electronics

MDPI

Article

Detection of Vulnerabilities by Incorrect Use of Variable Using Machine Learning

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
[2] Department of Computer Science & Engineering, Ewha Womans University, Seoul 03760, Republic of Korea
* Correspondence: bjchoi@ewha.ac.kr; Tel.: +82-2-3277-3508

Abstract: Common Weakness Enumeration (CWE) refers to a list of faults caused from software or hardware. The CWE includes the faults related to programming language and security. We propose a technique to detect the vulnerabilities from incorrect use of a variable in C language. There are various static/dynamic methods to detect the variable vulnerabilities. However, when analyzing the vulnerabilities, a static technique causes a lot of false alarms, meaning that there is no fault in the actual implementation. When monitoring the variable via the static analysis, there is a great overhead during execution, so its application is not easy in a real environment. In this paper, we propose a method to reduce false alarms and detect vulnerabilities by performing static analysis and dynamic verification using machine learning. Our method extracts information on variables through static analysis and detects defects through static analysis results and execution monitoring of the variables. In this process, it is determined whether the currently used variable values are valid and whether the variables are used in the correct order by learning the initial values and permissible range of the variables using machine learning techniques. We implemented our method as VVDUM (Variable Vulnerability Detector Using Machine learning). We conducted the comparative experiment with the existing static/dynamic analysis tools. As a result, compared with other tools with the rate of variable vulnerability detection between 9.17~18.5%, ours had that of 89.5%. In particular, VVDUM detects 'defects out of the range of valid' that are difficult to detect with existing methods, and the overhead due to defect detection is small. In addition, there were a few overheads at run time that were caused during data collection for detection of a fault.

Keywords: software fault detection; machine learning; variable vulnerability

Citation: Park, J.; Shin, J.; Choi, B. Detection of Vulnerabilities by Incorrect Use of Variable Using Machine Learning. *Electronics* **2023**, *12*, 1197. https://doi.org/10.3390/electronics12051197

Academic Editor: Claus Pahl

Received: 3 February 2023
Revised: 26 February 2023
Accepted: 28 February 2023
Published: 2 March 2023

1. Introduction

Defects in design or implementation processes are common during software development, which increase the susceptibility of the software or system as a whole. To prevent this, common hardware and software vulnerabilities were identified in Common Weakness Enumeration (CWE) [1]. CWE shows various types of vulnerabilities such as those of programming language or security, and a variable occupies much of vulnerabilities of C language, a programming language.

Typically, software programs use variables to save data. However, when a variable is used incorrectly due to a mistake on the part of the programmer, the operation of the program may be interrupted by various types of defects. For example, the program may malfunction when the value of a variable is initialized incorrectly, is out of the permissible range, or is input incorrectly. If local variables are used without initialization, defects can be detected during compilation. However, in the case of global variables, defects remain undetected even in the absence of initialization codes, as the compiler initializes their values to 0. This can be a problem when certain initial values are already defined, or when variables must or should not be initialized during program execution. According to CWE,

weakness continues to increase due to such variables, i.e., vulnerabilities induced by the use of such variables spread in various ways.

Several methods have been proposed to detect vulnerabilities associated with the incorrect use of variables. In general, vulnerabilities are detected by static analysis on a source code after completing implementation. Here, without the running of software, static analysis measures the complexity or predicts an error that may be generated during run time. Errors during run time include faults related to variables such as 'use of an undeclared variable' or 'use of a variable without initialization'. A program's flow graph or the syntax tree constructed from the source code is used to detect the flows associated with the variables in advance via static analysis. However, since run-time defects detected via static analysis are not based on actual execution, this involves false alarms [2]. Furthermore, since the faults are detected on a non-execution basis, it has a low detection rate for faults such as 'a value out of bound', which may be caused by a changed value of a variable during run time. To overcome the limits of static analysis and increase the accuracy of detection with the decreased false alarm, we conduct both static and dynamic analyses. Through static analysis, we extract info related to a false variable or necessary for fault analysis, such as the function a variable was used and the location where a variable was first declared. Then, we collect actual execution results and detect faults.

Detection of a fault based on the actual execution may decrease false alarms through dynamic verification. However, these incur significant overhead because of the need to collect information or track execution. In particular, the detection of defects caused by variables requires monitoring all locations where variables are used, resulting in significant overhead. Furthermore, existing defects may be difficult to detect if they are not executed. Moreover, when malfunctions are caused by the input of an incorrect value into a variable, validation of the value may be difficult, thereby complicating defect detection. To overcome such a limitation, we intend to decrease execution overhead by extracting a list and information of variables causing false alarm through static analysis and then collecting execution info about such a variable only.

This paper proposes a method to detect a fault caused from the use of a variable at the verification phase after software implementation. It suggests a method for detecting whether there was an actual fault via mixed use of static and dynamic verifications to analyze the vulnerabilities of variables. The proposed method uses static analysis to extract the initial values of the target variables, functions containing code that initializes them, and their permissible ranges. Then, it detects defects based on the static analysis results and execution monitoring. Machine learning (ML) techniques are used to learn the initial values and permissible ranges of variables. Based on this, the validity of the current variable values and that of their ordering are checked. This enables the detection of "defects induced by incorrect values of variables", which are difficult to detect using conventional methods.

The remainder of this paper is structured as follows. In Section 2, previous studies on the topic of defect detection related to variables using ML methods are discussed. In Section 3, we describe the techniques used to detect vulnerabilities caused from incorrect use of a variable in software using our proposed machine learning technique. In Section 4, the effectiveness of the proposed method is verified experimentally, and the conclusions and prospective directions of future research are presented in Section 5.

2. Related Works

CWE provides the software/hardware-related weakness enumeration. The soft-ware-related weakness enumeration is divided by programming language such as C, C++, Java, php, and it may be divided again into various categories such as memory/ concurrency faults, security vulnerability, and wrong control flow. Based on CWE, there exist various methods to detect faults [3–24].

As for CWE, this paper deals with the vulnerabilities caused by the use of a variable and the resulting faults. The faults by use of a variable include that of the undeclared/uninitialized variables and use of the values of invalid or out of bound variables.

Static and dynamic analyses exist to detect these faults. Static analysis-based vulnerability detection methods can be broadly classified into two categories—code similarity-based ones and pattern-based ones. For instance, VUDDY and VulPecker analyze the similarity of codes to detect vulnerabilities [13,14]. They show high accuracy for the locations where vulnerabilities have occurred, but show high false-positive rates because several of the vulnerabilities reported by tools are induced by code replication and defects that have not occurred.

On the other hand, pattern-based vulnerability detection methods use rules for defect detection. CppCheck, Flawfinder, and Coverity are typical examples of rule-based tools [15–17]. However, they also show high false-positive rates.

The static analysis, in which faults are detected based on a non-execution, reports the fault regardless of its actual occurrence and so has a higher ratio of false alarm. In case of such a higher ratio of false alarm, it takes a long time for the developer to analyze whether the reported defect is an actual defect or not. Studies have been conducted to decrease or discover false alarms reported from the static analysis [2,18], but they also have a lower level of detection for faults caused only during the execution of software, such as 'values out of bound' caused by modification by operation or external input during execution. We conducted both static and dynamic analyses to overcome these limitations of static analysis and raise the accuracy of detection with the decreased false alarm.

Dynamic verification means detecting a fault based on actual execution. According to the types of faults occurring dynamically, there are methods to detect a fault of memory or that of concurrency [19,20]. Particularly, as in this paper, there are methods that dynamically detect vulnerabilities, such as those for detecting vulnerabilities using fuzzing or those for detecting faults that cause a crash during the execution of a program [21–23]. However, dynamic verification cannot detect vulnerabilities of a path unexecuted in software because it is based on results of execution, so potential faults may still exist. In addition, since it is based on the results of execution, it must collect data while working together with software that is running. However, overhead makes it difficult to be applied in an actual software execution environment. To overcome these limitations of dynamic verification, we optimized the data to be collected and the location of their collection so that the execution info can be collected in a lightweight manner.

There are also methods or tools that use artificial Intelligence (AI) for vulnerability detection. Those methods include ML-based methods on learning pre-defined patterns, such as Controlflag [24], and deep learning-based techniques [25–28], such as VulDeeLocater, VulDeePecker, SySeVR, and mVulDeePecker. These methods use code similarity or code patterns and show higher rates of vulnerability detection than static analysis-based alternatives. However, these methods cannot detect vulnerabilities induced by the input of incorrect values into variables. This is because they consider patterns of code where values are assigned to the variables, but do not consider their values.

3. Vulnerability Detection Method by Incorrect Use of Variable Using ML

In this paper, we propose a detection method for variable-related vulnerabilities in software. Of the 81 vulnerability types related to the C language listed in CWE, 41 (50.62%) are associated with the use of variables or shared resources. Among them, 28 are related to variables. Although several methods have been proposed for the detection of most vulnerabilities, there are almost no viable methods for the detection of vulnerabilities caused by the incorrect use of variables, such as CWE-128 and CWE-457 [1].

We defined defect types to be detected by classifying vulnerabilities caused by incorrect use of variable values, as presented in Table 1. F_01_UINIT_VAR denotes the defect caused by the use of uninitialized variables. Local variables are usually detected by the compiler, whereas global variables are initialized by it. Therefore, when a local or global variable is not initialized with the correct value, it is classified as a defect. Furthermore, when the initialization codes exist for local variables, the compiler usually does not detect them as defects, except when they are used before initialization because of incorrectly ordered calling functions. F_02_INVALID_INIT refers to the defect in which the value of a variable is

changed to its initial value while using the variable in a function other than its initialization function. F_03_OUT_OF_BOUND refers to the defect in which the value of a variable is out of permissible range, which depends not only on variable type but also the semantically permissible range.

Table 1. Target defects related to incorrect use of variables to be detected.

Defect ID	Description	Related CWE-IDs
F_01_UINIT_VAR	Use of uninitialized variable	CWE-457
F_02_INVALID_INIT	Initialization during the use of the variable	CWE-481
F_03_OUT_OF_BOUND	The variable's value is out of permissible range	CWE-125, 128, 130, 131, 197, 805, 839, 843

Our method is divided into a training phase and a detection phase, as shown in Figure 1. The training phase for vulnerability detection is depicted in Figure 1a, and the detection phase, where the trained model is used to detect vulnerabilities, is depicted in Figure 1b.

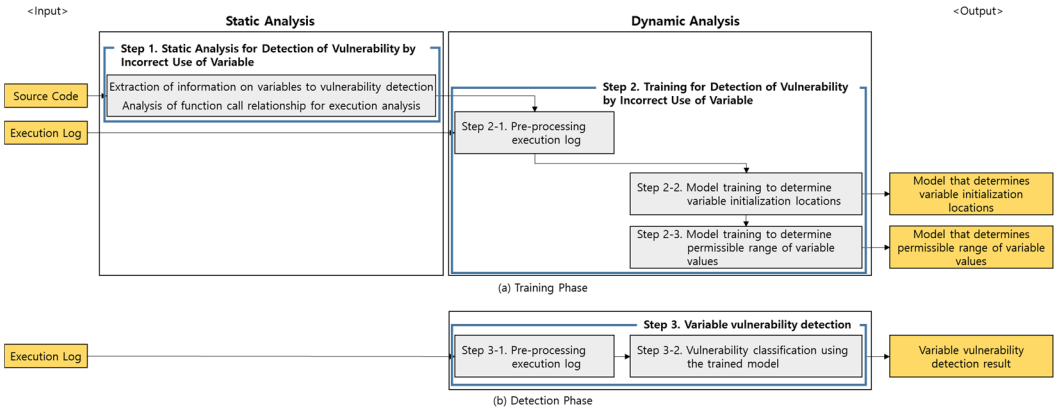

Figure 1. Vulnerability detection method by incorrect use of variable using ML.

3.1. Static Analysis for Detection of Vulnerability by Incorrect Use of Variable

The vulnerability detection targets include all local and global variables used in the program. The detection of the defects listed in Table 1 for these variables requires the definition of the maximum, minimum, and initial values of each variable and the initialization functions, which change the values to the initial values. Before detecting variable vulnerabilities, the source code is statically analyzed to extract the variable list and each variable's initial, maximum, and minimum values as variable information. The list of functions that use the variables and the call relationships between the functions are also extracted for execution analysis.

As shown in Figure 2, when a source code is inputted, after removing all codes irrelevant to analysis such as annotations and blank lines from the source code, the code is read and tokenized. Moreover, the syntax of each token is analyzed to identify whether it is a variable and the value of the variable or a name of function, and then stored in variable information.

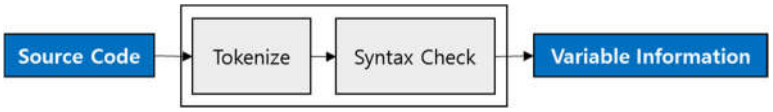

Figure 2. Static Analysis Process.

The maximum/minimum values of the variables and the list of initialization functions, which cannot be defined using static analysis, are defined during training using the execution logs.

3.2. Training for Detection of Vulnerability by Incorrect Use of Variable

After static analysis, the program is executed, and after collecting the execution logs, a training model is constructed based on the execution log. To this end, the execution logs are generalized into training data. Based on the execution logs, models are trained to determine initialization locations and determine permissible ranges of the variable values to detect vulnerabilities and evaluate the appropriateness of the values, respectively. The machine learning algorithm used for training is selected by comparing its effectiveness with that of the existing well-known algorithms.

Pre-processing execution log (Step 2-1) When a variable subject to monitoring is accessed during the operation of the program, the call function location, variable name, the variable values before and after access, and the call stack are included in the execution log. To make the raw data contained in the log more suitable for training, they are pre-processed to replace the call functions, variable names, and the call stack's functions with the respective function IDs and variable IDs.

Model Training to Determine Variable Initialization Locations (Step 2-2) In our method, the criteria for determining vulnerabilities related to the initialization of variables are related to the initialization of the variables, the changes in their values, and the call functions. ML is used to determine whether a function using variables (call function) initializes their values (initialization function). The aim of training is to enable the model to classify "global variable-call function" pairs as initialization functions. To this end, we use the decision tree method, which is a typical ML-based classification model. Various algorithms exist for the decision tree method, such as ID3, C4.5, CART, and ensemble algorithms. We compared the prediction performance of each algorithm using the NASA Nearest Object dataset, which is a dataset similar to the raw data type for prediction of the initialization function. Since RandomForest performs the best, as depicted in Figure 3, it is used to learn the initialization functions for variables.

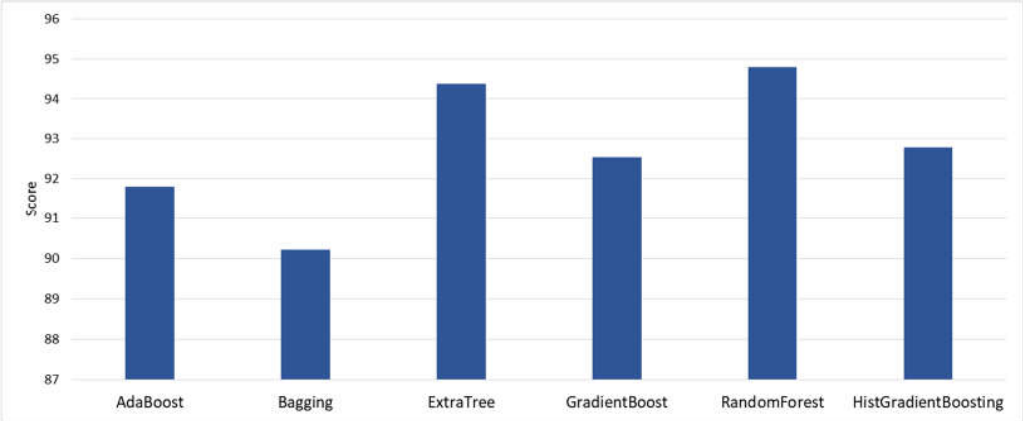

Figure 3. Comparison of algorithms for determining the initialization functions.

Model Training to Determine Permissible Ranges of Variable Values (Step 2-3) Determining permissible ranges of variable values refers to learning the maximum and minimum values that can be taken by a variable and determining whether the current value is in between them. Since our method records a log of changes in variable values whenever a variable is used, the maximum and minimum values are derived by regression analysis of the change trend of variable values. There are various regression-based learning methods, such as linear, ridge, logistic, and polynomial methods. The Motor Trend Car Road Tests [29] dataset was used to compare the various methods in terms of effectiveness—the results are depicted in Figure 4. The polynomial method shows the highest score; thus, polynomial regression was used to determine the permissible range by calculating the maximum and minimum values of the variable. Polynomial regression is a type of linear

regression, and training is performed by applying LinearRegression after converting data into scikit-learn's polynomialFeatures class.

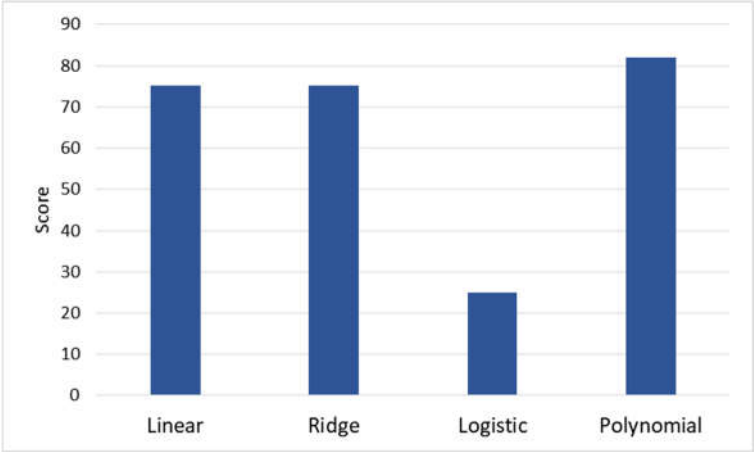

Figure 4. Comparison of algorithms for determining permissible ranges of variable values.

3.3. Variable Vulnerability Detection

To detect defects related to variable vulnerabilities, we use the "variable-initialization function" pair (the corresponding training is performed using the model used to determine the initialization locations of variables in Section 3.2) and each variable's maximum and minimum values (the corresponding training is performed using the model used to determine the permissible ranges of variable values). The data structure depicted in Figure 5 is used to detect defects.

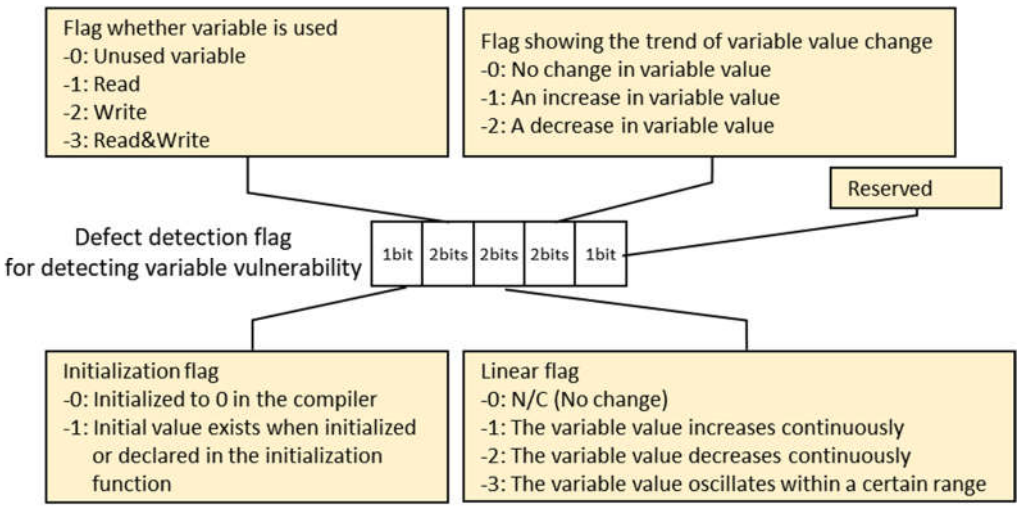

Figure 5. Defect detection flag.

Defects induced by variable vulnerabilities are determined according to the following defect detection algorithm illustrated in Figure 6. Here, the defect detection flag depicted in Figure 5 is used.

F_01_UINIT_VAR: Use of Uninitialized Variable

When the value of a variable is not initialized simultaneously with the declaration, 'the uninitialized variable use' defect is identified if the variable is used by a function other than the initialization function or if the variable's value is read at the location where it is used for the first time. Since the model can be trained to detect whether the function writing a value to a variable is an initialization function, it is determined as a candidate for an 'uninitialized variable use' defect.

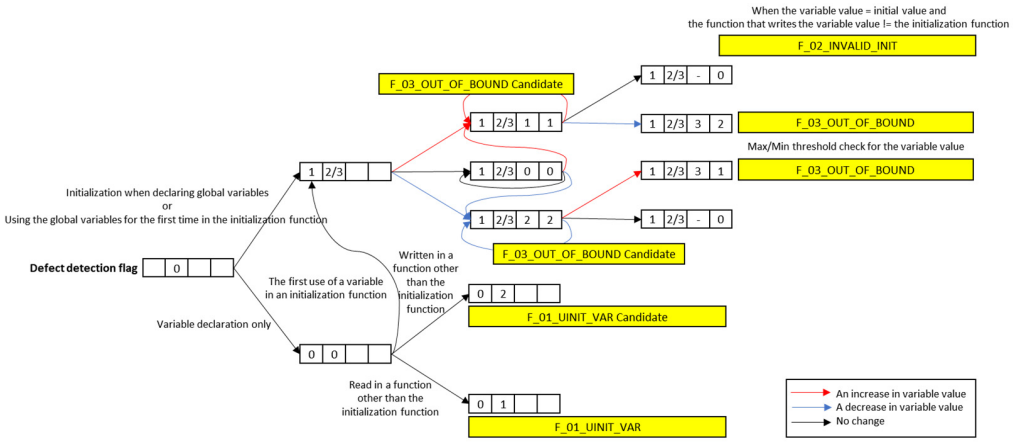

Figure 6. Defect detection algorithm.

F_02_INVALID_INIT: Initialization During the Use of Variable

When there exists an initial value of the variable, if the value that changed during the use of the global variable is the initial value and the calling function is not the initialization function, then the defect is identified to be initialization during the use of a global variable. In particular, if the value of a global variable changes to the initial value after increasing or decreasing continuously, a defect is flagged; otherwise, it is only flagged as a defect candidate.

F_03_OUT_OF_BOUND: The Variable Value Lies Outside the Permissible Range

During the extraction of a defect detection target, the global variable's name, data type, and initial value are extracted. If the value lies outside the range demarcated by the maximum and minimum values of the extracted data type, the defect is determined to be an access outside the range of the variable. As depicted in Figure 6, a flag is used to manage whether the variable's value continues to increase or decrease. However, if the value is set even after the completion of all executions, it can be concluded that the variable was not initialized during its use. Moreover, since its value can deviate from its permissible range when used for a long time, this case is determined as a defect candidate corresponding to access outside the variable's range.

Furthermore, when the variable's maximum and minimum thresholds are set based on training, an 'access outside the variable's range' defect is identified whenever the variable's value transcends the thresholds.

3.4. Automation

We implemented our method as VVDUM (Variable Vulnerability Detector Using Machine learning). The C language and Python are used as illustrated in Figure 7.

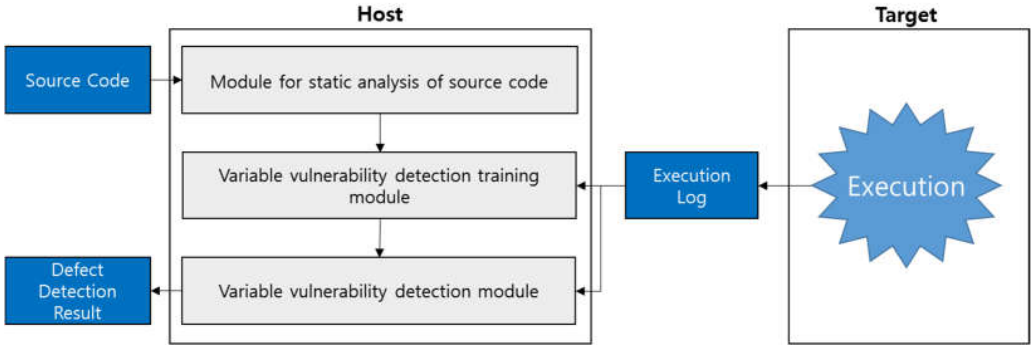

Figure 7. VVDUM (Variable Vulnerability Detector Using Machine learning).

The module for the static analysis of a source code tokenizes the code inputted and performs the syntactic analysis on each token. As a result of the syntactic analysis, it extracts

a variable, variable data type, function name, and variable enumeration used in function. In case the variable was initialized at the time of declaration, its initial value is also extracted. Here, the maximum/minimum values of each variable are the maximum/minimum permissible range according to the data type of the variable. The initialized function comes to also be stored in the variable information and in this module function is to be added to the initialized function enumeration of the variable only when the declaration and initialization of the variable occur at the identical function.

After the static analysis of the source code, execution logs are collected by running the software to detect an actual fault on the target.

The variable vulnerability detection training module and the variable vulnerability detection module train the model using the execution logs collected during execution at the target site to predict the initialization function list and the permissible range of each variable for variable vulnerability detection and detect defects based on this information, respectively. Figure 8 illustrates the internal learning processes of these modules.

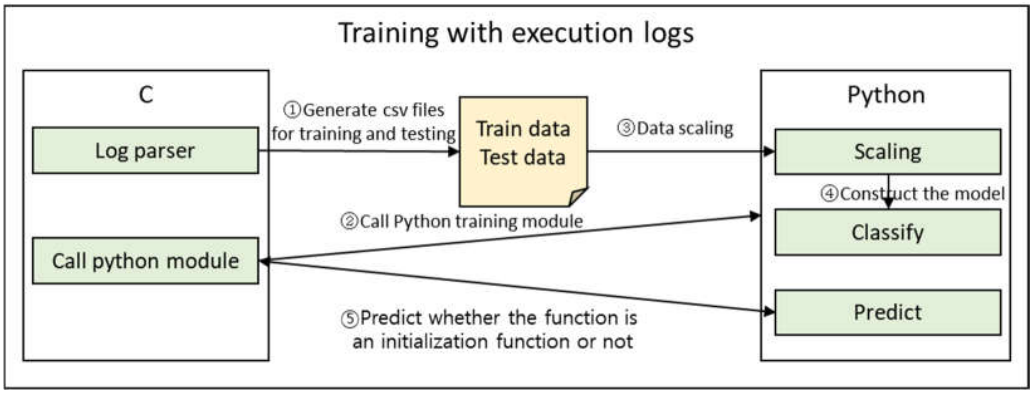

Figure 8. Variable vulnerability detection training module and variable vulnerability detection module.

Log parser: The log parser generates a train.csv file from the collected execution logs. As depicted in Figure 9, the train.csv file contains the following data: the name of the calling function (FuncName), the variable name (VarName), the number of times the variable is called by the function (CallCount), whether the keywords related to the initialization, such as init, set, clear, and reset, are included (KeywordIncluded), the variable's final value after alteration (Value), the variable's initial value (InitValue), and data indicating whether the calling function is the initialization function (InitFunc). FuncName and VarName utilize the calling function ID and the global variable ID, respectively, in the static analysis module of the source code for scaling, rather than text characters. A weight is inputted to KeywordIncluded depending on the keywords that are included—its value is 10 if init, reset, and clear are included, 5 if set is included, and 0 if none are included. Y or N is input into InitFunc; Y is entered for only the initialization function derived during the extraction of the defect detection target in the first training phase, and N is entered for the remaining functions. However, this value changes if an initialization function is added, as training is performed repeatedly.

	A	B	C	D	E	F	G
1	FuncName	VarName	CallCount	KeywordIn	Value	InitValue	InitFunc
2	1	2100	0	0	5	100	N
3	2	2100	0	0	100	100	N

Figure 9. Example of train.csv file.

Calling Python module: This module uses the Python library in the C system. The Python file's name and the training function are used to begin training, and the prediction function is called with the test data as a parameter to determine whether it is an initialization function or not.

Training using RandomForest or linear regression is performed in the Python system. The training process performed in the Python system can be broadly subdivided into scaling, classification, and prediction.

Scaling: The training data used for training shows unique characteristics and distribution—their use as is causes performance degradation (i.e., training may take a long time). Therefore, the scaler, StandardScaler, is used to adjust the data to ensure distribution of the training data within a certain range.

Classification: The random forest classifier and linear regression are used to perform training using the data. In this step, the maximum and minimum values of each variable are extracted.

Prediction: When an execution log is inputted using the final training model, it returns a result indicating whether the function used in the log is an initialization function.

Once training is complete, the initialization function list and each variable's maximum and minimum threshold values are determined. The variable vulnerability detection module uses this information to analyze the execution log, as depicted in Figure 6, and detects potential defects. The type of any detected defect, the location where the vulnerability is detected, and the variable name are output together, as depicted in Figure 10.

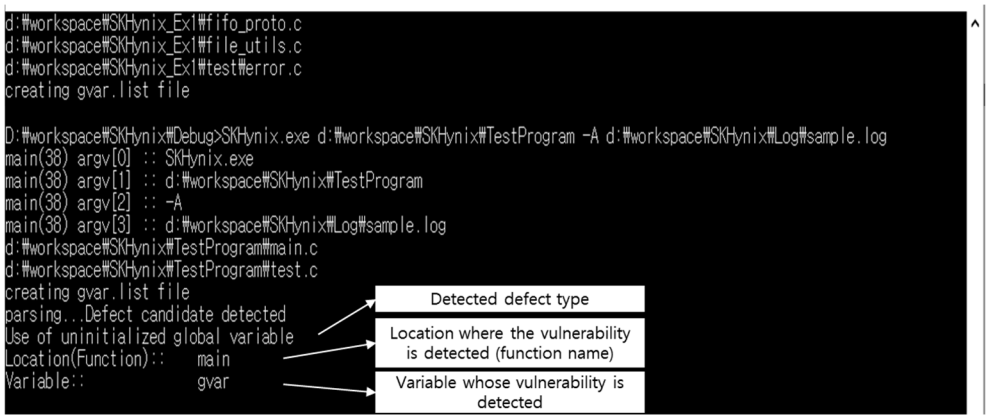

Figure 10. Results of variable vulnerability detection.

4. Experiments

The following research questions were answered based on experimental results.

RQ1. How Effective Is VVDUM in Variable Vulnerability Detection?

We analyzed the effectiveness of VVDUM by comparing it with that of other methods. To this end, defects induced by variables were artificially injected into a well-known Linux program. Then, the performances of the proposed method and baseline methods were evaluated.

RQ2. Is ML Effective for Variable Vulnerability Detection?

If the defect detection rate is high without the use of ML, then the application of ML may be insignificant at the cost of a large overhead. Therefore, the suggested defects were analyzed to evaluate the need for ML. The variable vulnerability detection method without ML was compared to the variable vulnerability detection method using ML in terms of detection performance to analyze the effectiveness of applying ML to variable vulnerability detection.

RQ3. Is VVDUM Applicable in Practical Settings?

VVDUM collects execution logs dynamically and uses them for training. During the operation of actual software, the overhead incurred by collecting execution logs may affect the actual execution. Therefore, the overhead incurred by VVDUM was measured to evaluate its applicability to software in real-world settings.

4.1. Experimental Environment

We conducted experiments in the Linux Unubtu environment. Table 2 presents the applications considered in the experiments. Seven well-known Linux applications were selected. The execution logs used for training were collected by running sample scripts of the Linux manual page, test cases provided by SARD [30], and test cases of the Linux Test Project (LTP) [31] for each application.

Table 2. Target applications considered in the experiments.

Application			Test Case for Collecting Execution Logs Used for Training
Name	**Description**	**Size (LOC)**	
grep	Search utility	5317	Linux manual-sample scripts, SARD
gzip	Compression program	7592	Linux manual-sample scripts, LTP
bc	Shell calculator	8803	Linux manual-sample scripts
wc	Word counter	1126	Linux manual-sample scripts, LTP
sort	Sorter	4850	Linux manual-sample scripts
sed	Script editor	5234	Linux manual-sample scripts
jpegtran	jpeg transformation	31,839	Linux manual-sample scripts

Defects were introduced into applications artificially using Clang-SFI [32] to detect vulnerabilities induced by variables. Table 3 lists the defect types corresponding to variable vulnerabilities related to the proposed method among the defects injectable using Clang-SFI. Local and global variables were extracted from the applications of Table 2, and the defect types listed in Table 3 were injected at each extracted location. In the case of the 'Wrong value assigned to variable' (WVAV) defect type, Clang-SFI toggles the value if it is the Boolean type; otherwise, it injects the 0xFF value as a wrong value. However, for the verification of more diverse cases, the type of defect injected using Clang-SFI is denoted by WVAV_1, the case of injecting the maximum and minimum values for the variable's type is denoted by WVAV_2, and the case of injecting a value selected randomly from the values that the variable can have is denoted by WVAV_3.

Table 3. Defect injection type.

Defect Type	Description	Injection Example	
		Before Defect Injection	**After Defect Injection**
MVAE	Missing variable assignment with an expression	hash ^= s -> elems [i]. index + s -> elems [i]. constraint;	-
MVAV	Missing variable assignment using a value	options = 0	-
MVIV	Missing variable initialization using a value	int options = 0;	int options;
WVAV_1	Wrong value assigned to variable	state_newline = 0	state_newline = 1894; (value returned from Clang-SFI)
WVAV_2			state_newline = INT_MIN; state_newline = INT_MAX;
WVAV_3			state_newline = rand()

Typical tools for vulnerability detection are listed in Table 4. Static analysis-based variants are divided into two types—tools that detect vulnerabilities by comparing source codes in terms of similarity and tools that detect vulnerabilities based on patterns. Since vulnerabilities to be detected by each tool are the same as the CWE vulnerabilities, a relatively recent tool based on ML/deep learning was selected alongside one that does not use ML. In the case of dynamic defect detection methods, we could not find any tool

based on ML. Therefore, MemCheck was selected as the most relevant tool for the defects selected for detection. MemCheck usually detects memory defects, but since defects may occur due to incorrect memory access if the targeted defect is not initialized during the use of the variable, this tool was a viable candidate for comparison.

Table 4. Vulnerability detection tools.

Category		Tool Name	Detectable Defect Type	Application of AI/ML Method	Applied in Experiment
Static	Code-based	VUDDY	CWE vulnerability	X	O
		VulPecker	CWE vulnerability	X	X
		VulDeeLocater	CWE vulnerability	O	X
		VulDeePecker	CWE vulnerability	O	X
		mVulDeePecker	CWE vulnerability	O	X
		SySeVR	CWE vulnerability	O	O
	Pattern-based	cppcheck	Use of uninitialized variable, type conversion error, out-of-bound access to array	X	O
		flawfinder	CWE vulnerability	X	X
		coverity	CWE vulnerability	X	X
		controlflag	CWE vulnerability	O	O
Dynamic		Memcheck	Memory defect	X	O

Four metrics were used to evaluate the effects. True Positive (TP) denotes the number of defects detected as vulnerabilities. True Negative (TN) denotes the number of non-defects not detected as vulnerabilities. False Positive (FP) denotes the number of non-defects detected as vulnerabilities. False Negative (FP) denotes the number of undetected defects.

$$\text{False positive rate} = \frac{\text{FP}}{\text{FP} + \text{TN}} \times 100 \quad (1)$$

$$\text{Accuracy} = \frac{\text{TP} + \text{TN}}{\text{Total number of experiments}} \times 100 \quad (2)$$

$$\text{Precision} = \frac{\text{TP}}{\text{TP} + \text{FP}} \quad (3)$$

$$\text{Variable vulnerability detection rate} = \frac{\text{TP}}{\text{Total number of injected flaws}} \quad (4)$$

4.2. Experimental Results

4.2.1. RQ1. How Effective Is VVDUM in Terms of Variable Vulnerability Detection?

To evaluate the effectiveness of VVDUM, experiments were conducted on VUDDY, SySeVR, CppCheck, and ControlFlag, i.e., both code-based and patterns-based methods, and the results were compared. VUDDY and CppCheck do not use ML or artificial intelligence, whereas SySeVR and ControlFlag do use them.

In the case of ControlFlag, the CPP source code provided by Intel comprised the training dataset. However, neither the small dataset nor the large dataset were effective. Training was performed using the source codes of the experimental target applications, and the results were examined.

The experimental results are presented in Table 5. The FP rate, accuracy, precision, and variable vulnerability detection rate corresponding to each defect type were analyzed—the values for the proposed method are 0%, 89.5%, 100%, and 89.5%, respectively. On the other

hand, the variable vulnerability detection rates of the other methods are observed to lie between 6% and 18.5%, which are significantly lower than that of the proposed method.

Table 5. Comparison of variable vulnerability detection results.

Tools		False Positive Rate (%)	Accuracy (%)	Precision (%)	Variable Vulnerability Detection Rate (%)
Code Similarity Comparison	VUDDY	0	9.17	100	9.17
	SySeVR	22.9	18.5	77.08	18.5
Pattern-Based	CppCheck	36.84	77.92	63.16	15.58
	ControlFlag	0	9.5	100	9.5
VVDUM		0	89.5	100	89.5
VVDUM without ML		0	83.5	100	83.5

In the case of the 'variable value lies outside the permissible range' defect, the variable vulnerability detection rate is the lowest, and the FP rate, accuracy, and precision are low. This is because this defect includes two cases—when the permissible range of the variable value simply deviates from that corresponding to its type and when it deviates from the intended range indicated by the developer.

4.2.2. RQ2. Is ML Effective for Variable Vulnerability Detection?

With regard to the detection of the vulnerability types, the performances of the proposed method and vulnerability detection methods that do not use ML were compared. Figure 11 depicts the results.

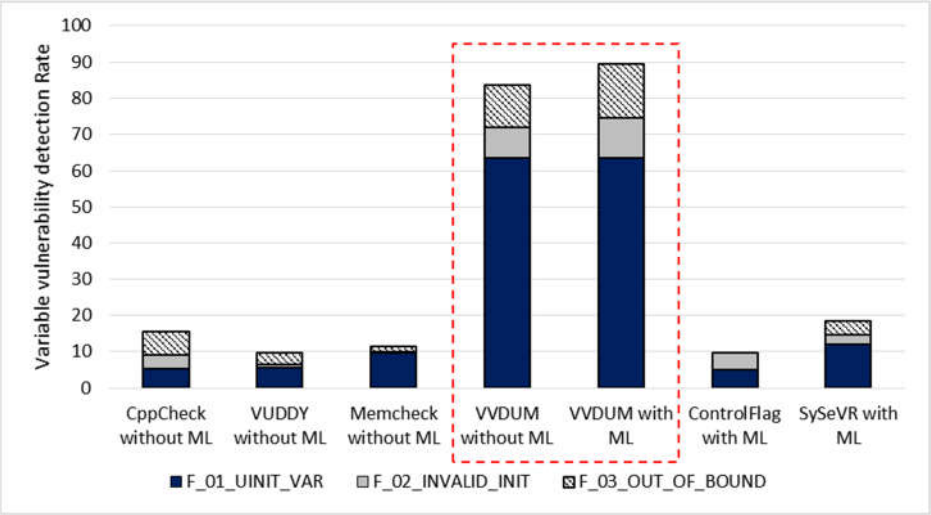

Figure 11. Comparison of results according to machine learning.

The variable vulnerability detection rates of VVDUM and SySeVR, which use ML, are higher than those of CppCheck, VUDDY, and MemCheck, which do not use ML. Even though ControlFlag uses ML, its detection rate is the lowest owing to inadequacies of the training dataset.

The 'F_01_UINIT_VAR—Use of uninitialized variable' defect (MVIV, MVAE, MVAV) was detected for local variables, irrespective of whether ML is applied.

Overall, 83.33% of the injected defects of the type 'F_02_INVALID_INIT—Initialization during the use of variable' (WVAV_1) were detected by VVDUM. However, in the case of CppCheck, VUDDY, and MemCheck, the detection rates for this defect type are 12.5%, 8.33%, and 4.17%, respectively. Both CppCheck and MemCheck determine whether the

variable is properly declared syntactically and whether the value after initialization is within the permissible range according to the variable type. After variable declaration and initialization, if the value does not exceed the permissible range of the variable type, both of tools are classified as normal. Determining whether the value of a variable is correct is to detect a semantic error. However, neither tool detected this. All cases detected as defects by MemCheck were detected during the process of using the initialized value after the injection of the 'initialization during the use of variable' defect.

VVDUM was observed to detect 61.22% of the 'F_03_OUT_OF_BOUND—The variable value is out of the allowable range' defects (WVAV_2, WVAV_3) injected in the experiments. In comparison, CppCheck, VUDDY, and MemCheck detected 10.2%, 12.24%, and 6.12%, respectively, as in the case of the 'initialization during the use of variable' defect. The comparison baselines detected the defect when the variable value is out of the permissible range. However, they failed to detect defects at all when the permissible range of variable values deviates from the range intended by the developer.

Finally, the variable vulnerability detection rates of VVDUM before and after the application of ML were compared. VVDUM detected 83.5% of defects without applying ML, but the defect detection rate improved to 89.5% when ML was applied. In particular, for the F_02_INVALID_VAR and F_03_OUT_OF_BOUND defects, whose detection rates were very low without using ML, the detection rate was increased by 14.28–20.84% after the application of ML, indicating the positive effect of the latter.

We targeted semantic defects such as determining whether a variable value is valid or not, rather than syntax defects. Conventional methods that do not use ML struggled to detect these defects. Our results clearly suggest that the application of ML enhances defect detection for the variable vulnerabilities considered in this paper.

4.2.3. RQ3. Is VVDUM Applicable in Practical Settings?

VVDUM analyzes the source code and execution logs on the host's end instead of the execution environment, and, thus, has no impact on the execution of the software. The overhead incurred by it was measured to be 6.1%, as shown in Figure 12.

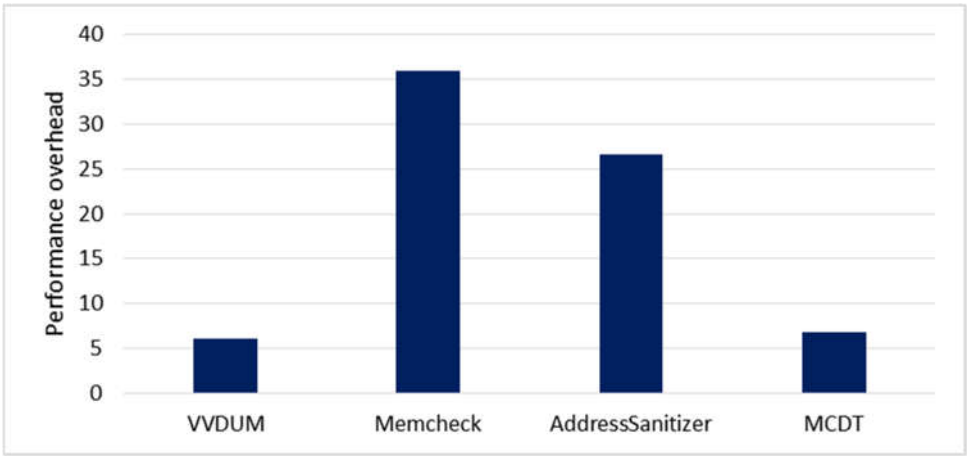

Figure 12. Performance overhead.

There is no clear standard on the permissible threshold for overheads during the execution of software. However, the overhead incurred by dynamic verification methods lies between 6.67% and 36.0%, approximately, and if reference values are used [33,34], the overhead incurred during the collection of execution logs is even lower. In other words, the incurred overhead does not affect execution significantly. Therefore, VVDUM can be applied to software in real-world settings

5. Conclusions

In this paper, we proposed a method that combines static analysis and dynamic verification to detect defects related to the use of variables. The proposed method uses static analysis to extract the information about the target variables for defect detection and detects defects by monitoring the execution of the software for each variable. In this process, ML is used to learn each variable's initial value and its permissible range. Based on this, the validity of the currently used variable value and the order of use is evaluated. This enables the proposed method to detect even 'malfunctions caused by incorrect variable values', which are difficult to detect using conventional methods.

The effectiveness of the proposed method was evaluated by implementing it in a Linux-based automated tool, VVDUM, and applying it to seven well-known Linux applications. VVDUM showed a high defect detection rate for defects that are difficult to detect using conventional variable vulnerability detection methods. By design, it detects defects by evaluating whether each variable's value lies within the permissible range. Based on this analysis, it also detects defects that cannot be detected using conventional methods. Further, VVDUM can be used during the execution of programs, as the execution overhead incurred while collecting execution logs is small.

In our method, since fault detection rate may vary depending on a data set on which training is done or a frequency with which a variable is used, we plan to extend the related research. In addition, we also plan to continue research on the application of machine learning not only to detect a fault but to analyze its cause.

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 & communications Technology Planning & Evaluation (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 unavailable due to privacy.

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

References

1. Common Weakness Enumeration. Available online: https://cwe.mitre.org/ (accessed on 12 December 2022).
2. Kang, H.J.; Aw, K.L.; Lo, D. Detecting False Alarms from Automatic Static Analysis Tools: How Far are We? In Proceedings of the 44th International Conference on Software Engineering (ICSE '22), Pittsburgh, PA, USA, 21–29 May 2022; ACM: New York, NY, USA, 2022. 13p. [CrossRef]
3. Heidari, A.; Navimipour, N.J.; Unal, M. A Secure Intrusion Detection Platform Using Blockchain and Radial Basis Function Neural Networks for Internet of Drones. *IEEE Internet Things J.* **2023**. [CrossRef]
4. Heidari, A.; Jamali, M.A.J. Internet of Things intrusion detection systems: A comprehensive review and future directions. *Clust. Comput.* **2022**, 1–28. [CrossRef]
5. Senanayake, J.; Kalutarage, H.; Al-Kadri, M.O. Android Mobile Malware Detection Using Machine Learning: A Systematic Review. *Electronics* **2021**, *10*, 1606. [CrossRef]
6. Sagar, R.; Jhaveri, R.; Borrego, C. Applications in Security and Evasions in Machine Learning: A Survey. *Electronics* **2020**, *9*, 97. [CrossRef]
7. Ghiasi, M.; Dehghani, M.; Niknam, T.; Kavousi-Fard, A.; Siano, P.; Alhelou, H.H. Cyber-Attack Detection and Cyber-Security Enhancement in Smart DC-Microgrid Based on Blockchain Technology and Hilbert Huang Transform. *IEEE Access* **2021**, *9*, 29429–29440. [CrossRef]
8. Dehghani, M.; Niknam, T.; Ghiasi, M.; Siano, P.; Alhelou, H.H.; Al-Hinai, A. Fourier Singular Values-Based False Data Injection Attack Detection in AC Smart-Grids. *Appl. Sci.* **2021**, *11*, 5706. [CrossRef]
9. Ghiasi, M.; Niknam, T.; Wang, Z.; Mehrandezh, M.; Dehghani, M.; Ghadimi, N. A comprehensive review of cyber-attacks and defense mechanisms for improving security in smart grid energy systems: Past, present and future. *Electr. Power Syst. Res.* **2023**, *215*, 108975. [CrossRef]

10. Dehghani, M.; Niknam, T.; Ghiasi, M.; Bayati, N.; Savaghebi, M. Cyber-Attack Detection in DC Microgrids Based on Deep Machine Learning and Wavelet Singular Values Approach. *Electronics* **2021**, *10*, 1914. [CrossRef]
11. Li, J. Vulnerabilities Mapping based on OWASP-SANS: A Survey for Static Application Security Testing (SAST). *Ann. Emerg. Technol. Comput.* **2020**, *4*, 1–8. [CrossRef]
12. Zaazaa, O.; El Bakkali, H. Dynamic vulnerability detection approaches and tools: State of the Art. In Proceedings of the 2020 Fourth International Conference On Intelligent Computing in Data Sciences (ICDS), Fez, Morocco, 21–23 October 2020; pp. 1–6.
13. Kim, S.; Woo, S.; Lee, H.; Oh, H. Vuddy: A scalable approach for vulnerable code clone discovery. In Proceedings of the 2017 IEEE Symposium on Security and Privacy (SP), San Jose, CA, USA, 22–24 May 2017; pp. 595–614.
14. Zhen, L.; Zou, D.; Xu, S.; Jin, H.; Qi, H.; Hu, J. Vulpecker: An automated vulnerability detection system based on code similarity analysis. In Proceedings of the 32nd Annual Conference on Computer Security Applications, Los Angeles, CA, USA, 5–8 December 2016; pp. 201–213.
15. Cppcheck. Available online: https://github.com/danmar/cppcheck/releases (accessed on 24 April 2022).
16. Oliver, F.; Gurevych, I.; Rittberger, M. FlawFinder: A Modular System for Predicting Quality Flaws in Wikipedia. In Proceedings of the CLEF (Online Working Notes/Labs/Workshop), Rome, Italy, 17–20 September 2012; pp. 1–10.
17. Coverity. Available online: https://scan.coverity.com/ (accessed on 24 April 2022).
18. Kharkar, A.; Moghaddam, R.Z.; Jin, M.; Liu, X.; Shi, X.; Clement, C.; Sundaresan, N. Learning to reduce false positives in analytic bug detectors. In Proceedings of the 44th International Conference on Software Engineering, Pittsburgh, PA, USA, 22–27 May 2022; pp. 1307–1316.
19. Xu, H.; Ren, W.; Liu, Z.; Chen, J.; Zhu, J. Memory Error Detection Based on Dynamic Binary Translation. In Proceedings of the 2020 IEEE 20th International Conference on Communication Technology (ICCT), Nanning, China, 28–31 October 2020; pp. 1059–1064.
20. Park, J.; Choi, B.; Jang, S. Dynamic analysis method for concurrency bugs in multi-process/multi-thread environments. *Int. J. Parallel Program.* **2020**, *48*, 1032–1060. [CrossRef]
21. Vadayath, J.; Eckert, M.; Zeng, K.; Weideman, N.; Menon, G.P.; Fratantonio, Y.; Balzarotti, D.; Doupé, A.; Bao, T.; Wang, R.; et al. Arbiter: Bridging the Static and Dynamic Divide in Vulnerability Discovery on Binary Programs. In Proceedings of the 31st USENIX Security Symposium (USENIX Security 22), Boston, MA, USA, 10–12 August 2022; pp. 413–430.
22. Andreas, S.; Binder, D. Deep-Learning-based Vulnerability Detection in Binary Executables. *arXiv* **2022**, arXiv:2212.01254.
23. Siddhasagar, P.; Nallagonda, H.V.; Prakash, S.; Vigneswaran, R.; Medicherla, R.K.; Rajan, M.A. Smart Contract Fuzzing for Enterprises: The Language Agnostic Way. In Proceedings of the 2022 14th International Conference on COMmunication Systems & NETworkS (COMSNETS), Bengaluru, India, 3–8 January 2022; pp. 1–6.
24. Niranjan, H.; Gottschlich, J. ControlFlag: A self-supervised idiosyncratic pattern detection system for software control structures. In Proceedings of the 5th ACM SIGPLAN International Symposium on Machine Programming, Virtual, 21 June 2021; pp. 32–42.
25. 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]
26. 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.
27. 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]
28. Zou, D.; Wang, S.; Xu, S.; Li, Z.; Jin, H. VulDeePecker: A Deep Learning-Based System for Multiclass Vulnerability Detection. *IEEE Trans. Dependable Secur. Comput.* **2019**, *18*, 2224–2236.
29. Motor Trend Car Road Tests. Available online: https://www.kaggle.com/datasets/ruiromanini/mtcars (accessed on 24 April 2022).
30. SARD Dataset. Available online: https://samate.nist.gov/SARD/ (accessed on 24 April 2022).
31. Linux Test Project. Available online: https://github.com/linux-test-project/ltp (accessed on 24 April 2022).
32. Gabor, U.T.; Siegert, D.F.; Spinczyk, O. High-Accuracy Software Fault Injection in Source Code with Clang. In Proceedings of the 2019 IEEE 24th Pacific Rim International Symposium on Dependable Computing (PRDC), Kyoto, Japan, 1–3 December 2019; pp. 75–84.
33. Memcheck. Available online: https://valgrind.org/docs/manual/mc-manual.html (accessed on 24 April 2022).
34. Park, J.; Choi, B.; Kim, Y. Automated Memory Corruption Detection through Analysis of Static Variables and Dynamic Memory Usage. *Electronics* **2021**, *10*, 2127. [CrossRef]