Hindawi
Security and Communication Networks
Volume 2019, Article ID 8391425, 13 pages
https://doi.org/10.1155/2019/8391425

Research Article

A Buffer Overflow Prediction Approach Based on Software Metrics and Machine Learning

Jiadong Ren,[1,2] Zhangqi Zheng,[1,2] Qian Liu,[1,2] Zhiyao Wei,[1,2] and Huaizhi Yan[3]

[1]*School of Information Science and Engineering, Yanshan University, Qinhuangdao, Hebei, China*
[2]*The Key Laboratory for Computer Virtual Technology and System Integration of Hebei Province, Qinhuangdao City 066004, China*
[3]*Beijing Key Laboratory of Software Security Engineering Technique, Beijing Institute of Technology, South Zhongguancun Street, Haidian District, Beijing 100081, China*

Correspondence should be addressed to Zhangqi Zheng; 451499304@qq.com

Received 11 October 2018; Revised 26 December 2018; Accepted 10 February 2019; Published 3 March 2019

Guest Editor: Jiageng Chen

Buffer overflow vulnerability is the most common and serious type of vulnerability in software today, as network security issues have become increasingly critical. To alleviate the security threat, many vulnerability mining methods based on static and dynamic analysis have been developed. However, the current analysis methods have problems regarding high computational time, low test efficiency, low accuracy, and low versatility. This paper proposed a software buffer overflow vulnerability prediction method by using software metrics and a *decision tree* algorithm. First, the software metrics were extracted from the software source code, and data from the dynamic data stream at the functional level was extracted by a data mining method. Second, a model based on a *decision tree* algorithm was constructed to measure multiple types of buffer overflow vulnerabilities at the functional level. Finally, the experimental results showed that our method ran in less time than *SVM, Bayes, adaboost, and random forest* algorithms and achieved 82.53% and 87.51% accuracy in two different data sets. The method presented in this paper achieved the effect of accurately predicting software buffer overflow vulnerabilities in C/C++ and Java programs.

1. Introduction

With the popularity and rapid development of computer network technology, software security is facing a growing number of threats. Since the first buffer overflow attack occurred in 1988, the buffer overflow vulnerability [1] has been the most common and serious software vulnerability, posing a great danger to the security of software systems. According to the distribution of vulnerability types in the June 2017 Security Vulnerability Report of the National Information Security Vulnerability Database (*CNNVD*), the buffer error category has the largest proportion of vulnerabilities, at approximately 14.08%. Due to the various forms of buffer overflow vulnerabilities, it is challenging to accurately predict buffer overflows. Due to the large amount of software source code and the complex logic level of invocation, these kinds of vulnerabilities are not easily discovered in a timely manner.

Software vulnerability analysis usually requires source code analysis and binary analysis. Although the source code vulnerability analysis can comprehensively consider the execution path according to the rich and complete semantic information in the program source code, it cannot fully reflect the dynamics of the software [2]. Although binary code vulnerability analysis has been studied comprehensively through static or dynamic and manual or automated methods, there are problems such as low coverage and path explosion. Recently, machine learning algorithms have been widely used in software vulnerability prediction. In vulnerability prediction, machine learning techniques such as *logistic regression*, *naive Bayes methods*, *C4.5 decision tree algorithms*, *random forest classifiers,* and *SVMs* [3, 4] are commonly used for prediction. Although it is common to use a machine learning algorithm to predict software vulnerabilities, the research based on software metrics is still immature. For instance, in the area of predicting software vulnerability models based on software metrics, most approaches consist of a static analyses based on software source code. It does not consider the characteristics of the dynamic data flow during software

operation. It is not possible to more accurately reflect the type of vulnerability, its severity, and the distribution of the module.

This paper proposes a method based on software metrics for buffer overflow vulnerability prediction, called the software vulnerability location (*BOVP*) method. Compared with traditional prediction methods, such as *support vector machines* (*SVMs*), *Bayesian methods, adaboost,* and *random forest* (*RF*) algorithms, this new research method is applicable to software on different platforms (C/C++, JAVA) and can also accurately predict software buffer overflow vulnerabilities. The main contents of this paper are as follows. (1) Software metrics were extracted according to the static analysis of software source code, and a data mining method was used to extract data from the dynamic data stream at the functional level. (2) A *decision tree* algorithm was established to measure multiple types of buffer overflow vulnerability models based on the functional level. The algorithm not only affects the accuracy of the experiment but also reduces the measurement dimension and reduces the experimental overhead. (3) The analyses led to the research idea of considering multiple vulnerability types and different functional classifications for the software buffer overflow vulnerability analysis.

The remainder of this paper is organized as follows: Section 2 introduces the related work of the paper. Section 3 introduces the description of buffer overflow vulnerability overview and source code function classification. Section 4 introduces the overview of the *BOVP* method. Section 5 presents data sources, cross-validation, experimental procedures and results. Section 6 describes the advantages of the *BOVP* approach. Finally, the conclusion and future research direction are presented in Section 7.

2. Related Work

At present, the academic community proposes dual solutions to the phenomenon of software buffer overflow vulnerability based on both detection and protection. For example, to avoid buffer overflow, people use the polymorphic canary to detect an overflow of the stack buffer based on computer hardware research [5, 6]. J Duraes et al. [7] proposed an automatic identification method for buffer overflow vulnerability of executable software without source code. S Rawat et al. [8] proposed an evolutionary calculation method for searching for buffer overflow vulnerability; this method combined genetic algorithms and evolutionary strategies to generate string-based input. It is a lightweight intelligent fuzzy tool used to detect buffer overflow vulnerabilities in C code. IA Dudina et al. [9] used static symbolic execution to detect buffer overflow vulnerabilities, described a prototype of an in-process heap buffer overflow detector, and provided an example of the vulnerability discovered by the detector. In terms of protection from buffer overflow vulnerabilities, S Nashimoto et al. [10] proposed injecting buffer overflow attacks and verified the countermeasures for multiple fault injections. The results showed that their attacks could cover the return address stored in the stack and call any malicious function. J Rao et al. [11] proposed a protection technology called a BF Window for performance and resource-sensitive embedded systems. It verified each memory write by speculatively checking the consistency of the data attributes within the extended buffer window. Through experiments, the proposed mechanism could effectively prevent sequential memory writes across the buffer boundary. Although these methods have effectively detected buffer overflow vulnerabilities to a certain extent, the traditional research methods of software buffer overflow vulnerabilities have the problems of large time overhead, path explosion and lack of dynamic software analysis. Therefore, research on using machine learning methods to predict software buffer overflow vulnerabilities has become increasingly widespread.

At present, research on using machine learning algorithms to predict various types of software vulnerabilities [12] is becoming widespread, and the machine learning methods can improve vulnerability coverage and reduce running time in software vulnerability analysis and discovery processes. However, there are problems in that they that cannot more accurately reflect the type of vulnerability, the severity of the vulnerability or the distribution of the module. For these problems, a method based on the "software metrics" to predict various types of software vulnerabilities was proposed [13]. Early software metrics focused on structured program design. By measuring the complexity of the software architecture, researchers described some basic performance metrics of software systems such as functionality, reliability, portability, and maintainability. Software metrics can be divided into three typical representative categories based on information flow, attributes and behaviour. In recent years, software metrics have been studied extensively and have become a vibrant research topic in software engineering.

James Walden et al. [14] compared models based on two different types of features: software metrics and term frequencies (text mining features). They experimented with Drupal, PHPMyAdmin, and Moodle software. The experiments evaluated the performance of the model through hierarchical and cross-validation. The results showed that both of the models for text mining and software metrics had high recall rates. Based on this research, in 2017, Jeffrey Stuckman et al. [15] explored the role of dimension reduction through a series of cross-validation and project prediction experiments. In the case of software metrics, the dimension reduction technique based on a confirmatory factor analysis produced the best F-measure value when performing project prediction. Choudhary G R et al. [16] studied the change metrics in combination with code metrics in software fault prediction to improve the performance of the fault prediction model. Different versions of the Eclipse project and extracted change metrics from GitHub were used for experimental research. In addition to the existing change metrics, several new change metrics were defined and collected from the Eclipse project repository. The experimental results showed that the variation metrics had a positive impact on the performance of the fault prediction model. Thus, a high-performance model could be established through multiple variation metrics. Sanaz Rahimi et al. [17] proposed a vulnerability discovery model (VDMS), which uses the compiler's code base for static analysis, extracts code characteristics such as complexity (circle complexity) and uses code attributes

Table 1: C/C + + language experimental data set specific data.

	GOOD				BAD			Total
	Good	Good Sink	Good Source	Good Auxiliary	Bad	Bad Sink	Bad Source	
Stack-based buffer overflow	5810	2748	312	7149	4716	2052	288	23075
Heap-based buffer overflow	7122	3076	968	9064	6733	2448	628	30039
Buffer overflow	2380	1180	208	3192	2472	1024	144	10600
Integer overflow	4608	4284	576	10872	4716	2052	288	27396
Integer underflow	3348	3234	408	8100	3414	1548	204	20256
Total	23268	14522	2472	38377	22051	9124	1552	111366

as its parameters to predict vulnerabilities. By performing a static analysis on the source code of real software, it was verified that the code attributes have a high level of influence on the accuracy of vulnerability discovery.

In recent years, many studies have predicted software buffer overflow vulnerability by using a machine learning algorithm. With the maturity of software measurement technology, most scholars have adopted software attribute measurement based on machine learning algorithms to predict vulnerabilities. Most of the software metrics for predicting software vulnerability are based on the static analysis of the software source code, but there are dynamic behaviours such as function calls, memory read and write, and memory allocation during the execution process. Therefore, if the software metrics are to accurately predict software vulnerabilities that are persistent and concealed, the dynamic behaviour of the software needs to be considered.

3. Problem Description

Buffer overflow is the most dangerous vulnerability of software. If the buffer overflow vulnerability can be effectively eliminated, the security threat to the software will be reduced. This paper analysed the data flow based on the source code function level and combined the machine learning algorithm to study the software metrics for real software. Metrics need to be selected in software metrics, such as the number of lines containing source code, the average base complexity, all nested functions or methods, and the average number of lines of source code that contain all nested functions or methods (as shown in Table 1).

3.1. Buffer Overflow Vulnerability Overview. A buffer is a contiguous area of memory within a computer that can hold multiple instances of the same data type. The reason for the buffer overflow is that the computer exceeds the capacity of the buffer itself when it fills the buffer with data. The overflow data are overlaid on the legal data (data, pointer of the next instruction, return address of the function, etc.). Buffer overflows can be divided into stack-based buffer overflows, heap-based buffer overflows, and integer overflows. Integer overflows are classified into three categories based on underflow and overflow of unsigned integers, symbolic problems, and truncation problems. We use the Intel processor (register EBP) as an example to introduce stack-based buffer overflows, heap-based buffer overflows, and integer overflows.

Stack-based buffer overflows: in computer science, a stack is an abstract data type that serves as a collection of elements, with two principal operations: push, which adds an element to the collection, and pop, which removes the most recently added element that was not yet removed. A stack is a container for objects that are inserted and removed according to the LIFO principle. The stack frame is the collection of all data in the stack associated with one subprogram call. The stack frame generally includes the return address, the argument variables passed on the stack, local variables, and the saved copies of any registers modified by the subprogram that need to be restored. The EBP is the base pointer for the current stack frame. The ESP is the current stack pointer. Each time the function is called, a new stack frame is generated and stored in a certain space of the stack. The EBP always points to the top of the stack frame (high address), while the ESP points to the next available byte in the stack. The EBP does not change during function execution, and the ESP varies with function execution. A stack-based buffer overflow occurs when a program writes data with a data length that exceeds the space allocated by the stack to the buffer to the memory address in the stack.

Heap-based buffer overflows: The heap is memory that is dynamically allocated while the program is running. The user generally requests memory through malloc, new, etc. and operates the allocated memory through the returned starting address pointer. After using it, it should be released by free, delete, etc. as part of the memory; otherwise it will cause a memory leak. The heap blocks in the same heap are usually contiguous in memory. If data is written to an allocated heap block, the data overflow exceeds the size of the heap block, causing the data overflow to cover the neighbours behind the heap block.

Integer overflows: Similar to other types of overflows, integer overflows cause data to be placed in a storage space smaller than itself. The underflow of unsigned integers is caused by the fact that unsigned integers cannot recognize negative numbers. The symbol problem is caused by the comparison between signed integers, the operation of signed integers, and the comparison of unsigned integers and signed integers. The truncation problem mainly occurs when a high-order integer (for example, 32 bits) is copied to a low-order integer (for example 16 bits).

Buffer overflow attacks consist of three parts: code embedding, overflow attack, and system hijacking. There are four types of buffer overflow attacks: destroy activity records, destroy heap data, change function pointers, and overflow fixed buffers. Buffer overflow is a very common and very dangerous vulnerability that is widespread in various operating systems and applications. The use of buffer overflow vulnerabilities can enable hackers to gain control of a machine or even the highest privileges. The use of buffer overflow attacks can lead to program failure, system shutdown, restarting, and so on. The buffer overflows we studied in this article include stack-based buffer overflows, heap-based buffer overflows, buffer overflows, integer overflows, and integer underflows.

3.2. Source Code Function Classification. Most software metrics that were previously used to predict software vulnerabilities use metrics to predict whether a vulnerability is included. During the execution of the software, there is a lack of analysis of the calling relationships between functions. Among the characteristics of software are functionality, reliability, usability, efficiency, maintainability and portability. Thus, the software implementation process is dynamic. In the vulnerability prediction of software, the calling relationship of the intrinsic functions in the software execution process should be fully considered, especially the data flow changes during the software call. Therefore, this article describes data flow analysis to distinguish features that contain vulnerabilities.

In general, software source code data flow analysis is based on understanding the logic of the code, tracking the flow of data from the beginning of the analysed code to the end. There are three difficulties in the analysis process. (1) The change in data should accumulate with the increase in the path length. (2) Calculation of the value of the branch condition based on the latest data analysis results since the branch condition should accumulate as the path grows, and each additional branch node on the path must be in the path. (3) The number of paths usually grows at a geometric progression as the code size increases. A more efficient approach is to decompose the global data stream analysis into partial data stream analysis.

This paper analyses the functions called in the test case data and uses the partial data flow analysis method to distinguish the functions into different "source" and "accept" functions. This research performs a functional level analysis on the program source code. According to the behaviour of the data flow of each function in the actual execution of the program, this paper uses a data mining method to distinguish the function categories from the data in the data set. First, there are two types (good and bad) depending on whether or not there are vulnerabilities as a whole. Second, this paper analyses the functions using data flow analysis. Functions that do not contain vulnerabilities are classified into four categories: Good - the only code in a good function is a call to each of the secondary good functions. This function does not contain any nonflawed constructs; Good Sink and Good Source - cases that contain data flows using "source" and "sink" functions that are called from each other from the primary good function. Each source or sink function is specific to exactly one secondary good function; Good Auxiliary - where a Good Source is passing safe data to a potentially Bad Sink and where a Bad Source is passing unsafe or potentially unsafe data to a Good Sink. The categories containing vulnerabilities are divided into three groups: Bad: this is a function that contains a flawed construct; Bad Sink and Bad Source: these are cases that contain data flows using "source" and "sink" functions that are called from each other or from a primary bad function. Each source or sink function is specific to either bad function for the test case.

According to whether or not the vulnerability is included in one of the 8 categories, this paper defines the set of categories as $X = \{x_1, x_2 \ldots, x_8\}$, where x_1, x_2,..., x_8 are the number of functions that do not contain vulnerabilities (Good), sinks that do not contain vulnerabilities (Good Sink), source functions that do not contain vulnerabilities (Good Source), auxiliary functions that do not contain vulnerabilities (Good Auxiliary), a main function containing a bug (Bad), a sink that contains the vulnerability's accept function (Bad Sink), a source function containing a vulnerability (Bad Source), and a function that is not yet explicitly defined (Other). In this study, only the first seven categories were studied.

4. Method

4.1. BOVP Method. This paper first extracts the source code of the executable program and then establishes a dynamic data flow analysis model at the software function level. Finally, a vulnerability prediction method based on software metrics is implemented. The process of establishing the *BOVP* {M, A, S} model in this method is shown in Figure 1.

(1) For the source code of the target software, this paper applied the functional level static analysis, and the software attribute measurement method was used to extract all software metric data (M).

(2) According to the characteristics of the software buffer overflow vulnerability, this paper established a metric "A" of the software level buffer overflow vulnerability based on the function level, A = {a1, a2, ..., an}.

(3) For the characteristics of the data flow where the functions are calling each other, the data mining method was used to extract the data regarding the function class, and items in the "Other" function category were removed.

(4) This paper applied the mutual information value for feature selection and incorporated the metric A to measure the impurity of the data partition D for the split criterion. The cost complexity pruning algorithm was applied as the pruning method to determine the buffer overflow vulnerability classification algorithm S (strategy).

4.2. BOVP Prediction

4.2.1. Model Overview. The *CART* (classification and regression tree) model was proposed by Breiman et al. [18] in 1984 and has become a widely used *decision tree* learning method. The *CART* method assumes that the *decision tree* is a binary tree. The input feature space is divided into a finite number of

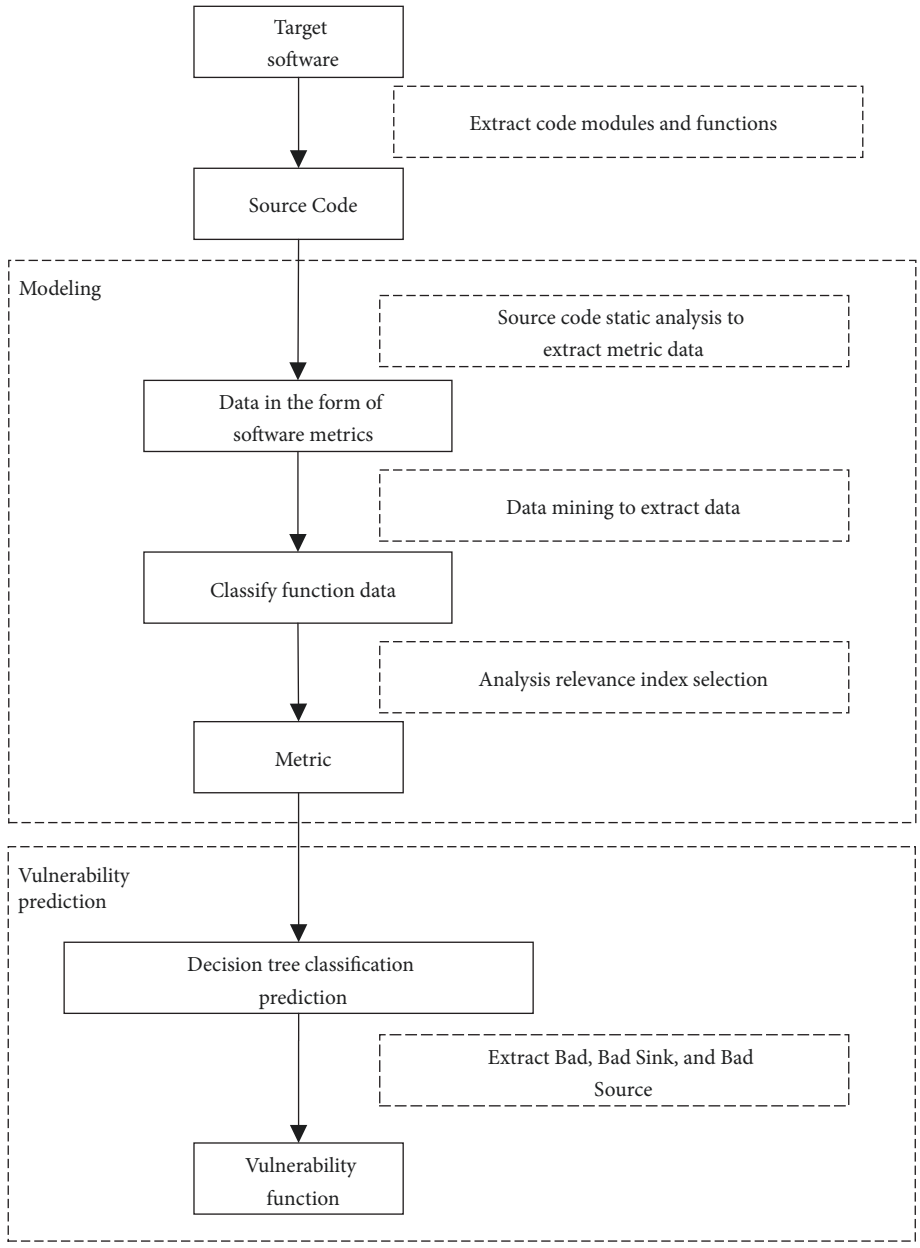

Figure 1: The framework for the *BOVP* method.

cells, and the predicted values are determined by these cells, for instance, for predicting the output values to map to a given condition.

The *CART* model in this article consists of the following three steps:

(1) *CART* generation: generating a *decision tree* based on the training data set.

(2) *CART* pruning: pruning the *decision tree* according to certain constraints, such as the maximum depth of the tree, the minimum number of samples of the leaf nodes, the purity of the nodes, etc.

(3) Optimization of the *decision tree*: generating different CARTs for multiple parameter combinations (maximum depth of the tree (max_depth), minimum sample number of the leaf node (min_samples_leaf), node impurity (min_impurity_split)) and choosing the model with the best generalization ability.

We first chose to use a supervised learning algorithm. The next consideration was the data problem, for example, whether the eigenvalue is a discrete variable or a continuous variable, whether there is a missing value in the eigenvalue vector, and whether there is an abnormal value in the data. Therefore, after analysis, the *CART* tree classification algorithm was selected. *CART* has the following advantages over other classical classification and regression algorithms: (1) less data preparation: no need to standardize data; (2) the ability to process continuous and discrete data; (3) the ability to handle multiclassification problems; (4) the prediction

process of the *CART* model which can be easily explained by Boolean logic and compared to other models such as *Naive Bayes* and *SVMs*.

4.2.2. Training. In the classification prediction part of the *BOVP* method, first, the feature selection operation was performed on the attribute set A. Second, a specific classification algorithm was selected according to the data characteristics. This research adopts the *decision tree* algorithm [19] to establish the classification model based on the characteristics of the integrated software and the characteristics of the classification algorithm.

First, the classification *decision tree* model is a tree structure that classifies instances based on features. The *decision tree* consists of nodes and directed edges. There are two types of nodes: internal nodes and leaf nodes. The internal nodes represent n attributes, including the number of lines of source code, the code path that can be executed, and the circle complexity in A. The leaf nodes represent the type of $X = \{x_1., x_2, ..., x_7\}$. Second, when the *decision tree* is classified, one of the features of an instance is tested from the root node, and the instance is assigned to its child node according to the test result. Each child node corresponds to a value of the feature, so it recursively moves down until the leaf node is reached. Finally, the instance is assigned to the class of the leaf node. The *decision tree* can be converted to a set of if-then rules, or it can be considered a conditional probability distribution defined in the feature and the class space. Compared with the naive Bayes classification method, the *decision tree* has the advantage that the construction process does not require any domain specific knowledge or parameter settings. Therefore, the *decision tree* is more suitable for exploratory knowledge discovery in practical applications. The construction of the *decision tree* algorithm [19] is divided into three parts: feature selection, *decision tree* generation, and *decision tree* pruning, as follows:

(1) Feature selection: the features in this model are a_n ($n <= 22$), an attribute that includes the number of lines in the source code, the code path that can be executed, and the circle complexity. The feature selection is applied to select the feature that has the ability to classify the training data in a way that can improve the efficiency of *decision tree* learning. The feature selection process uses information gain and the information gain ratio or *Gini* indexes.

(2) Generation of *decision tree*: the *Gini* index is used as the criterion of feature selection to measure the impurity of data partition D such that the local optimal feature is continuously selected and the training set is divided into subsets that can be feasible.

(3) *Decision tree* pruning: the cost complexity pruning algorithm is used to prune the tree; that is, the tree complexity is regarded as a function of the number of leaf nodes and the error rate of the tree, specifically from the bottom of the tree. For each internal node N, the cost complexity of the subtree of N and the cost complexity of the subtree of N after the pruning of the subtree are calculated, and the subtree is clipped; otherwise, the subtree is retained.

Combining the attribute set $A = \{a_1, a_2, ..., a_{22}\}$, we can set the training data partition to D, and construct the *decision tree* model through three steps of feature selection, generation *decision tree* and pruning. The basic strategy of the algorithm is to call the algorithm with three parameters D, *attribute_list* and *Attribute_selection_method*. D is a data partition which is a collection of training tuples and corresponding categories of labels. *attribute_list* is a list describing the tuple attributes. *Attribute_selection_method* specifies the heuristic process that is used to select the "best" attributes by class in tuple.

In Algorithm 1, Lines (1) to (6) specify the *Gini* minimization criterion used to obtain the feature attributes and the partition points of the split selection. When the feature selection is started, a tree is constructed step by step. Lines (7) to (8) describe the pruning process of the tree according to three indicators (γ, α, β). Once the stopping condition is satisfied, data set D is split into two subtrees, D1 and D2, which are expressed in Lines (9) to (10). Lines (11) to (12) state that if the sample number of the nodes is less than β, the node will be dropped. Lines (13) to (17) show that if the node does not satisfy all of the conditions, it will be converted into a leaf node whose output class is the highest of the classes on the node. Finally, the process of prediction is conducted by using the TreeModel in Line (19). This method is highly complex, especially when searching for the segmentation point, as it is necessary to traverse all the possible values of all the current features. If there are F features each having N values, and the generated *decision tree* has F or S internal nodes, then the time complexity of the algorithm is O(F∗N∗S).

The *decision tree* in the algorithm uses the *Gini* coefficient as the splitting point of the selection feature. That is, the training data set D is divided into two parts D_1 and D_2 according to whether feature A takes a certain value a. Then, under the condition of feature A, the *Gini* index of set D is defined as

$$Gini(D, A) = \frac{|D_1|}{D} Gini(D_1) + \frac{|D_2|}{D} Gini(D_2) \quad (1)$$

The *Gini* index *Gini*(D) represents the uncertainty of set D and the *Gini* index *Gini*(D, A) represents the uncertainty of set D after A=a partitioning. The larger the *Gini* index is, the greater the uncertainty of the sample is. Compared to other classical classification and regression algorithms, *decision trees* have the advantage of generating easy-to-understand rules, requiring a relatively small number of computations. *Decision trees* are capable of processing continuous categorical fields and have the ability to clearly display more important fields. Therefore, using the existing data set to train the *decision tree* model, the model can be trained to predict whether the program has any of the 7 buffer vulnerabilities by using the metrics, such as the number of rows containing the source code, the number of times called, the code path that can be executed, and the loop complexity.

The specific method of generating the subtree sequences T0, T1,..., Tn in pruning is by pruning the branch in Ti that has the smallest increase in error in the training data set from T0 to obtain Ti+1. For example, when a tree T is pruned at node t, the increase in error is R(t)−R(Tt), where R(t) is the error of node t after the subtree of node t is pruned. R(Tt) is the error of the subtree Tt when the subtree of node t is not pruned. However, the number of leaves in T is

input: D={(X_1, Y_1),(X_2,y_2),...,(X_n, y_n)}, X = {$X_1, X_2, X_3, \ldots, X_n$} represents a feature property set, Y = {$y_1, y_2, y_3, \ldots, y_n$} represents the predicted attribute set.
output: *CART* model sets
(1) for x_i in X
(2) for T_j in x_i
(3) search min(Gini);
(4) end for
(5) end for
(6) Build Trees TreeModel from Tj and xi;
(7) if H(D) ≤ γ && the current depth < α && |D| ≥ β // |D| sample number of D
(8) Divide D to D1 and D2;
(9) DecisionTreeClassifier (D1, α, γ, β);
(10) DecisionTreeClassifier (D2, α, γ, β);
(11) else if |D| < β
(12) drop D
(13) else
(14) D ⟶ leaf_Node; // convert to leaf node
(15) predictive class = the most number of classes; // average of samples
(16) break;
(17) end else
(18) return TreeModel
(19) Prediction on TMS.

Algorithm 1: *BOVP* based on *Gini* indexes.

reduced by L(Tt)−1 after pruning, so the complexity of T is reduced. Therefore, considering the complexity factor of the tree, the error increase rate after the tree branches are pruned is determined by

$$\alpha = \frac{R(t) - R(T_t)}{L(T_t) - 1} \tag{2}$$

where R(t) represents the error of node t after the subtree of node t is pruned, R(t)=r(t)∗p(t), r(t) is the error rate of node t, and p(t) is the ratio of the number of samples on node t to the number of samples in the training set. R(Tt) represents the error of the subtree Tt when the subtree of node t is not pruned, that is, the sum of the errors of all the leaf nodes on subtree Tt. Ti+1 is the pruning tree with the smallest alpha value in Ti.

4.2.3. Prediction. The input part of the prediction section is the buffer overflow vulnerability data set D={(x1, y1), (x2, y2)...(xn, yn)}. Vulnerability data include stack buffer overflow, heap buffer overflow, buffer underwriting ("buffer overflow"), integer overflow, and integer underflow.

When constructing the *decision tree* in the prediction process, the feature-selected metrics A = {$a_1, a_2, \ldots, a_{16}$} were used as internal nodes to represent an attribute test, and the *Gini* index was used to select the partition attribute. Each branch represents an output of the test, and the leaf nodes correspond to the function class X = {$x_1, x_2, \ldots, x_7$}. When constructing a *decision tree*, this paper uses attribute selection metrics to select the attributes that best divide the tuple into different classes.

The prediction result output is based on the class predicted by the *decision tree* algorithm belonging to the X = {$x_1, x_2, \ldots, x_7$} category. Further analysis can be performed on the function containing the vulnerability based on the prediction results. The prediction part used the accuracy rate, recall rate and F_1 evaluation index to measure the effect of the *decision tree* algorithm. Accuracy is used to measure the ratio of the number of correctly classified samples to the total number of samples in the test data set. It reflects the ability of the *decision tree* classifier to examine the entire sample. The recall rate is used to measure the proportion of all that should be retrieved correctly, which reflects the proportion of positive data that is correctly determined by the *decision tree* algorithm to the total positive data. The F1 value is used to measure the accuracy of the recall and the recall average to measure the ideal degree of *decision tree* algorithm classification.

5. Experiment

In this paper, the relationship of function nesting, iteration and looping in the process of software execution, and the dynamic behaviour and operations of the functions were considered comprehensively. The vulnerability classification and prediction were carried out according to the unified pattern of software metrics. This paper carried out the following steps for the program: (1) This article extracted code that contained only one type of defect based on the buffer overflow vulnerability code fragment and that did not contain code of other unrelated defect types. (2) Then, this paper statically analysed the size, complexity, and coupling of the source code for each code segment that could be generated. At the same time, software was used to extract the metrics from the program, and 58 functional level metrics were extracted. (3) Due to the diversity of the types of buffer

overflow vulnerabilities involved in the program, there is data imbalance in the extracted static code indicators. Therefore, it was necessary to perform data cleaning on the initially extracted data to reexamine and verify the data and delete duplicate information. Twenty-two indicators were selected for research. (4) The method of feature selection based on mutual information values was used to select the software metric attribute A. (5) According to the 4:1 ratio between the training set and test set, the *Gini* coefficient was selected as the splitting point for feature selection, and the *CART decision tree* was constructed by the K-times crossover test.

5.1. Data Set Introduction. Using the Juliet data set of the NIST library (https://samate.nist.gov/SRD/view.php), this paper selects 163,737 new types of buffer overflow vulnerabilities in an actual program extraction. The information contained in the data set is made up of the National Security Agency (NSA) Reliable Software (CAS). All the programs selected in this article can be compiled and run on Microsoft Windows using Microsoft Visual Studio. This article extracts software metrics based on the source code of the Juliet data set cross-project test case. A buffer overflow occurs when a write stream flows through the buffer and the function return address is modified during program execution. In this study, source code written in C and Java were used for experiments.

The C/C++ language experimental data set had 111,366 stack-based buffer overflows, heap-based buffer overflows, buffer overflows, integer overflows, and integer underflows. A total of 52,371 buffer overflows were collected from the Java language experimental data set. The summaries of the data sets are shown in Tables 1 and 2.

5.2. Metrics Indexes. This paper extracted a large number of real software attributes through analysis, from software source code. Twenty-two metrics are listed in Table 3. Because some attributes are irrelevant, feature selection based on mutual information is used to eliminate irrelevant metrics. Finally, the accuracy recall and accuracy indicators are used to measure the predicted results.

Software metric indexes provide a quantitative standard to evaluate code quality. It not only visually reflects the quality of software products but also provides a numerical basis for software performance evaluation, productivity model building, and cost performance estimation. According to the development of software from structure, modularization to object-oriented design, metrics can be divided into traditional software metrics and object-oriented software metrics. According to traditional software measurement indexes, the attribute set of the software is first defined as $A = \{a_1, a_2, \ldots, a_n\}$. First, *Understand*, a code analysis tool, was used to extract the software data based on 58 metrics. Second, because the software source code is written in different languages and the coding conventions are different, it is necessary to perform data cleaning on the initially extracted data to reexamine and verify the data, delete duplicate information and correct errors. To ensure data consistency, we need to filter data that does not meet certain requirements. For example, the index of the extracted data contains a large number of zero values and attribute values for the same numerical metrics. Finally, 22 indicators remained for study, so n = 22. The specific indicators are shown in Table 3.

Feature selection based on these 22 metrics can not only improve the efficiency of classifier training and application by reducing the effective vocabulary space but also remove noise features and improve classification accuracy. In this paper, the mutual information method is adopted to select features based on the correlation of attributes.

Mutual information indicates whether two variables X and Y have a relationship [20, 21], defined as follows: let the joint distribution of two random variables (X, Y) be p(x, y), the marginal distributions are p(x) and p(y), the mutual information I(X;Y) is the relative entropy of the joint distribution p(x, y) and the product distribution p(x)p(y), as shown in Equation (3) .

$$I(X;Y) = \sum_{x \in X} \sum_{y \in Y} p(x,y) \log \frac{p(x,y)}{p(x)\,p(y)} \tag{3}$$

The mutual information of feature items and categories reflects the degree of correlation between feature items and categories. When taking mutual information as the evaluation value for feature extraction, the largest number of features of mutual information will be selected.

5.3. Feature Selection. Due to the limited number of samples, the design of a classifier with a large number of features is computationally expensive, and the classification performance is poor. Therefore, feature selection was adopted for the metrics set A [21]; that is, a sample of the high-dimensional space was converted into a low-dimensional space by means of mapping or transformation and dimensionality reduction was achieved. Then, redundant and irrelevant features were deleted by feature selection to achieve further dimensionality reduction. In this experiment, the mutual information value was used to measure the correlation for feature selection. If there was no correlation, the mutual information value was zero. Conversely, if the mutual information value is larger, the correlation is greater. When the feature was selected, the mutual information value was calculated for both the C/C++ and Java data sets. The results are shown in Table 4.

The mutual information values of the two data sets are synthesized to select a_n, n<=22, and finally, 16 metrics are obtained after mutual information calculation. Define the set of attributes as $A = \{a_1, a_2, \ldots, a_{16}\}$. Among them, $a_1, a_2, \ldots, a_{16}$ is *CountInput, CountLine, CountLineCode, CountLineCode Decl, CountLineCode Exe, CountLine Comment, CountOutput, CountPath, CountSemicolon, CountStmt, CountStmtDecl, CountStmtExe, Cyclomatic Modified, Cyclomatic Strict, Ratio CommentCode*.

Although the global results before and after the feature selection seem similar, we achieve the same effect with fewer features by applying feature selection. The obtained data features and the 16 indicators adopted after the feature selection could achieve high-efficiency cross-project software metrics, thereby reducing the number of indicators, dimensions and amount of overfitting. The process also

Table 2: Java language experimental data set specific data.

	GOOD				BAD			Total
	Good	Good Sink	Good Source	Good Auxiliary	Bad	Bad Sink	Bad Source	
Buffer overflow	8073	9522	828	21114	7866	4554	414	52371

Table 3: Metrics.

	Name	Description
1	CountInput	Number of calls. Calls by the same method are counted only once, and calls by fields are not counted.
2	CountLine	The number of lines of code.
3	CountLineCode	The number of lines containing the code.
4	CountLineCodeDecl	The number of lines of the name class, the method name line is also recorded in this number.
5	CountLineCodeExe	The number of lines of pure executing class code.
6	CountLineComment	Annotation class code line number.
7	CountOutput	The number of calls to other methods, multiple calls to the same method are counted as one call. Return statement counts a call.
8	CountPath	Code paths that can be executed are related to cyclomatic complexity.
9	CountPathLog	log 10, truncated to an integer value, of the metric CountPath.
10	CountSemicolon	The number of semicolons.
11	CountStmt	The number of statements, even if multiple sentences are written on one line, is counted multiple times.
12	CountStmtDecl	Defines the number of class statements, including the method declaration line.
13	CountStmtExe	The number of class statements executed.
14	Cyclomatic	Circle complexity (standard calculation method).
15	CyclomaticModified	Circle complexity (the second calculation method).
16	CyclomaticStrict	Circle complexity (the third calculation method).
17	Essential	Basic complexity (standard calculation method).
18	Knots	Measure overlapping jumps.
19	MaxEssentialKnots	The maximum node after the structured programming structure has been deleted.
20	MaxNesting	Maximum nesting level, relating to cyclomatic complexity.
21	MinEssentialKnots	The minimum node after the structured programming structure has been deleted.
22	RatioCommentToCode	Code comment rate.

enhanced the understanding of the relationship between the indicators' characteristics and the indicator feature values, thus increasing the ability of the model to generalize

5.4. Experimental Results. In the classification algorithm, there is often a phenomenon in which the model performs well for the training data but performs poorly for data outside of the training data set. In these cases, cross-validation can be used to evaluate the predictive performance of the model, especially the performance with new data, which can reduce overfitting to some extent. There are three general cross-validation methods: Hold-Out Method, K-fold Cross-Validation, and Leave-One-Out Cross-Validation. In this experiment, the K-fold cross-validation method was selected. K-fold cross-validation randomly divides the training set into k, k-1 for training, and the remaining data for testing, repeating the process k-times, and thus obtaining k models to evaluate the performance of the model. In this experiment, we validated the model into two parts: 80% of data set D was used as training data, and 20% was used as test data.

First, we group the data D into a training set to train the model and a test set to test the predictive performance of the model. This process is repeated until all data are used. Next, the K value is chosen to be 10, which is so that the data set is divided into 10 subsamples, 9 of which are used as training data and one is used as test data. Finally, the sensitivity of the model performance to data partitioning is reduced by averaging the results of 10 groups of results [22].

5.4.1. Results in C/C++ Programs. In this experiment, the program written in C/C++ language was used to conduct experiments, and the *decision tree* algorithm and other machine learning algorithms were compared with respect to accuracy and running time. The results are shown in Table 5.

Experiments showed that the *decision tree* algorithm achieved the best results regardless of the accuracy or running time, and the accuracy in the experiments reached 82.55%. Second, the data feature selection was verified by comparing the 22 metrics without feature selection and the 16 metrics after feature selection, as shown in Table 6.

Table 4: Mutual information value calculation results.

Metrics	C/C++ dataset		Java dataset	
	Mutual information value	Sort	Mutual information value	Sort
CountInput	0.1892	11	0.6375	3
CountLine	0.5584	1	0.7996	1
CountLineCode	0.4902	2	0.6446	2
CountLineCodeDecl	0.4327	5	0.4373	11
CountLineCodeExe	0.3893	9	0.5	7
CountLineComment	0.3980	8	0.499	8
CountOutput	0.1625	12	0.3477	13
CountPath	Nan	13	0.3959	12
CountPathLog	Nan	13	0.2638	17
CountSemicolon	0.4307	7	0.5484	5
CountStmt	0.4465	4	0.5879	4
CountStmtDecl	0.4313	6	0.439	10
CountStmtExe	0.3434	10	0.5123	6
Cyclomatic	Nan	13	0.3122	14
CyclomaticModified	Nan	13	0.3122	15
CyclomaticStrict	Nan	13	0.2968	16
Essential	Nan	13	0.0099	20
Knots	Nan	13	0.2508	18
MaxEssentialKnots	Nan	13	0.0099	21
MaxNesting	Nan	13	0.2236	19
MinEssentialKnots	Nan	13	0.0099	22
RatioCommentToCode	0.4603	3	0.4959	9

Table 5: C/C++ language dataset machine algorithm results.

	SVM	Bayes	Adaboost	Random forest	Decision tree
Accuracy (%)	82.06	37.69	35.47	82.54	82.55
Recall rate (%)	67.8	30.23	34.25	68.65	68.68
Precision rate (%)	71.06	31.46	33.75	73.59	73.62
Running time (s)	576.32	19.98	19.02	17.94	17.03

It was proved by experiments that the running time of the prediction buffer overflow vulnerability algorithm when using the $A = \{a_1, a_2, \ldots, a_{16}\}$ metrics after feature selection is reduced from the previous 17.03 s to 15.94 s, but the accuracy has hardly changed, so feature selection not only helps to reduce the running time but also yields higher accuracy and improves the efficiency of the vulnerability prediction. In the final experimental results, the functions predicted by Bad, Bad Sink and Bad Source are the probabilities of a high level of buffer overflow vulnerability. The three types of data can be extracted according to the function name for further analysis.

5.4.2. Results in Java Programs. The experimental results of the data set extracted by the program written in the Java language are shown in Tables 7 and 8.

In this experiment, we divided the buffer overflow vulnerability into eight categories. When there is vulnerability in real software, it does not necessarily contain all types of buffer vulnerabilities. There may be only one or several of them, and when we collect data in real software data, the data that may be vulnerable is much less than the data without vulnerability and may even reach a ratio of 1:10000. In the Java data set, because the experiment is a data set extracted from real software, the data volume of Good Source and Bad Sink is very small, which results in extremely unbalanced data. It also leads to the accuracy and recall rate of both types of data being 0. The accuracy and recall rate of other basic balanced vulnerability data have better results. To maintain the distribution of the original data, we need to improve the accuracy and recall rate of other basic balanced vulnerability data. To improve the accuracy of the results, data sampling is not used to balance the data, which then demonstrates the validity of the SVL model used to predict most types of buffer overflow vulnerabilities in real software. At the same time, the experiment proves that using the $A = \{a1, a2, \ldots, a16\}$ metrics to predict the buffer overflow vulnerability after feature selection reduces the running time and ensures high accuracy.

Table 6: *Decision tree* algorithm-specific operation results.

	Before feature selection			After feature selection		
	Precision	Recall	F1	Precision	Recall	F1
Good	88.09	83.56	85.76	88.03	83.47	85.69
Good Sink	61.27	52.91	56.78	61.27	52.91	56.78
Good Source	53.13	25.37	34.34	53.13	25.37	34.34
Good Auxiliary	77.16	96.34	85.69	77.16	96.34	85.69
Bad	77.08	49.23	60.09	77.08	49.23	60.09
Bad Sink	67.94	77.47	72.4	67.94	77.47	72.39
Bad Source	90.71	96	93.23	90.66	95.87	93.12
Average	73.63	68.7	69.76	73.61	68.67	71.05
Accuracy		82.55			82.53	
Time		17.03			15.94	

Table 7: Accuracy and time of Java dataset.

	SVM	Bayes	Adaboost	Random forest	Decision tree
Accuracy (%)	87.16	57.95	49.40	87.41	87.44
Recall rate (%)	59.56	48.25	34.98	59.64	59.64
Precision rate (%)	58.74	44.28	33.11	58.91	58.93
Running time (s)	104.32	9.98	8.99	12.94	11.99

Table 8: *Decision tree* algorithm-specific operation results.

	Before feature selection (%)			After feature selection (%)		
	Precision	Recall	F1	Precision	Recall	F1
Good	95.85	93.04	94.42	96.11	93.04	94.55
Good Sink	47.15	53.1	49.95	47.15	53.1	49.95
Good Source	0	0	0	0	0	0
Good Auxiliary	98.56	98.13	98.35	98.56	98.13	98.34
Bad	77.27	73.99	75.6	77.27	73.99	75.6
Bad Sink	0	0	0	0	0	0
Bad Source	95.74	99.51	96.54	93.75	99.6	96.59
Average	59.22	59.69	59.27	58.98	59.69	59.33
Accuracy		87.44			87.51	
Time		11.99			7.59	

6. Discussion

The software vulnerability analysis method proposed in this paper has good capability and has been applied and tested using real code and reliable software. Furthermore, the experiments verified the effectiveness of the *BOVP* model and provided guidance for its effective use. The model minimized the probability of misclassification while finding more vulnerabilities. This method has the following advantages: (1) Given the relative advantages of static and dynamic analysis of software vulnerabilities, the static metrics for source code are extracted by static analysis, and the dynamic metrics at the functional level are analysed according to the data flow of the function. The model can ensure the accuracy of vulnerability prediction and take into account the change in data flow between functions in running programs. This is a new vulnerability prediction method based on data flow analysis in a running program. (2) In this paper, the data set contained two different languages, namely, C/C++ and Java, and thus fully validated the validity of the *BOVP* model and proved that the model does not depend on a specific language type. (3) The model can be applied to vulnerabilities caused by multiple types of buffer overflows. The most common and most difficult software security vulnerability is buffer overflow vulnerability. The data set contains multiple types of buffer overflow vulnerabilities, providing valuable data for analysing different types of buffer overflows and offering a basis for future research. (4) To some extent, this research saves investment costs, time and human resources for software development. Applying feature selection without affecting the prediction results reduces the dimension and the experimental overhead, and it greatly improves the practicality of the software vulnerability prediction study. This paper provides a new way for software metrics to study

data collection in software vulnerability prediction. However, this method is only suitable for source code, not for nonopen source software.

7. Conclusion

The *BOVP* model proposed in this paper preprocesses the program source code and then employs the method of dynamic data stream analysis on the software functional level. By studying the characteristics of practical software code and different types of buffer overflow vulnerability features, the *decision tree* algorithm was developed through feature selection and the *Gini* index. Finally, a vulnerability prediction method based on software metrics was constructed. The capability of the *BOVP* model is verified by experiments using program source code written in different languages, and the prediction results are more accurate than other methods. The time taken for the data set collected using C/C++ was less, and the accuracy rate was 82.53%. The accuracy rate of the data set using Java was also high, reaching 87.51%, which fully demonstrated that this model is robust in predicting buffer overflow vulnerability in different types of languages.

The current buffer overflow vulnerability is very complicated and very common. Although it is divided into 8 categories, it does not contain all of the buffer overflow vulnerabilities. Therefore, the classification of buffer overflow vulnerabilities should be more comprehensive in the future. There are still some shortcomings in the method. For example, there is no corresponding improvement in the imbalance between Good Source and Bad Sink in the Java data set, and the methods of this research are mainly for open source software. It is believed that there is no detailed analysis of nonopen source software. In the future, we plan to solve the problem of unbalanced data, make up for the shortcomings of this model, and study the binary buffer overflow prediction model combined with dynamic symbolic execution technology to find increasingly complex types of buffer overflow vulnerabilities and also forecast vulnerabilities for nonopen source software.

Data Availability

The original data (Juliet Test Suite for C/C++ and Juliet Test Suite for Java) used to support the findings of this study can be download publicly from the website (https://samate.nist.gov/SRD/testsuite.php). The thirteenth page of the articles has footnotes giving the link address.

Conflicts of Interest

The authors declare that they have no conflicts of interest.

Acknowledgments

This work is supported by the National Key R&D Program of China under Grant no. 2016YFB0800700, the National Natural Science Foundation of China under Grants nos. 61802332, 61807028, 61772449, 61772451, 61572420, and 61472341, and the Natural Science Foundation of Hebei Province China under Grant no. F2016203330.

References

[1] C. Cowan, P. Wagle, C. Pu et al., "Buffer overflows: attacks and defenses for the vulnerability of the decade," IEEE, 2003.

[2] S. Wu, *Software Vulnerability Analysis Technology*, Science Press, 2014.

[3] Y. Shin and L. Williams, "Is complexity really the enemy of software security?" in *Proceedings of the ACM Workshop on Quality of Protection*, pp. 47–50, ACM, 2008.

[4] I. Chowdhury and M. Zulkernine, "Using complexity, coupling, and cohesion metrics as early indicators of vulnerabilities," *Journal of Systems Architecture*, vol. 57, no. 3, pp. 294–313, 2011.

[5] M. Frieb, A. Stegmeier, J. Mische et al., "Lightweight hardware synchronization for avoiding buffer overflows in network-on-chips," in *Proceedings of the International Conference on Architecture of Computing Systems*, pp. 112–126, Springer, Cham, Switzerland, 2018.

[6] Z. Wang, X. Ding, C. Pang, J. Guo, J. Zhu, and B. Mao, "To detect stack buffer overflow with polymorphic canaries," in *Proceedings of the 2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pp. 243–254, IEEE, June 2018.

[7] J. Durães and H. Madeira, "A methodology for the automated identification of buffer overflow vulnerabilities in executable software without source-code," in *Dependable Computing*, pp. 20–34, Springer, Berlin, Germany, 2005.

[8] S. Rawat and L. Mounier, "An evolutionary computing approach for hunting buffer overflow vulnerabilities: a case of aiming in dim light," in *Proceedings of the 6th European Conference on Computer Network Defense, EC2ND '10*, pp. 37–45, IEEE, October 2010.

[9] I. A. Dudina and A. A. Belevantsev, "Using static symbolic execution to detect buffer overflows," *Programming and Computer Software*, vol. 43, no. 5, pp. 277–288, 2017.

[10] S. Nashimoto, N. Homma, Y. I. Hayashi et al., "Buffer overflow attack with multiple fault injection and a proven countermeasure," *Journal of Cryptographic Engineering*, vol. 7, no. 1, pp. 1–12, 2016.

[11] J. Rao, Z. He, S. Xu, K. Dai, and X. Zou, "BFWindow: speculatively checking data property consistency against buffer overflow attacks," *IEICE Transaction on Information and Systems*, vol. E99D, no. 8, pp. 2002–2009, 2016.

[12] M. Bozorgi, *A machine learning framework for classifying vulnerabilities and predicting exploitability [Ph.D. thesis]*, Dissertations and Theses - Gradworks, 2009.

[13] X. Xu, R. Zhao, and L. Yan, "Research and progress in software metrics," *Journal of Information Engineering University*, vol. 15, no. 5, pp. 622–627, 2014.

[14] J. Walden, J. Stuckman, and R. Scandariato, "Predicting vulnerable components: software metrics vs text mining," in *Proceedings of the International Symposium on Software Reliability Engineering*, pp. 23–33, IEEE, 2014.

[15] J. Stuckman, J. Walden, and R. Scandariato, "The effect of dimensionality reduction on software vulnerability prediction models," *IEEE Transactions on Reliability*, vol. 99, no. 1, pp. 1–21, 2017.

[16] G. R. Choudhary, S. Kumar, K. Kumar, A. Mishra, and C. Catal, "Empirical analysis of change metrics for software fault

prediction," *Computers and Electrical Engineering*, vol. 67, pp. 15–24, 2018.

[17] S. Rahimi and M. Zargham, "Vulnerability scrying method for software vulnerability discovery prediction without a vulnerability database," *IEEE Transactions on Reliability*, vol. 62, no. 2, pp. 395–407, 2013.

[18] L. Breiman, J. H. Friedman, R. Olshen et al., "Classification and regression trees," *Biometrics*, vol. 40, no. 3, article 356, 1984.

[19] J. Han and M. Kamber, *Data Mining: Concepts and Techniques, Morgan Kaufmann*, Morgan Kaufmann, 2001.

[20] H. Peng, F. Long, and C. Ding, "Feature selection based on mutual information: criteria of max-dependency, max-relevance, and min-redundancy," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 27, no. 8, pp. 1226–1238, 2005.

[21] P. Viola and W. M. Wells III, "Alignment by maximization of mutual information," IEEE Computer Society, 1995.

[22] P. K. Shamal, K. Rahamathulla, and A. Akbar, "A study on software vulnerability prediction model," in *Proceedings of the 2nd IEEE International Conference on Wireless Communications, Signal Processing and Networking, WiSPNET '17*, pp. 703–706, 2017.