-source static analysis tool based on C/C++. Furthermore, the top five most frequently occurring defect types were detected based on Common Weakness Enumeration (CWE), a software vulnerability classification system for runtime error types employed in research to examine the effects of static analysis tools. CWE defects encompass flaws, vulnerabilities, bugs, faults, and other errors in software implementation, code, design, and architecture [10]. According to CWE's examination of C/C++-based software defect types as of 2022, the top five defects include memory buffer errors, out-of-bound read/write, use-after-free, and improper synchronization. This study applies static analysis tools to these five types of defects and compares their defect detection effectiveness.

The following are the primary contributions of this study.

(1) It demonstrates the detection of runtime errors in static analysis, focusing on certain defects. It analyzes the types of defects detected in runtime over the last 20 years as indicated by published studies. The findings reveal that most reported static analyses focused on defects such as memory buffer errors and incorrect memory release.
(2) It analyzes the latest static analysis techniques to decrease false positives by categorizing them into those that apply ML/DL and those that do not. A comparative analysis of the defect detection effects of each technique shows that the application of ML/DL affects the reduction in false positives in static analysis.
(3) It measures the time taken to apply the latest static analysis techniques to the test data, identifying whether ML/DL can be applied to actually defect detection.
(4) It demonstrates that measures to reduce false positives in static analysis focus on limited types of vulnerabilities/defects; it thus emphasizes the necessity for approaches in static analysis that accurately detect a wide range of vulnerabilities/defects.

The remainder of this study is organized as follows: Section 2 addresses the techniques for analyzing runtime errors and the causes of false positives in a static analysis. In Section 3, static analysis tools are used for performing assessments of a variety of runtime errors. Section 4 analyzes the effects of the techniques on reduction in false positives in static analysis. Section 5 provides the conclusions and describes future research directions.

2. Related Studies

In general, an automated static analysis tool (AST) is used for static analysis. These ASTs detect runtime errors or vulnerabilities through pattern-based, dataflow-based, annotation-based, and constraint-based approaches [11].

Pattern-based ASTs, such as Flawfinder [10] and cppcheck [12], analyze source code based on pre-determined classification rules, that is, code patterns, and report mismatched patterns as defects. Hence, conventional codes implemented with a non-predetermined pattern are reported as defects, resulting in a high false positive rate; these ASTs as such cannot detect new types of defects that are not in a pre-determined pattern. Furthermore, they cannot identify semantic errors (e.g., data-type mismatches) as defects. These limitations led Frama-C [13] to add rules for new defect types. Therefore, new security vulnerability patterns can be detected; however, the false positive rate is still high.

A Clang static analyzer [14] detects defects through semantic analysis. Because defects are detected based on the data flow, the defect detection rate is higher than existing pattern-based ASTs. However, the false positive rate is also high because the data flow cannot be analyzed based on actual execution.

Rahimi and Zargham [15] proposed a technique for extracting code metrics, such as code complexity and compliance with security coding rules, from source code using compiler-based static analysis to predict vulnerabilities using these code metrics.

However, as in all the aforementioned studies, as the analysis is not based on actual execution, the false positive rate is high. Because the actual execution result is unknown, it is necessary to presume that the software is operating or utilizing an approximation. Therefore, things may be reported as a defect, although it is not. Such a high false-positive rate makes using AST for developing programs challenging. To identify actual defects, developers must review all reports on defects as many may contain cases that are not defects; this is time-consuming.

Artificial intelligence technology has recently garnered prominence, and research on static analysis techniques employing ML or DL to decrease false positives or remove defects from the static analysis has been performed. As a result of our analysis of papers published in conferences or journals for 10 years from 2012 through IEEExplore, we discoverd that 2130 papers focused on predicting or detecting defects and vulnerabilities using static analysis with a related restoration theme. Particularly, 24 of the 38 relevant papers published in the preceding year examined techniques for analyzing vulnerabilities using ML or DL, demonstrating a wide range of applications.

There are various techniques and approaches to detecting defects using ML/DL-based static analysis. Generally, these approaches involve the following steps: data collection, feature engineering, data preprocessing, model training, and vulnerability detection using the trained model. Data for both defective and normal code is collected, and features are extracted from the collected code. These features quantify code structures, variable usages, function call relationships, and program control flow. The data were preprocessed to make it suitable for application to the models of ML/DL. Various ML/DL models are trained on this preprocessed data to predict vulnerabilities. Models such as Support Vector Machines (SVMs), Random Forests, Gradient Boosting, and Neural Networks are commonly used.

Refs. [16,17] used a support vector machine (SVM) to detect vulnerabilities. This model predicts vulnerabilities or components with vulnerabilities by extracting code features and statistically analyzing them.

Various techniques can be used to predict vulnerabilities using deep learning. Tree-based convolution neural network (TBCNN) [18] is a technique to extract features from the source code as a tree structure and predict vulnerabilities using a convolution neural network (CNN). Using recurrent neural network (RNN), VulDeePecker [19] made source code "code gadget" divided source code semantically to detect vulnerabilities through learning. VulDeePecker has been extended to μVulDeePecker [20] and SySeVR [21]. There is also a technique for detecting vulnerabilities using an ensemble of CNN and RNN rather than using them alone. To detect vulnerabilities, [22] extracted features from lexically analyzed function source code and proposed a random forest technique that uses a classifier with an ensemble of CNN and RNN.

Studies have been conducted to reduce the false-positive rate when predicting vulnerabilities with deep learning. VulDeeLocator [23] attempts to increase the accuracy of vulnerability analysis by eliminating false alarms using deep learning based on source code analysis. VulDeBERT [24] applied the BERT model to vulnerability logs collected from the static analysis tools to eliminate false positives and enhance the accuracy of vulnerability analysis.

This study compares these analytical tools regarding the vulnerabilities they target and identifies their effects by learning the same dataset and applying it to the test set.

3. Runtime Error Types and Dataset

Table 1 lists the results of the statistical analyses. Most commercial tools can detect almost all the vulnerabilities defined by the CWE standard. However, open-source-based static analysis tools are limited to particular vulnerabilities, such as memory corruption and null pointer reference.

Table 1. Types of defects detectable by static analysis tools.

Defect	Static Analysis Tools					
	Flawfinder	**Cppcheck**	**RATS**	**Infer**	**Frama-C**	**Coverity (Commercial)**
Dead pointers		O				O
Division by zero		O			O	O
Integer overflows	O	O			O	O
Invalid bit shift operands		O				O
Invalid conversions		O				O
Invalid usage of STL		O				O
Memory management		O			O	O
Null pointer dereferences		O		O		O
Out of bounds checking	O	O				O
Uninitialized variables		O				O
Writing const data		O				O
Coding conventions				O		O
Unavailable API's				O		O
Memory out-of-bound read/write	O	O	O		O	O
Race Condition within a Thread	O					O
Memory leakage		O		O		O
Use of Potentially Dangerous Function	O					O

Most defects detectable by the tools occur frequently in the software. According to a CWE standard-based analysis of the most frequently occurring defect types in C/C++-based software as of 2022, the top five were memory buffer error, out-of-bound read/write, use-after-free, and improper synchronization defects. The CWE-ID related to these type is shown in Table 2, and the detectable defects by AST are shown in Table 3.

Table 2. Most frequently occurring defect types.

CWE-ID	CWE Title	Defect Type
404	Improper resource shutdown or release	Improper synchronization
476	NULL pointer dereference	use after free
119	Improper restriction of operations within the bounds of memory	Memory buffer error Out-of-bound read Out-of-bound write

This study used two datasets to conduct a comparative analysis of the ASTs. One was acquired from the National Institute of Standards and Technology (NIST) Software Assurance Reference Data Set (SARD), which includes the source code of various vulnerabilities based on the CWE. The second database was obtained from National Vulnerability Database (NVD). The NVD contains information such as program, time, vulnerability description, CWE ID, hyperlinks to patches for the security-related vulnerability list, CVE, and a subset of the CWE.

As shown in Tables 4 and 5, the datasets consisted of a training set for learning vulnerability detection tools using ML/DL and a test set for measuring vulnerability detection performance. The source code is sliced to learn the ML/DL-applied tools in the

training dataset. Hence, it included 28,561 C/C++ code snippets for 6091 samples related to the CWE defect type targeted in this study.

Table 3. Types of defects detectable by AST.

Tool		CWE ID			Remark
		404	476	119	
Without ML/DL	Flawfinder	O	O	O	
	Cppcheck	O	O	O	
	Frama-C	O	O	O	
	Clang static analyzer	O	O	O	
Vulnerability detection using ML/DL	VulDeePecker	X	X	O	
	μVulDeePecker	X	X	O	
	SySeVR	X	X	O	
	VUDDY	O	O	O	Network security
	HyVulDect	O	O	O	Network security
False positive reduction using ML/DL	VulDeeLocator	X	X	O	
	VulDeBERT	X	X	O	

Table 4. Dataset for training.

Dataset Source	No. of Samples
SARD	4927
NVD	1164
Total	6091

Table 5. Dataset for testing.

Dataset Source	No. of Samples
FFMpeg	637
GNU Grep	380
Total	1017

4. Comparison of Techniques for Reducing False Positives in Static Analysis Tools

4.1. Experimental Environment

We developed the following research questions and conducted experiments to address them.

RQ1. Is applying ML/DL to detect vulnerabilities using static analysis practical?

We classified static analysis techniques for detecting vulnerabilities into existing static analysis techniques and those applying ML/DL to conduct a comparative analysis of vulnerability detection performance. Metrics for measuring the vulnerability analysis performance were defined, and each tool was applied to the test set for the analysis. By comparing the effects of existing techniques and ML/DL-applied techniques, we analyzed the effectiveness of ML/DL for the detection of vulnerabilities.

RQ2. Can vulnerability detection techniques using ML/DL be applied to detect actual software vulnerabilities?

Previous studies demonstrated that vulnerability detection techniques that apply ML/DL have higher levels of accuracy. However, whether these techniques are effective when applied to real-world software rather than a dataset for experiments is unknown. This study compared the results of vulnerability detection when adjusting the size of the training

dataset and analyzed the software requirements for training or vulnerability detection to identify whether vulnerabilities can be detected effectively in real-world software.

(1) Experimental Environment

The environment for the experiments consisted of a 64-bit Ubuntu 20.04.3, Intel(R) Core (TM) i5-6200U CPU, 2.4 GHz, NVIDIA Geforce 940MX, and 2 GB RAM. The training data for vulnerability detection techniques based on ML/DL was preprocessed using low-level virtual machine (LLVM), Clang, and Python 3.7 environments.

(2) Techniques for Comparison

The vulnerability-detection techniques are described in Section 2. In the experiments, open-source software techniques were compared for the C/C++ language, as shown in Table 6. µVulDeePecker, extended from VulDeePecker, was excluded from the experiments because of the existing latest SySeVR technique. VUDDY and HyVulDect were excluded from the experiment because they are tools for detecting vulnerabilities related to network security rather than software defects.

Table 6. Dataset for testing.

Name	Target Language	Open Source	Selection for Comparison
Flawfinder	C/C++	O	O
Cppcheck	C/C++	O	O
Frama-C	C	O	X
Clang static analyzer	C/C++/Objective C	O	X
VulDeePecker	C/C++	O	O
µVulDeePecker	C/C++	O	X
SySeVR	C/C++	O	O
VUDDY	C/C++	O	X
HyVulDect	C/C++	O	X
VulDeeLocator	C/C++	O	O
VulDeBERT	C/C++	O	O

Flawfinder [10] and cppcheck [12], when given a program, analyze the abstract syntax tree of the input program to verify if it adheres to predefined syntactic rules set by the tools. If the program violates the defined syntactic rules, it is reported as a flaw.

In the process of vulnerability detection, VulDeePecker [19] extracts library/API function calls and their arguments from the program to create semantic units called 'code gadgets'. Each code gadget is labeled for vulnerability and then vectorized. These vectorized code gadgets are trained using a BLSTM model, an extension of the LSTM model, which falls under the category of RNN. When detecting vulnerabilities, code gadgets are created and vectorized similarly from the target program, and the trained BLSTM model is used to determine if a vulnerability exists.

SySeVR [21] extracts Syntax-based Vulnerability Candidates(SyVC) from program syntax and converts them to Semantics-based Vulnerability Candidates(SeVC). SeVC is transformed into vectors for deep neural network training and vulnerability detection using bidirectional RNN, specifically BGRU (Bidirectional Gated Recurrent Unit).

Similarly, VulDeeLocator [23] performs syntactic analysis on the program to extract SyVC, which is then transformed into SeVC. It introduces the concept of granularity refinement during semantic information extraction and utilizes intermediate code-based representation. SeVC is vectorized for deep neural network training and vulnerability detection using bidirectional RNN, including BGRU.

VulDeBERT [24] converts information related to variable and function calls from a program's source code into code gadgets. Ambiguous code gadgets that might be misclassified as vulnerabilities are removed. The remaining code gadgets are encoded with tokens indicating the start and end of input vectors. BERT model is employed for training using these encoded tokens to detect vulnerabilities. By fine-tuning BERT model with C/C++ source code transformed into code gadgets, VulDeBERT aims to enhance vulnerability detection, leveraging the strong performance of BERT in natural language processing tasks.

(3) Measurement Metrics

We measured the performance of each vulnerability detection technique using six standard metrics. A false positive (FP) refers to the number of samples reported as defects, even though they are not. False negatives (FN) refer to vulnerable samples that were undetected as defects. The number of samples that were defects and were reported as such by the tool are referred to as true positives (TP). True negative (TN) is the number of samples that were not defects and were unreported as defects by the tool.

$$\text{False Positive Rate (FPR)} = \text{FP}/(\text{FP} + \text{TN}) \quad (1)$$

$$\text{False Negative Rate (FNR)} = \text{FN}/(\text{TP} + \text{FN}) \quad (2)$$

$$\text{Accuracy (A)} = \text{TP} + \text{TN}/(\text{TP} + \text{FP} + \text{TN} + \text{FN}) \quad (3)$$

$$\text{Precision (P)} = \text{TP}/(\text{TP} + \text{FP}) \quad (4)$$

$$\text{Recall (R)} = (\text{TP})/(\text{TP} + \text{FN}) \quad (5)$$

$$\text{F1 Score} = (2 \times \text{P} \times \text{TP}/(\text{TP} + \text{TN}))/(\text{P} + \text{TP}/(\text{TP} + \text{TN})) \quad (6)$$

(4) Experimental Methods

Experiments were conducted to measure the performance of vulnerability detection techniques and identify their applications to real-world software.

First, experiments were conducted to determine how the performance of each vulnerability detection varied with the application of ML/DL. For the vulnerability detection techniques that apply ML/DL, an additional experiment would be conducted to measure its effects when applied to real-world software. Section 3 describes the datasets used in our experiments. The performance of the existing technique without the application of ML/DL was measured by static analysis of the training dataset. Considering the vulnerability detection technique applying ML/DL, 80% of the training datasets were used as training data and 20% as test data. The experiment was iterated five times and the average value of the metrics measured in each experiment was used as the final performance.

The experiment used all training datasets to determine whether the vulnerability detection technique using ML/DL is applicable to real-world software rather than the one used in the experiment. Once the model training was completed, it was applied to the test dataset, and the results were analyzed.

4.2. Experimental Results and Analyses

The study analyzes the experimental results of each tool by RQ.

4.2.1. RQ1. Is It Effective to Apply Machine Learning/Deep Learning to Detect Vulnerabilities with Static Analysis?

Table 7 presents the results of the vulnerability detection techniques used for the training dataset.

Table 7. Results of vulnerability detection (unit: %).

Tools	Precision	Recall	F1-Score	Accuracy	FPR	FNR
flawfinder	47.39	32.83	37.99	55.49	26.64	70.95
cppcheck	58.05	57.07	57.56	46.06	41.95	57.07
VulDeePecker	81.17	62.14	67.54	69.9	14.77	33.13
SySeVR	79.05	78.38	78.63	82.45	14.15	22.1
VulDeeLocator	90.28	86.86	88.52	90.08	6.25	15.9
VulDeBERT	95.51	90.38	91.95	99.15	0.28	9.62

Existing static analysis tools, such as flawfinder and cppcheck, accurately analyzed only 46–55% of the vulnerabilities. The remaining vulnerabilities were reported as defects, although they were not properly detected. For example, in case 1 of Table 8, it is reported that even though the defect was not present, there could be a buffer overflow due to the lack of size checking on the destination (dst) when using the memcpy function. In case 2, it was not reported as a defect when there was an out-of-bound access to the array of the variable 'pattern' at line 1301, but it is reported as a potential overflow when declaring the variable 'pattern' at line 1294, due to its size causing a possible overflow. This is because existing static analysis tools only detect defects when predefined syntactic rules are violated, and they do not validate the variables or indexes used in the source code.

Table 8. Example source code.

Case	Source Code

1	static void copyset(const int src[((1 << 8) + 8 * sizeof(int) − 1)/(8 * sizeof(int))],charclass dst)
	{
	memcpy(dst,src,sizeof(charclass));
	}

2	else {
	/* Defer to the system regex library about the meaning of range expressions. */
	regex_t re;
	char pattern[6] = {('['), (0), ('-'), (0), (']'), (0)};**//line 1294**
	char subject[2] = {(0), (0)};
	. . .
	pattern[7] = c1; **//line 1301**
	. . .
	}

Vulnerability detection techniques applying ML, such as VulDeePecker, SySeVR, VulDeeLocator, and VulDeVERT, showed an increased accuracy of the detected defects by 14.41–53.09% compared with existing static analysis tools. Particularly, VulDeBERT, which removed false positives after static analysis, accurately reported over 99% of the defects and had a relatively low false positive defect rate of 0.275%.

These findings demonstrate that techniques applying ML/DL, rather than existing static analyses, reduce false positives and increase the reliability of results. Additionally, these ML/DL techniques demonstrated a higher level of vulnerability detection accuracy on average, proving that they are more effective for detecting vulnerabilities than existing static analysis tools.

4.2.2. RQ2. Can Vulnerability Detection Techniques Applying Machine Learning/Deep Learning Be Applied to Vulnerability Detection of Real-World Software?

We deployed VulDeePecker, SySeVR, VulDeeLocator, and VulDeBERT to the real-world software FFMpeg and GNU Grep to identify whether vulnerability detection techniques that apply ML/DL are effective for detecting vulnerabilities in real-world software. To detect vulnerabilities using these techniques, the training dataset must first be learned, and the source code and training data for testing must be preprocessed. However, for VulDeePecker, SySeVR, and VulDeeLocator, pre-processing of the source code cannot be completed or used for analysis if it cannot be compiled. Figure 1 represents the data preprocessing process in SySeVR. Other tools have similar data preprocessing processes as well. When the size of the source code used for training is large, the time required for preprocessing increases. In our experimental environment, preprocessing the SARD dataset for training involved parsing the source code, extracting line numbers of vulnerable lines from the test case information, and creating labels. These steps consumed more than 12 h, and the entire data preprocessing process took around 3 days to complete.

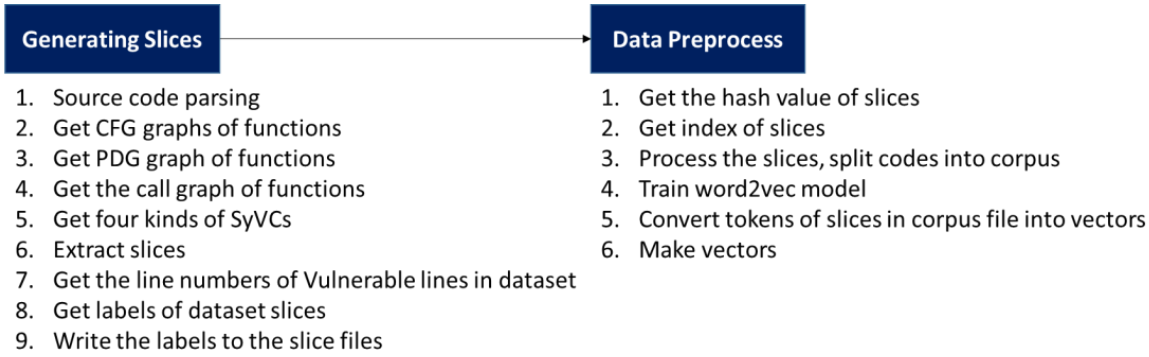

Figure 1. SySeVR data pre-processing process.

Table 9 shows the results of the vulnerability detection for FFMpeg and Grep. Compared with the application to the training dataset at RQ1, the vulnerability detection accuracy decreased, and the false positive rate increased. These results show that the training dataset used in the lab may have insignificant effects on vulnerability detection using real-world software.

Table 9. Results of vulnerability detection for FFMpeg, Grep (unit: %).

Tools	Precision	Recall	F1-Score	Accuracy	FPR	FNR
VulDeePecker	50.65 (6.54)	72.3 (11.3)	50.4 (3.57)	63.25 (10.16)	38.46 (11.81)	32.33 (8.25)
SySeVR	57.6 (11.21)	78.91 (8.07)	63.7 (9.55)	57.52 (12.17)	55.41 (25.86)	27.8 (3.91)
VulDeeLocator	78.41 (2.15)	80.37 (1.68)	79.26 (0.28)	76.68 (2.73)	36.98 (15.8)	19.63 (5.29)
VulDeBERT	90.08 (3.28)	53.43 (23.67)	59.97 (20.9)	85.48 (8.66)	0.83 (0.34)	46.57 (23.67)

Figure 2 shows the Receiver Operating Characteristic (ROC) curves for vulnerability detection results when tested in experimental and real environments after training on the data set for RQ1. The ROC curve is a performance indicator for the model, and a curve closer to the *y*-axis indicates better performance. The training dataset was divided, using 80% of it for training in the experimental environment, and the remaining 20% was used as the test dataset in the same environment. The test dataset in the real-world environment was the dataset from Table 5. The results show that the dataset in the experimental environment had a curve closer to the *y*-axis, indicating better vulnerability detection performance.

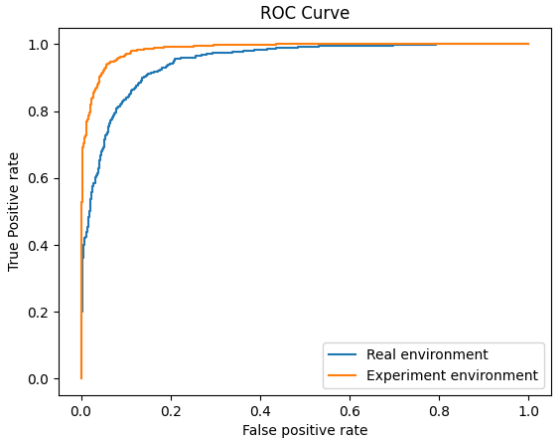

Figure 2. ROC Curve.

The existing static analysis techniques can be applied if the language is the same, regardless of the target software used for vulnerability detection. However, depending on the target software utilized for detection, vulnerability detection techniques based on ML/DL may be inapplicable. Furthermore, data pre-processing is time-consuming, and the false-positive rate may be similar to that of the existing static analysis method according to the training dataset. Hence, more efficient vulnerability detection techniques require application using ML/DL, not in research but in practice.

4.3. Discussion

In this paper, we compared the effectiveness of existing static analysis tools and vulnerability detection approaches using ML/DL for limited vulnerability types, such as memory buffer errors and out-of-bound read/write. The traditional static analysis tools show a high rate of false positives as expected. However, when applying ML/DL approaches, we observed a reduction in false positives and a more accurate detection of vulnerabilities.

Based on accuracy as the evaluation metric, VulDeBERT shows the best performance. Accuracy can be distorted depending on the dataset used in the experiments, so other evaluation metrics should also be analyzed. In RQ1, VulDeBERT outperformed other tools in terms of precision, recall, and F1 score, with the lowest false positive rate. However, in RQ2, although VulDeBERT still exhibited high accuracy and precision, its recall was the lowest. This indicates a higher proportion of undetected vulnerabilities, suggesting the need for improving precision and recall through the application to various datasets.

We focused only on a small subset of all vulnerability types for vulnerability detection. Existing static analysis tools like flawfinder and cppcheck exhibit high false positive rates but can detect various vulnerability types. On the other hand, vulnerability detection approaches using ML/DL like VulDeBERT are limited by the information obtained from preprocessed data and do not currently support detection of various vulnerability types. This means that while VulDeBERT accurately detected vulnerabilities, it does not guarantee high performance for other vulnerability types.

5. Conclusions

This study examined runtime error detection techniques using existing static analysis tools and identified the causes and rates of false positives. It also compared the existing techniques with the latest static analysis techniques that apply ML/DL to reduce false positives and investigated the effectiveness of such techniques regarding the false positive rate and performance.

For open-source static analysis tools based on C/C++, the experiments conducted were to detect memory buffer errors, out-of-bound read/write, and their use after free and improper synchronization, which frequently occurs in CWE. Consequently, compared

with existing static analysis techniques, those applying ML/DL exhibited a higher defect detection accuracy of 14.41–53.09% with a lower false-positive rate of 0.28–24.53%. However, static analysis techniques that employ ML/DL have varied effects depending on the dataset learned. In addition, the time for analysis increased in proportion to the amount of data available in the process of learning data, and the performance was lowered when detecting vulnerabilities in real-world software. Therefore, studies have to be conducted on efficient techniques that can be applied to real-world software rather than to training datasets. In the future, we plan to investigate vulnerability detection techniques that can be applied to real-world software using ML/DL.

Author Contributions: Conceptualization, methodology, J.P. and B.C.; software, J.P.; validation, J.P. and J.S.; formal analysis, investigation, resources, data curation, J.P. and J.S.; writing—original draft preparation, J.P.; writing—review and editing, J.P. and B.C.; visualization, J.P.; supervision, B.C.; project administration, B.C.; funding acquisition, B.C. All authors have read and agreed to the published version of the manuscript.

Funding: This work was supported by Institute of Information and Communications Technology Planning andEvaluation (IITP) grant funded by the Korea government (MSIT) (No. RS-2022-00155966, Artificial Intelligence Convergence Innovation Human Resources Development (Ewha Womans University)).

Data Availability Statement: Data is available on each tool's github, NIST, and SARD homepages.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Khaleel, S.I.; Anan, R. A review paper: Optimal test cases for regression testing using artificial intelligent techniques. *Int. J. Electr. Comput. Eng.* **2023**, *13*, 1803–1816. [CrossRef]
2. Coverity. Available online: https://scan.coverity.com/ (accessed on 6 July 2023).
3. CodeSonar. Available online: https://www.grammatech.com/our-products/codesonar/ (accessed on 6 July 2023).
4. Infer. Available online: https://github.com/facebook/infer (accessed on 6 July 2023).
5. Splint. Available online: https://splint.org/ (accessed on 6 July 2023).
6. Johnson, B.; Song, Y.; Murphy-Hill, E.; Bowdidge, R. Why don't software developers use static analysis tools to find bugs? In Proceedings of the International Conference on Software Engineering, San Francisco, CA, USA, 18–26 May 2013; pp. 672–681.
7. Christakis, M.; Bird, C. What developers want and need from program analysis: An empirical study. In Proceedings of the International Conference on Automated Software Engineering, Singapore, 3–7 September 2016; pp. 332–343.
8. Beller, M.; Bholanath, R.; McIntosh, S.; Zaidman, A. Analyzing the state of static analysis: A large-scale evaluation in open source software. In Proceedings of the International Conference on Software Analysis, Evolution, and Reengineering, Osaka, Japan, 14–18 March 2016; pp. 470–481.
9. Heckman, S.; Williams, L. A systematic literature review of actionable alert identification techniques for automated static code analysis. *Inf. Softw. Technol.* **2011**, *53*, 363–387. [CrossRef]
10. Flawfinder. Available online: https://dwheeler.com/flawfinder/ (accessed on 6 July 2023).
11. Shahriar, H.; Zulkernine, M. Classification of Static Analysis-Based Buffer Overflow Detectors. In Proceedings of the 2010 Fourth International Conference on Secure Software Integration and Reliability Improvement Companion, Singapore, 9–11 June 2010; pp. 94–101. [CrossRef]
12. Cppcheck. Available online: https://cppcheck.sourceforge.io/ (accessed on 6 July 2023).
13. Frama-C. Available online: https://frama-c.com/ (accessed on 6 July 2023).
14. Kremenek, T. *Finding Software Bugs with the Clang Static Analyzer*; Apple Inc.: Cupertino, CA, USA, 2008.
15. Rahimi, S.; Zargham, M. Vulnerability scrying method for software vulnerability discovery prediction without a vulnerability database. *IEEE Trans. Reliab.* **2013**, *62*, 395–407. [CrossRef]
16. Hovsepyan, A.; Scandariato, R.; Joosen, W.; Walden, J. Software vulnerability prediction using text analysis techniques. In Proceedings of the 4th International Workshop on Security Measurements and Metrics, Lund, Sweden, 21 September 2012; pp. 7–10.
17. Pang, Y.; Xue, X.; Namin, A.S. Predicting vulnerable software components through n-gram analysis and statistical feature selection. In Proceedings of the 2015 IEEE 14th International Conference on Machine Learning and Applications (ICMLA), Miami, FL, USA, 9–11 December 2015; pp. 543–548.
18. Mou, L.; Li, G.; Jin, Z.; Zhang, L.; Wang, T. TBCNN: A tree-based convolutional neural network for programming language processing. *arXiv* **2014**, arXiv:1409.5718.
19. 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* **2018**, arXiv:1801.01681.

20. Zou, D.; Wang, S.; Xu, S.; Li, Z.; Jin, H. μVulDeePecker: A Deep Learning-Based System for Multiclass Vulnerability Detection. *IEEE Trans. Dependable Secur. Comput.* **2019**, *18*, 2224–2236.
21. Li, Z.; Zou, D.; Xu, S.; Jin, H.; Zhu, Y.; Chen, Z. Sysevr: A framework for using deep learning to detect software vulnerabilities. *IEEE Trans. Dependable Secur. Comput.* **2021**, *19*, 2244–2258. [CrossRef]
22. 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 Proceedings of the 2018 17th IEEE International Conference on Machine Learning and Applications (ICMLA), Orlando, FL, USA, 17–20 December 2018; pp. 757–762.
23. Li, Z.; Zou, D.; Xu, S.; Chen, Z.; Zhu, Y.; Jin, H. Vuldeelocator: A deep learning-based fine-grained vulnerability detector. *IEEE Trans. Dependable Secur. Comput.* **2021**, *19*, 2821–2837. [CrossRef]
24. Kim, S.; Choi, J.; Ahmed, M.E.; Nepal, S.; Kim, H. VulDeBERT: A Vulnerability Detection System Using BERT. In Proceedings of the 2022 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW), Charlotte, NC, USA, 31 October–3 November 2022; pp. 69–74.