Contents lists available at ScienceDirect

The Journal of Systems & Software

journal homepage: www.elsevier.com/locate/jss

The impact factors on the performance of machine learning-based vulnerability detection: A comparative study

Wei Zheng [a,e,f], Jialiang Gao [a], Xiaoxue Wu [b,*], Fengyu Liu [c], Yuxing Xun [a], Guoliang Liu [a], Xiang Chen [d]

[a] *School of Software, Northwestern Polytechnical University, Xi'An, China*
[b] *School of Cyberspace Security, Northwestern Polytechnical University, Xi'An, China*
[c] *School of Computer Science, Northwestern Polytechnical University, Xi'An, China*
[d] *School of Information Science and Technology, Nantong University, Nantong, China*
[e] *Key Laboratory of Advanced Perception and Intelligent Control of High-end Equipment, Ministry of Education, Anhui Polytechnic University, China*
[f] *National Engineering Laboratory for Integrated Aero-Space-Ground-Ocean Big Data Application Technology, Northwestern Polytechnical University, China*

ARTICLE INFO

Article history:
Received 31 March 2020
Received in revised form 18 May 2020
Accepted 21 May 2020
Available online 2 June 2020

Keywords:
Vulnerability detection
Machine learning
Comparative study
Deep learning
Feature extraction

ABSTRACT

Machine learning-based Vulnerability detection is an active research topic in software security. Different traditional machine learning-based and deep learning-based vulnerability detection methods have been proposed. To our best knowledge, we are the first to identify four impact factors and conduct a comparative study to investigate the performance influence of these factors. In particular, the quality of datasets, classification models and vectorization methods can directly affect the detection performance, in contrast function/variable name replacement can affect the features of vulnerability detection and indirectly affect the performance. We collect three different vulnerability code datasets from two various sources (i.e., NVD and SARD). These datasets can correspond to different types of vulnerabilities. Moreover, we extract and analyze the features of vulnerability code datasets to explain some experimental results. Our findings based on the experimental results can be summarized as follows: (1) Deep learning models can achieve better performance than traditional machine learning models. Of all the models, BLSTM can achieve the best performance. (2) CountVectorizer can significantly improve the performance of traditional machine learning models. (3) Features generated by the random forest algorithm include system-related functions, syntax keywords, and user-defined names. Different vulnerability types and code sources will generate different features. (4) Datasets with user-defined variable and function name replacement will decrease the performance of vulnerability detection. (5) As the proportion of code from SARD increases, the performance of vulnerability detection will increase.

1. Introduction

In the field of software engineering, security is always an unavoidable topic. Hackers have launched many cyberattacks on vulnerabilities in software, which have caused high losses (Abdullah et al., 2018). Many different methods for detecting and fixing vulnerabilities have been proposed, but the number of potential threats reported in the *Common Vulnerabilities and Exposures* (CVE) (Common Vulnerabilities Exposures (CVE), 0000) is still increasing. It can be found that the defense based on human experience, project management technology and code analysis technology is still limited. It promotes the combination of machine learning with traditional vulnerability detection methods (Wan et al., 2019).

Existing vulnerability detection frameworks are mainly based on code analysis techniques. Code analysis techniques can be divided into two main categories: dynamic code analysis techniques and static code analysis techniques. In particular, the dynamic code analysis technique requires a real-time running environment and multiple tests to ensure the accuracy of the analysis. This makes it difficult for us to combine it with machine learning to further improve results. The dynamic analysis technique (such as Li, 2012) is more used for functional testing and performance testing. In contrast to the dynamic technique, the static technique can be easier combined with machine learning. The static code analysis is a study of source code, and does not require real-time tracking calculations. Static analysis tools

* Corresponding author.
E-mail address: wuxiaoxue00@gmail.com (X. Wu).

https://doi.org/10.1016/j.jss.2020.110659
0164-1212/© 2020 Elsevier Inc. All rights reserved.

(such as Cppcheck, 0000, FlawFinder, 0000, and Clang, 0000) are usually used for the detection of potential vulnerabilities. However, the most concerning problem of these tools is high false-negative rates. Therefore, the combination of machine learning (i.e., traditional machine learning (Tommy et al., 2017; Malik et al., 2019) or deep learning (Li et al., 2018b)) and static code analysis has become a new research topic.

There are two different granularity (i.e., the file level and the code level) in static code analyzing and vulnerability detecting. When analyzing at the granularity of function level, Tommy et al. (2017) concentrated on the file dependencies or the project complexity. File-level detection can help macro adjustments in software engineering. When analyzing at the granularity of code level, Li et al. (2018b) focused on the logic and variable of the code. Experience results show that checking at the granularity of code-level is finer, and the results of vulnerability detection are better. VulDeePecker is currently the best tool for vulnerability detection and is developed by Li et al. (2018b). The framework of VulDeePecker includes a set of processing procedures based on data flow. The purpose of these processing procedures is to generate *code gadget*. The *code gadget* contains almost all the function calls, variables, and a few noises, which is a kind of code slice. Therefore the focus of the problem is transformed from code analysis to code processing.

VulDeePecker provides researchers with a new vulnerability detection framework. The whole process can be divided into the following steps. The first step is the collection of the datasets. Li et al. collect code from two different sources. The second step is the preprocessing of the code. The preprocessing steps include code analysis, code slicing and user-defined name replacement. The third step is vectorization. VulDeePecker uses the Word2vec tool (Word2vec, 0000) to transform codes to vectors. Finally, these datasets that have been transformed into vectors will be passed to the machine learning classification model. VulDeePecker uses BLSTM (bi-directional LSTM) as the training model. In the framework of VulDeePecker, there are five key points worth studying:

- BLSTM is the only training model used by VulDeePecker. The training model can directly affect the results of vulnerability detection. Are there any other deep learning and traditional machine learning models, which can improve the results of vulnerability detection?
- The text vectorization method is also one of the factors, which can affect vulnerability detection. As a deep learning-based tool, VulDeePecker transforms codes into vectors through the use of the Word2vec tool. Because of the dimension problem, vectors need to be truncated. Is there a better vectorization method, which can achieve better performance?
- It is difficult to interpret the internals of trained deep learning models. We can get the features of vulnerability by using the feature extraction algorithm. These features may help us explain and analyze the experimental results. Are there any regular expressions that we do not know?
- VulDeePecker replaces the names of user-defined variables and functions. This step seems to improve the generalization ability of the trained model and remove some noises in the datasets. But whether this step could lead to a loss of some useful information is still unknown.
- When testing and measuring the vulnerability detection tool VulDeePecker, Li et al. used datasets built from two different sources. Obviously, there are some differences in the codes from these two sources. How code datasets from different sources affect the vulnerability detection result?

In this article, we conduct a comparative study to evaluate the impact of different factors. We studied the impact of these factors on three types of vulnerability. Specifically, to the best of our knowledge, our contributions can be summarized as follows:

First, we summarize the overall framework of machine learning-based vulnerability detection. After analysis and comparison, we select dataset source, vectorization methods, classification models and user-defined name replacement as the main factors for the study in this article.

Second, we focus on a new type of vulnerability and collect the code datasets using the same approach proposed by Li et al. (2018b).

Third, we perform feature extraction on the datasets gathered from real-world projects and conduct a large-scale comparative study on impact factors in machine learning-based vulnerability detection.

Some interesting findings can be summarized as follows: (1) The overall performance of deep learning models can achieve better performance than machine learning models. BLSTM is the best model for vulnerability detection. (2) For traditional machine learning, compared to other vectorization methods, CountVectorizer tool (CountVectorizer, 0000) can achieve better performance. (3) Features extracted from the vulnerability datasets contain many keywords, which are similar to the human experience. These features are related to preprocessing and data sources. (4) Datasets without user-defined variable and function name replacement can achieve better performance. (5) Experimental results show that the more proportion of code from SARD (Software Assurance Reference Dataset) increases, the performance of vulnerability detection will increase.

Paper organization: Section 2 reviews the related work for our study. Section 3 introduces the framework of machine learning-based vulnerability detection, the preprocessing of data and the classification models. Section 4 describes the datasets and experimental evaluation methods. Section 5 presents our experimental results and discusses the reasons. Section 6 summarizes the conclusion of our study and then discusses potential future work.

2. Related work

Prior studies related to machine learning-based vulnerability detection can be divided into traditional machine learning-based detection and deep learning-based detection. Previous studies on traditional machine learning-based vulnerability detection include the studies (Feng et al., 2015; Ahmadi et al., 2016; Kawaguchi and Omote, 2015; Khodamoradi et al., 2015; Kwon et al., 2015; Sexton et al., 2016; Chen et al., 2012; Dahl et al., 2013; Graziano et al., 2015; Raff and Nicholas, 2017; Mohaisen et al., 2015; Chen et al., 2020; Xu et al., 2018). Most of the preprocessing methods used in these studies are similar to ours. However, the experimental results of traditional machine learning-based vulnerability detection are not satisfactory. Just like (Feng et al., 2015; Ahmadi et al., 2016), many frameworks use accuracy rate or TP rate as the evaluation indicator, but the F1-score of their detection results is unfavorable. Previous studies on deep learning-based vulnerability detection mainly include the studies (Li et al., 2018b; Dahl et al., 2013; Li et al., 2019; Li et al., 2018a). It is worth mentioning that the abstract syntax tree is used in the study (Li et al., 2018a) and the experiment has achieved very good results.

There are also some studies, which analyze the influence of factors for vulnerability detection. Shabtai et al. (2009), Li et al. (2019), LeDoux and Lakhotia (2015), Souri and Hosseini (2018) and Xu et al. (2019) organized related factors on this topic, which contains classification algorithms, feature extraction methods and the class imbalance methods. Since these influencing factors are present in any machine learning training process, they look very macroscopic. The authors did not use the vulnerability detection framework as a starting point for research. Some details, such as datasets, always have a significant influence on the experimental results. Kawaguchi and Omote (2015) and Asquith (2015) studied the factors of vulnerability detection by combining dynamic analysis and static analysis. The methods combined with dynamic analysis are very intuitive, but very accidental.

The datasets used in this study are written by C/C++ programming language. Different from this study, there are some studies in other languages. Saccente et al. (2019) developed a prototype static method-level detection tool of Java source code. Malik et al. (2019) analyzed the detection of Android security vulnerabilities by combining machine learning and system calls. Both of these two frameworks are similar to the framework used in this study, and both have achieved satisfactory results. Ndichu et al. (2019) analyzed features of JavaScript by using vector representation, but they did not show the features.

We conduct a comparative study to evaluate the impact of different factors. However, most of the surveys like Shabtai et al. (2009), Li et al. (2019), LeDoux and Lakhotia (2015) and Souri and Hosseini (2018) did not mention the vectorization methods for vulnerability detection, the user-defined name replacement and the source of datasets. Therefore, our study is an important supplement to other previous studies in this research topic and our findings can provide guidance for better using the framework of vulnerability detection investigated in our study.

3. Machine learning-based Vulnerability detection

In this section, we first show the framework of machine learning-based vulnerability detection. Then we introduce the details of each factor investigated in our comparative study.

3.1. Framework

The machine learning-based vulnerability detection framework used in this article combines natural language processing technology and program analysis technology. In particular, the program analysis technology is mainly used for dataset construction and preprocessing. The natural language processing technology is mainly used to transform the code into the vector form that can be accepted by the classification model. Fig. 1 shows the overall framework of machine learning-based vulnerability detection. In this figure, machine learning-based vulnerability detection can be mainly divided into the following four steps.

First, we generate the *code gadget* through the data flow based code analysis method and label them. *Code gadget* is the basic unit of preprocessing and model training (Li et al., 2018b). In vulnerability detection, the granularity choice of vulnerability detection is an important topic. Without an appropriate vulnerability detection granularity, the location of the vulnerability detection cannot be determined. *Code gadget* is the product of data flow analysis and control flow analysis. It can meet the requirements of function-level vulnerability detection.

Second, we clean datasets by removing all of the non-ASCII characters and comments. Comments are not the focus of this research and the non-ASCII characters are useless during vulnerability detection. Although comments usually contain a lot of information (such as code functions and parameter explanations), there are no general rules for comments. In this article, we do not consider the impact of comments. We also replace the user-defined variable and function names with generic names. This step is one of the factors considered in our comparative study.

Third, we transform these *code gadgets* into vectors by using natural language processing techniques. Different vectorization methods will cause changes in the value and dimensions of the vector. This step has an essential impact on the classification performance of machine learning. At the same time, because different vectorization will produce different dimensions and different vector value ranges, this step will also affect the training time of machine learning models.

Finally, we divide the datasets into the training sets and the testing sets to train the model and then evaluate the performance of the trained model. To avoid the random factors in the process of the datasets division, we randomly divide the training and testing sets randomly for multiple times and the machine learning model will also be constructed multiple times. In this article, we will evaluate the model by combining the results of multiple experiments.

3.2. Replace function names and variable names

In Section 1, we mention that the step of replacing names of user-defined functions and variables is important. After removing all the comments of code, the datasets may still have noises because codes used in this article are from real-world software projects. During software development, developers often give meaningful names to variables and functions to improve the readability of the implemented code (Avidan and Feitelson, 2017). Obviously, we do not need to use these names to determine whether the code contains a potential vulnerability. These names without any regularity are the main source of noises in the gathered datasets. VulDeePecker (Li et al., 2018b) replaces the names of user-defined functions and variables with generic names. Then we show the specific steps for this preprocess. Suppose there are two user-defined variables and two user-defined functions called **userstr**, **usernum**, **Numfun** and **Strfun**. For one *code gadget*, these names will be transformed as **var1**, **var2**, **Fun1** and **Fun2**. This process is completely independent. Therefore, different *code gadget*s do not affect each other.

To be able to perform the replacement accurately and quickly, we use regular expressions to match the target in this article. In this process, due to the irregularity of custom names, we collect keywords related to the C language and C++ language as many as possible, and all the APIs related to the project. In this way, we can distinguish system keywords from user-defined names. Then we follow the above-mentioned replacement rules to complete this preprocessing step.

The impact of this step is one of the factors that this article focuses on. We want to know whether the generalization ability of the model can be further improved and whether this step can cause information loss. Moreover, we also want to study the effect of this step on the vulnerability detection results and features.

3.3. Text vectorization

Since machine learning models only accept matrices as their inputs, the codes need to be transformed into vectors. This step is actually one of the standard steps for natural language processing (Zheng et al., 2018, 2019). In this article, three vectorization methods are considered and need to be compared. The chosen methods are Index-based vectorization, CountVectorizer and Word2vec.

Index-based vectorization. This is the simplest vectorization method. It directly transforms different words into different numbers. Obviously, the vector lengths generated in this way are inconsistent. Therefore, if using index-based vectorization, we need to determine a suitable dimension and then adjust all the vectors, including interception and padding. When the dimension of the generated vector is less than the specified dimension, the tail of the vector needs to be filled with ‘0’. When the dimension

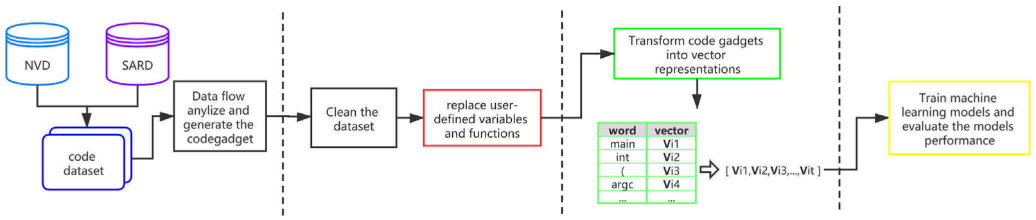

NVD
SARD
code dataset
Data flow anylize and generate the codegadget
Clean the dataset
replace user-defined variables and functions
Transform code gadgets into vector representations
word | vector
main | Vi1
int | Vi2
(| Vi3
argc | Vi4
... | ...
[Vi1,Vi2,Vi3,...,Vit]
Train machine learning models and evaluate the models performance

Fig. 1. Framework of machine learning-based vulnerability detection.

Table 1
Classification algorithms used in our comparative study.

Type of model	Algorithm
Traditional machine learning	Random Forest (RF) Gradient Boosting Decision Tree (GBDT) K-Nearest Neighbors (KNN) Support Vector Machine (SVM) Logistic Regression (LR)
Deep learning	Bi-directional LSTM (BLSTM) Gated Recurrent Unit (GRU) Convolutional Neural Networks (CNN)

of the generated vector is more than the specified dimension, the tail of the vector needs to be intercepted.

CountVectorizer. This is a kind of bag-of-words models. The entire vectorization process can be divided into two steps. First, CountVectorizer builds the words dictionary by counting the number of words in a dataset. Through the word dictionary, we can find the word and its corresponding frequency. Second, CountVectorizer converts the sentences into vectors. Vectors generated by CountVectorizer are aligned because the length of the vector is the word dictionary length. Notice the vectors generated by this method usually form a sparse matrix, which has a great impact on the computation load.

Word2vec. This method has been widely used in text mining (Wolf et al., 2014) and software engineering (Chen et al., 2019b). Word2vec can directly convert a word into a vector, which is trained by the corpus. The vector generated by Word2vec can contain semantic information. This means the closer the distance between word vectors, the closer the semantic of corresponding words. By merging vectors, we can get a vector representation of a piece of code. Obviously, we need to make the same adjustment as index-based vectorization.

3.4. Classification models

The classification model is one of the most important factors in machine learning-based vulnerability detection. Table 1 shows all the classification algorithms used for comparison in our experiment. The considered algorithms can be mainly divided into two categories: traditional machine learning algorithms and deep learning algorithms.

3.4.1. Traditional machine learning algorithms

Traditional Machine learning has been widely used because it has strong explanatory and good performance. In our study, we consider two ensemble learning models and three traditional machine learning models. In particular, ensemble learning is a kind of classification algorithm that can integrate multiple machine learners. We choose Random Forest (RF) and Gradient Boosting Decision Tree (GBDT) as representative of this kind of method. Traditional machine learning only uses one learner to perform classification task and we choose K-Nearest Neighbors (KNN), Support Vector Machine (SVM) and Logistic Regression (LR) as the representative of this kind of methods. Later, we show the details of these chosen traditional machine learning algorithms.

RF. RF is a classification algorithm that includes multiple decision trees. Each tree participates in classification tasks and gives results. RF is a classic ensemble learning algorithm. The final classification results will be elected through a voting-like mechanism. In addition, because RF is a decision tree-based algorithm, it has the same feature extraction capability as the decision tree. Besides using the voting mechanism to determine classification results, RF will also use this mechanism to select meaningful features.

GBDT. GBDT is also a classification algorithm based on multiple decision trees. Unlike RF, there is a mutual iterative process between the weak learners included in GBDT. GBDT does not have a voting mechanism, but connects the learners it contains. Each learner is trained on the previous learner and generates a weight. The final classification result is the weighted results of all the learners.

KNN. KNN is one of the easiest methods in data mining classification technology. KNN is a classification method that finds instances based on distance metrics (such as Euclidean distance). For one test sample, KNN first finds the nearest k neighbor samples in the training set according to the distance. Then determine the classification results of the test samples based on the labeled values in the neighbor samples.

SVM. SVM is a linear classifier located on the feature space. SVM classifies the test set on the hyperplane by solving the separated hyperplane with the largest geometric interval. SVM is a linear classification method and it can also solve non-linear problems. When performing a non-linear classification, the feature space needs to be upgraded using a kernel function.

LR. LR is a classification algorithm that uses the sigmoid function to discriminate discrete variables, and it is a kind of generalized linear model. LR and multiple linear regression have many similarities. The main difference is that their dependent variables are different. When the dependent variable obeys the binomial distribution, we prefer to use LR to achieve the purpose of binomial classification.

In these considered algorithms, RF and GBDT can retain the features when training the model. Machine learning models based on decision trees will automatically assign a score to features in the dataset during the training process. This provides algorithm support for our experiments in the feature extraction section.

3.4.2. Deep learning algorithms

Deep learning is commonly used in the field of natural language processing and software engineering (Chen et al., 2019a). Although the internal structure of the model is complex, deep learning models can always achieve good results on classification tasks. In VulDeePecker (Li et al., 2018b), it used BLSTM to classify code and detect vulnerabilities. Based on this method, we add two new methods for our comparative study when analyzing this factor: Convolutional Neural Networks (CNN) and Gated Recurrent Unit (GRU).

BLSTM. BLSTM includes a dense layer, a softmax layer and a number of BLSTM layers. Like most deep learning network structures, its input is a vector and its output is the classification result. The BLSTM layers contain some complex LSTM (Long-Short Term Memory) cells that have the memory function. In most time series problems, BLSTM can achieve better performance than traditional neural networks.

GRU. GRU is a type of RNN (Recurrent Neural Network). Like LSTM, it is also proposed to solve problems, such as long-term memory and gradients in back propagation. GRU is similar to LSTM in experimental results, but GRU is less computationally expensive and easier to train the network. Therefore, GRU can greatly improve training efficiency.

CNN. CNN is a type of feed-forward neural network that includes convolution calculations and has a deep structure. It is one of the representative algorithms for deep learning. CNN has the capability of representation learning, which can perform shift-invariant classification of input information according to its hierarchical structure. CNN imitates the visual perception mechanism construction, which can perform supervised and unsupervised learning. It has a stable effect and has no additional feature engineering requirements for the dataset (Gu et al., 2015).

4. Experimental setup

4.1. Datasets

Based on the analysis of previous vulnerability detection studies, we use two dataset sources (i.e., the National Vulnerability Database (NVD) (NVD, 0000) and the Software Assurance Reference Dataset (SARD) (Software Assurance Reference Dataset, 0000)) in our study. We mainly use Common Vulnerabilities and Exposures IDentifier (CVE ID) (Common Vulnerabilities Exposures (CVE), 0000) and Common Weakness Enumeration IDentifier (CWE ID) (Common Weakness Enumeration (CWE), 0000) to represent and find vulnerabilities. CVE ID contains specific information about a vulnerability (Chen et al., 2018), such as a release date and severity. CWE is a community-developed list of common software and hardware weakness types that have security ramifications. CWE ID contains information on the type of vulnerability. In the NVD, the specific information of each vulnerability can be found according to the CVE ID, and then we can find the corresponding code. But not every vulnerability has a patch or fix. In the SARD, we can find each vulnerability by searching one or more CWE ID because there are dependencies among different CWE ID. One CWE ID may be the parent of another. Anyway, they all correspond to the same kind of vulnerability. We use buffer error vulnerabilities (CWE 119), resource management error vulnerabilities (CWE 399) and improper resource lifetime control (CWE 664) for our experiment. First, two kinds of vulnerability have been used by VulDeePecker (Li et al., 2018b), and the third kind of vulnerability needs to be collected in the same way suggested by VulDeePecker (Li et al., 2018b).

In our study, we use another method to label these data. This method is similar to the labeling method used by VulDeePecker (Li et al., 2018b). In our used method, we provide different labeling methods for data from different sources. For program data from NVD, we label them by comparing source code and corresponding patches. Through the patch, we can know the location of the vulnerable code. We need to determine whether the vulnerable code is in *Code gadget*. However, for data from SARD, we have a different labeling method. Each program in the SARD is already labeled as *good*, *bad* and *mixed*. Program with *good* and *bad* can be easily labeled. But the program with *mixed* is special because it contains both vulnerability implementations and fixes. It is wrong to directly label these samples as "1" or "0" as suggested by VulDeePecker (Li et al., 2018b). Therefore, we do some processing manually. After observing the *mixed* code, we can find that the vulnerability implementation part and the fix part can be clearly delimited. Therefore, we can directly divide this code into the vulnerability part and the fixed part, and then label it as "1" and "0". Obviously, this will increase the number of programs and *code gadgets*. The statistics of positive and negative samples are shown in Table 2.

Table 2
Datasets for experiments and vulnerability statistics.

Vulnerability type	Total num	Vulnerable	Not vulnerable
CWE 119	40 113	10 531	29 582
CWE 399	24 551	7324	17 227
CWE 664	7051	2685	4366

Table 3
Statistics of data sources.

Vulnerability type	From SARD	From NVD
CWE 119	25 885	14 228
CWE 399	18 257	6294
CWE 664	6051	1000

Table 3 shows the source proportion of each dataset. Notice that in CWE 119 dataset, the proportion of the data from SARD is 64.5% ($= 25885/(25885 + 14228)$), in CWE 399 dataset, the proportion of the data from SARD is 74.4%. But in CWE 664 dataset, the proportion of the data from SARD is 85.8%. We also notice that there are some differences between codes from NVD and code from SARD. The code in NVD is not directly stored on the database website. NVD provides a lot of directed links of code sources. All of these links point to real software code. But the code in SARD is stored in the online database of the National Institute of Standards and Technology (NIST). To include vulnerabilities as many as possible, a lot of code in SARD are written to reproduce the specific vulnerability. Developers in SARD will change different control flows and data flows, and then reproduce the vulnerability. Therefore, this part of the code will be more targeted and more similar to each other. In addition, we can also construct some new datasets based on the types of vulnerabilities and data sources. By changing the proportion of the data from SARD and the data from NVD can help us study the impact of data sources on machine learning-based vulnerability detection.

4.2. Model evaluation method

In our comparative study, the performance of trained models is closely related to the training sets. Although we can divide the datasets into testing sets and training sets randomly, the training model only once always brings accidental results. Therefore, in this article, we will use data processing technology to randomly divide the datasets and run the machine learning models for multiple times. The entire model evaluation process is shown in Algorithm 1.

First, in our comparative study, we chose to shuffle the datasets directly and then randomly divide the original datasets into the training sets and the testing sets. This method can ensure that the size of the training sets and the testing sets are enough. To reduce the impact of accidents in datasets division, there must be enough iterations to shuffle and allocate the datasets. Notice that the random division for the training sets and the

Algorithm 1 Model Performance Evaluation Process

```
$shuffleNum = 30$
/*The repetition number*/
$codeData = \{code1, code2, \cdots, codeN\}$
$labelData = \{label1, label2, \cdots, labelN\}$
/*The code gadget dataset*/
$data = \{\}$
for $num = 0$ to $shuffleNum$ do
  $train, test = TrainTest\_split(codeData, labelData)$
  /*Randomly divide Data into train and test*/
  if $IsNotSimilar(\{train, test\}, data)$ then
    $data.apppend(\{train, test\})$
    /*Control the similarity*/
  else
    $num = num - 1$
    Continue
  end if
end for
$classificationModel = \{RF, KNN, \cdots, BLSTM\}$
/*Collection of classification algorithms*/
$result = \{\{RF, \varnothing\}, \{KNN, \varnothing\}, \cdots, \{BLSTM, \varnothing\}\}$
/*Performance of classification algorithms*/
for each $train, test$ in $data$ do
  for each $model$ in $classificationModel$ do
    $scores = evaluate(model, train, test)$
    /*Record the results*/
    $result[model].append(scores)$
  end for
end for
/*Take the average as the final evaluation*/
$evaluation = average(result)$
```

testing sets are not unlimited. To ensure the fairness of the experiment and improve the evaluation efficiency, we will control the similarity between training sets generated by each iteration to avoid duplication issues in the datasets division. From the experimental results, if the similarity between training sets is controlled, this evaluation method can obtain a stable result.

Then, we need to record the experimental results of each trained model in each iteration. Due to the relatively high-quality of our used datasets, the results of each round of iteration are relatively close. After repeating multiple shuffles and dataset divisions, for each trained model, we can get a set of classification results in terms of different performance measures. We set the number of repetitions to 30, which means that for each model, 30 experimental results can be recorded. The final evaluation result will be the average results of these performance measures.

The feature extraction experiment is similar to the above process. Repeatedly shuffling the datasets allows us to get multiple trained models and various feature extraction results. In this article, we use the intersection of multiple results as the final result of feature extraction.

4.3. Model performance measures

Vulnerability detection can be modeled as a classical binary classification task. We use *confusion matrix* shown in Table 4 to record four possible outputs of binary classification. Notice the positive instance means vulnerable sample and the negative instance means non-vulnerable sample. Based on this *confusion matrix*, we can compute the following performance measures: Recall (R), Precision (P), F1-score (F1), False-Positive Rate (FPR) and False-Negative Rate (FNR).

Recall. Recall rate is a measure of true-positive sample coverage. It shows the ability to predict positive examples. The Recall is the ratio of true-positive samples to the total samples that are classified correctly. It can be calculated by the following formula:

Table 4
Confusion matrix for vulnerability detection.

	Actually positive	Actually negative
Predict Positive	TP	FP
Predict Negative	FN	TN

$$R = \frac{TP}{TP + FN} \quad (1)$$

Precision. Precision rate indicates how many of the samples predicted to be positive are truly positive samples. The Precision is defined by the ratio of true-positive samples divided by the total samples that are detected as true. It can be calculated by the following formula:

$$P = \frac{TP}{TP + FP} \quad (2)$$

F1-score. F1-score (F1) denotes the overall effectiveness that considers both precision and recall rate. It is one of the most commonly used measures to evaluate the performance of the trained machine learning-based model. F1-score is the harmonic mean of Recall and Precision. It can be calculated by the following formula:

$$F1 = \frac{2 * P * R}{P + R} \quad (3)$$

False-Positive Rate and False-Negative Rate. False-Positive Rate (FPR) and **False-Negative Rate** (FNR) are used to count the number of wrong samples. In particular, FPR is the ratio of false-positive samples to the whole samples that are not vulnerable. FNR is the ratio of false-negative samples to the whole samples that are vulnerable. FPR and FNR represent the ratio of prediction error, and they are defined by Eqs. (4) and (5).

$$FPR = \frac{FP}{FP + TN} \quad (4)$$

$$FNR = \frac{FN}{TP + FN} \quad (5)$$

5. Experimental results

In this article, we mainly studied the effect of different factors (such as classification models and different preprocess methods) on vulnerability detection. In our comparative study, we focus on the following five research questions (RQs):

RQ1: How different machine learning models impact the performance of vulnerability detection?
RQ2: Which vectorization method is best for vulnerability detection?
RQ3: What features does machine learning focus on when performing vulnerability detection? Then are these features in line with expectations from the human experience?
RQ4: How does user-defined variable and function name replacement affect the performance of vulnerability detection?
RQ5: How do datasets from different sources affect the performance of vulnerability detection?

The purpose of RQ1 and RQ2 is to analyze the impact of algorithms on vulnerability detection. The framework of detection includes the steps of natural language processing, so the classification model and vectorization algorithm are particularly crucial for the experimental results. The purpose of RQ3 is to explain the conclusion of RQ4 and RQ5. Only by analyzing the features that machine learning focuses on can we know the impact of changes in datasets and names replacement on vulnerability detection.

5.1. RQ1: How different machine learning models impact the performance of vulnerability detection?

In Section 3.1, we introduced the framework of machine learning-based vulnerability detection. VulDeePecker (Li et al., 2018b) uses deep learning models and replace the names of user-defined variables and functions. In this experiment, we will use the same preprocess method as VulDeePecker to compare the classification models mentioned in Section 3.4. The experimental results of the vulnerability detection are shown in Table 5.

First, we can find from Table 5 that the overall performance of the deep learning algorithms are better than the traditional machine learning algorithms. The deep learning models can increase the performance in terms of F1-score measure by 18.78%, increase the performance in terms of recall measure by 24.71%, and increase the performance in terms of precision measure by 8.41% on average. Based on the empirical results, the deep learning models are more suitable for vulnerability detection indeed. At the same time, the average performance of the deep learning algorithms in terms of false-positive rate is 2.32%, which is lower than the traditional machine learning algorithms. The average performance of the deep learning algorithms in terms of false-negative rate is 24.07%, which is lower than the traditional machine learning. It can be found that two performance measures (i.e., the recall rate and the false-negative rate) are the main performance gap between the traditional machine learning methods and the deep learning methods. The number of false-negative samples is significantly different. This means that deep learning-based detection models can detect more vulnerabilities while having fewer prediction errors.

Second, according to Table 5, we find that the BLSTM model can achieve the best performance in vulnerability detection. Specially, The BLSTM model can achieve an F1-score of 88.72%, 93.13% and 97.79% on all these three datasets respectively. In terms of other performance measures, the BLSTM model can also achieve the best performance.

Third, based on the results of the traditional machine learning models, we find that the ensemble learning models can achieve better vulnerability detection performance than the simple machine learning. Specifically, RF performs best on the first two datasets. The F1-score of RF is 71.59% on CWE 119 dataset and 88.23% on CWE 399 dataset. On CWE 664 dataset, GBDT has the highest F1-score of 96.08%. Although the experimental results of RF are somewhat different from BLSTM, these differences are not huge. This means that the features we extract using the scoring mechanism in RF will have great reference value.

In summary, the answers for RQ1 are: (1) In terms of F1, the average performance of the deep learning models is 18.78% higher than the traditional machine learning models; (2) BLSTM is the best machine learning algorithm for machine learning-based vulnerability detection; (3) RF performs best in traditional machine learning algorithms.

5.2. RQ2: Which vectorization method is best for vulnerability detection?

To answer RQ2, we study the effect of different vectorization methods illustrated in Section 3.3 on the performance of the trained model. In this RQ, we only compare the impact of different vectorization on the traditional machine learning models because the deep learning algorithms itself use the embedding layer for the word embedding work. We also evaluate the model performance on different datasets without user-defined variable and function replacement. We do not choose to do this step because we want the datasets can keep as much information as possible.

We compute the average performance of the machine learning models in this experiment with different vectorization methods. The results can be found in Table 6. In this table, we observe that the best experimental result is given by using CountVectorizer tool in all of the three datasets. The average performance of the machine learning models with Index-based vectorization and Word2Vec in terms of F1-score are 66.8% and 74.1% respectively, but the performance of the models with CountVectorizer in terms of F1-score is 87.1%. In other words, the experimental results of CountVectorizer are indeed better and vectors generated by CountVectorizer can contain more useful information for classification.

The above results do not satisfy our expectations. We think this phenomenon is caused by the processing step after vectorization. As we all know, the number of words and symbols in different codes may be inconsistent, and therefore, when we convert words into vectors and combine these vectors into vectors of the sentence, their dimensions are also inconsistent. When using vectorization methods to transform all the code into vectors, to align dimensions, vectors usually need to be padded and truncated. This operation will cause a loss of information. Both Index-based Vectorization and Word2vec have this problem. But Countvectorizer does not have this problem because the dimension of vectors transformed by Countvectorizer can be automatically aligned, and these vectors do not need to be adjusted. Although Countvectorizer can only transform the dataset into a sparse matrix, it also ensures that all the information is retained. This can explain why Countvectorizer has the best performance in vulnerability detection. However, if a good dimension can be determined, Word2vec may still be a satisfactory vectorization method.

In summary, the answer for RQ2 is that Countvectorizer has the best performance on the traditional machine learning-based vulnerability detection.

5.3. RQ3: What features does the machine learning focus on when performing classification tasks? then are these features in line with expectations from the human experience?

Human experts always use features to detect vulnerabilities. These features are usually summarized by human experts and contain a lot of keywords and specific system functions. Therefore, features can help us explain a lot of questions and rules in datasets. In this RQ, we use RF algorithm to extract features because RF can score features and performs well on all the three datasets. We select the top 100 features with the highest score and show them in Table 7. We mark all system-related features, which include code syntax keywords and system functions. We also considered the impact of user-defined variable and function replacement. Findings in this experiment can be summarized as follows:

First, features extracted by RF algorithm are very similar to those defined by human experience. In the result, there are many syntax keywords, system functions, user-defined variables and user-defined functions. For example, experts often detect vulnerabilities through system functions because system functions have many historical problems, meanwhile, our experimental results also include many system functions (such as “memcpy”, “malloc” and so on). These functions have already been proven to be unsafe. The reason why the RF algorithm can achieve results similar to artificial experience features is that RF performs well in the classification task on the three vulnerability datasets.

Table 5
The performance of classification models on vulnerability detection.

Vulnerability type	Type of model	Model	F1	P	R	FPR	FNR
CWE 119	Traditional machine learning	**RF**	**0.715913**	0.810887	**0.640854**	0.053368	**0.359145**
		KNN	0.64041	0.683926	0.602089	0.100262	0.397911
		SVM	0.415572	0.729150	0.290598	0.038895	0.709401
		GBDT	0.448598	**0.840630**	0.305927	**0.020710**	0.694072
		LR	0.222113	0.481402	0.144359	0.055530	0.855640
	Deep learning	CNN	0.817201	0.833333	0.801681	0.057385	0.19832
		BLSTM	**0.887202**	**0.933003**	**0.845609**	**0.034530**	**0.154299**
		GRU	0.800585	0.817965	0.783928	0.06244	0.216072
CWE 399	Traditional machine learning	**RF**	**0.882338**	0.925025	**0.843418**	0.033628	**0.156581**
		KNN	0.852248	0.878802	0.827251	0.056123	0.172748
		SVM	0.789019	0.909036	0.696997	0.034310	0.303002
		GBDT	0.747936	**0.925119**	0.627713	**0.024994**	0.372286
		LR	0.627906	0.712609	0.561200	0.111338	0.438799
	Deep learning	CNN	0.907594	0.931252	0.885108	0.033012	0.114892
		BLSTM	**0.931375**	**0.939109**	**0.923768**	**0.030621**	**0.076232**
		GRU	0.907547	0.913175	0.901987	0.043329	0.098013
CWE 664	Traditional machine learning	RF	0.950793	**0.959935**	0.941823	**0.021186**	0.058176
		KNN	0.848081	0.903914	0.798742	0.045762	0.201257
		SVM	0.869085	0.871835	0.866352	0.068644	0.133647
		GBDT	**0.960815**	0.957812	**0.963836**	0.022881	**0.036163**
		LR	0.805401	0.813804	0.797169	0.098305	0.202831
	Deep learning	CNN	0.974911	0.942731	0.989803	0.001401	0.010162
		BLSTM	**0.977907**	**0.956205**	**0.995999**	0.000200	**0.004061**
		GRU	0.951822	0.932121	0.983714	**0.000199**	0.016260

Table 6
Impact of different vectorization methods based on machine learning.

Vulnerability type	Vectorization methods	F1	P	R	FPR	FNR
CWE 119	Index-based	0.509	0.699	0.433	0.056	0.567
	CountVectorizer	0.773	0.861	0.706	0.042	0.294
	Word2Vec	0.523	0.696	0.438	0.067	0.562
CWE 399	Index-based	0.762	0.874	0.681	0.048	0.319
	CountVectorizer	0.853	0.948	0.793	0.021	0.207
	Word2Vec	0.778	0.867	0.709	0.052	0.291
CWE 664	Index-based	0.733	0.835	0.731	0.039	0.269
	CountVectorizer	0.986	0.987	0.985	0.007	0.015
	Word2Vec	0.921	0.911	0.929	0.049	0.071

Second, user-defined function and variable replacement can increase the number of system functions in features. In Section 4.1, we mentioned the noises in the code datasets. User-defined variable and function replacement is actually an attempt to reduce the noises. After converting a variety of user-defined variables and function names into names with a uniform law, the importance of system functions and syntax keywords has increased. From the experimental results, we can observe that this step increases the number of system-related features. But the importance of user-defined names is weakened. This experimental result can partially explain the impact of user-defined variable and function replacement on the vulnerability detection.

Third, we find that feature extraction results are related to the source of datasets. As the proportion of code from SARD increases, the number of system-related features decreases. In Table 3 we can find that in the dataset of CWE 119, the proportion of code from SARD is 64.5%, in the dataset of CWE 399, the proportion of code from SARD is 74.4% and in the dataset of CWE 664, the proportion of code from SARD is 85.8%. The experimental results of RQ3 show that when we do not perform name replacement, the number of system-related features of CWE 119 is 31, the number of system-related features of CWE 399 is 28 and the number of system-related features of CWE 644 is 20. That means when we perform feature extraction on code from NVD, we can get more system functions and keywords, when we perform feature extraction on code from SARD, we can get more user-defined functions and names.

In summary, the answers for RQ3 are: (1) Features that machine learning models focus on include system functions, syntax keywords which are similar to human experience; (2) User-defined variable and function replacement can enhance the system-related features; (3) The increase in the proportion of code from SARD leads to a decrease in system-related features.

5.4. RQ4: How does user-defined variable and function name replacement affect the performance of vulnerability detection?

The function variables and user-defined name replacement are introduced in Section 3.2. Based on the experiment in RQ3, we find that replacement can enhance the number of system-related features. Therefore the answer for RQ3 includes the effect of user-defined function and variable name replacement on features. However, in this experiment, we want to study this effect on classification models and vulnerability detection performance. We use Word2vec for vectorization and calculate the average performance of the trained models from all the three datasets.

The classification performance of the models can be shown in Table 8. We study the effect of user-defined function and variable name replacement by comparing experimental results. It can be found from Table 8 that both the deep learning models and the traditional machine learning models without functions and user-defined variable name replacement have better vulnerability detection performance.

To explain the experiment result of RQ4, we need to analyze the conclusions in RQ3. Table 7 shows the features, including

Table 7
Features of vulnerability datasets.

Vulnerability type	Name replace	Features
CWE 119	No	data badsink goodg2bsink databadbuffer source badsource **char** datagoodbuffer wchar dest **void** databuffer buffer **sizeof** pr **int null if malloc return** state **free** goodg2bsource **strlen static new for** elsedata bad **strncpy strcpy** alloca **wcscpy** pointer **memset memcpy** size before baddata **wmemset wcslen** wide va split **snprintf memmove** action overflow inputbuffer **printf** printline **atoi** goodg2b declare stonesoup dataref src tcp goodb2gsink info datalen string **const wcsncpy** intpointer array **fscanf** format goodg2b1source recvresult goodg2bdata stack root recv datacopy start testdata buf dataptr2 argptr trigger global goodg2b2source stdin heap logfile buff list svoid uniontype oc **break** other datamap globaltrue **sprintf struct** path **vfprintf** false
	Yes	var2 var0 var4 var3 var1 var5 **char** var6 fun0 fun1 **void sizeof int if** var7 **strlen free malloc** fun2 **static strcpy memset** var8 **strncpy** var9 **getenv for new memcpy return** var10 **memmove wcsncpy wmemset wcscpy snprintf const** fun3 var12 **atoi sprintf** recv var11 var14 var13 **while** var15 **struct** fun4 **calloc fgets printf strncat** stdin var18 var17 **fprintf** var0 **void fopen strcat** elsevar4 elsevar5 var20 var16 elsevar6 **unsigned** var21 **break vfprintf fscanf** var19 var22 **case** var24 fun5 **goto** elsevar3 **switch** elseint var23 elsechar **else** fun6 var29 elsevar7 var26 **strchr** var25 **fread** var27 **fseek double sscanf typedef delete default vsnprintf fclose** elsevar2
CWE 399	No	data badsink **sizeof** mystring size **strcpy null** goodb2gsink badsource **free wcslen new** string **char malloc delete if** inputbuffer **strlen** wchar **void** txt **rand strtoul fopen for** hello stonesoup recv goodg2bsink **break sprintf** param goodb2gsource files **int** goodg2bsource ud **static** alloc stdin finished array num evaluating **fgets** open htons **memset fscanf** ulimit dataarray goodb2g2sink printline dataref filename sock datalen contents buffer read uncontrolled file opt md socket **return wcscpy** error map elsechar bind funcptr setup format len temp **unsigned** baddata start index elsedata connectsocket list st **isalnum** update at inaddr stream back datacopy datamap trace ss context memory tainted due **fflush**
	Yes	var5 var6 var7 var3 var2 var4 data **free malloc** fun0 **sizeof if** var8 **strlen char** var0 var1 **void** fun2 fun1 var11 var9 **int** var10 **for** stdin **rand** fun3 **wcscpy break strtoul delete** var12 **static return strcpy new** var17 var15 **fclose struct** var13 fun4 **sprintf** var16 **unsigned while fopen fread** var19 **memset** var14 **fscanf fflush** var18 elsechar **snprintf feof** recv fun5 **getenv fgets vfprintf** var22 var24 **long strncat** var23 var25 var27 var21 **memcpy goto** var20 **sscanf const** var31 var26 fun7 constvar2 elsevar4 var29 **atoi else** fun6 var28 fun8 var36 **fputs rewind switch** elsevar7 var32 var30 **calloc** 79s var35 **fseek** elsevar6 **ftell**
CWE 664	No	**static** bad badsink goodg2b fixed of heap uninitialized memory w32 on string use **free** wchar variable uncontrolled data goodg2b1 **malloc** goodg2bsink listen not databuffer goodg2b2 **if** initialization printline **else** size **sizeof** confusion **for** datalen search between extension exposure **int** family **exit** failed improper **atoi** badsource in file goodb2g2 ip dataptr provider basic wsacleanup be **strlen char** pointer password **void** array stdin shortbuffer phandle environment **getenv** operation error accept globalreturnstrueorfalse no sign incomplete goodb2g unverified bool numeric staticreturnstrue **default** we resource **wcsncat** that tcp filename to init virtuallock addr goodb2gsink **while return** good1 incorrect **memset** stdthreadlock 17 stdthreadlockrelease variables access **wcslen**
	Yes	**free malloc** var0 var3 **if** var4 **char** size **for** var1 var2 **atoi int** var8 **sizeof** fun0 var11 **else void** var5 fun2 fun1 var6 var9 var7 fun3 **fgets wmemset wcslen** fun9 **short return struct** var30 fun7 var10 var17 var29 fun4 **memset** var19 **double** fun5 **wcschr** var18 var25 **strlen** var32 **getenv atof** var20 **wcscat** var13 var21 var23 var26 **fscanf strcpy long** var15 var16 var31 **case** var24 **strcat while** var22 **goto fopen wcsncat break** fun6 var14 var12 var27 **fclose fgetws** var28strncat **default fwprintf** fun11 **switch strchr fprintf extern** var37 **float** fun10 **do** fun8 var34 **wcscpy sqrt const** fun12 fun13 fun14 **fwscanf** var33 var35

Table 8
The performance of user-defined name replacement on vulnerability detection.

Vulnerability type	Type of model	Name replace	F1	P	R	FPR	FNR
CWE 119	Traditional machine learning	No	0.522603	0.696169	0.437771	0.067162	0.562228
		Yes	0.488521	0.709199	0.396765	0.053753	0.603234
	Deep learning	No	0.879868	0.903231	0.857733	0.032973	0.142266
		Yes	0.834995	0.861433	0.810402	0.051451	0.189563
CWE 399	Traditional machine learning	No	0.781864	0.867299	0.712979	0.051755	0.291022
		Yes	0.779889	0.870118	0.711316	0.052079	0.288683
	Deep learning	No	0.955677	0.956832	0.954533	0.021779	0.045467
		Yes	0.915505	0.927845	0.903621	0.035654	0.096379
CWE 664	Traditional machine learning	No	0.920228	0.911045	0.929857	0.049196	0.070142
		Yes	0.886835	0.901461	0.873584	0.051355	0.126415
	Deep learning	No	0.972815	0.982804	0.961312	0.000200	0.004065
		Yes	0.968213	0.943685	0.989501	0.000600	0.010162

many system-related features. Obviously, user-defined functions and variables always get a high score because we can find features (such as "goodsink", "databuffer" or "goodb2g") in datasets without name replacement. It is easy to find this part of features contains semantics and sufficient information. These features can directly affect the final classification results. This is why user-defined function and variable name replacement can increase the average performance of the classification models and lead to a loss of some available information.

Another important reason is the proportion of source. The answers for RQ3 show that the more codes from SARD in the datasets, the fewer system-related features we will get. In contrast, the impact of codes from NVD can be almost ignored because they are from real-world software projects. However, codes from SARD is different. The purpose of codes from SARD is to

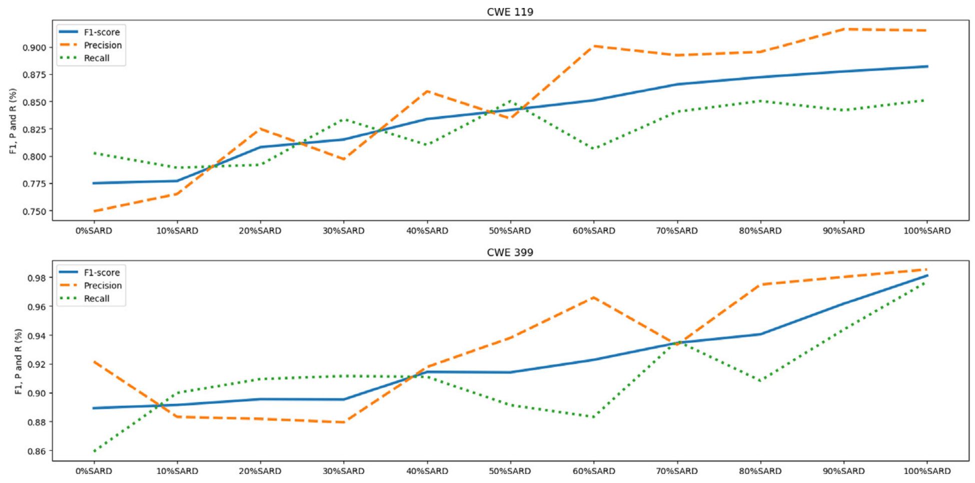


Fig. 2. The impact of different data source proportions on vulnerability detection.

reproduce the vulnerability, names of functions and variables in the codes will contain a lot of information. Such code can provide more regular features for machine learning and help the developers understand the principle of vulnerability. Therefore, the loss of information is closely related to the source of datasets.

In summary, the answer for RQ4 is that models without user-defined variable and function name replacement have better vulnerability detection performance.

5.5. *RQ5: How do datasets from different sources affect the performance of vulnerability detection?*

In Section 4.1, we introduced the datasets used in our comparative study and the corresponding sources. The code data are collected through two different websites (i.e., SARD and NVD). We also mention some differences and priorities of the two data sources between these two sources. The significance of RQ5 is to explore how the vulnerability code collected from different sources affects the final vulnerability detection performance. To answer RQ5, we study the effect of sources on classification by adjusting the proportion of sources in each dataset. We counted all the codes from SARD and NVD and divided them into two different parts according to the source. Then we built a series of new datasets with different proportions of sources in a random way. In this way, for one vulnerability type, we made ten different datasets and used them for experiments. In these datasets, the proportion of code from SARD spans 0% to 100%, while the proportion of code from NVD spans from 100% to 0%. In this process, although the datasets changed, the proportion of positive and negative samples still remains the same. We use BLSTM to train the model in this experiment.

Fig. 2 shows the experimental results of RQ5. Obviously, we can find that differences in the datasets will lead to changes in the results. As we increase the proportion of code from SARD spans 0% to 100%, the overall performance of the machine learning model also increases. When we build the dataset entirely using codes from NVD, the F1 scores of the model on two vulnerabilities are 77.5% and 88.9% respectively. In contrast, when we build the dataset entirely using codes from SARD, both results will grow to 88.2% and 98.1%. Although the precision rate and recall rate do not seem to be as stable as the f1 score, their growth trend is still obvious.

The reason for such experimental results is the code. In Section 4.1, we mention the difference between code from SARD and NVD. The code from NVD are vulnerabilities that exists in the real project. NVD provides source links for each vulnerability so that the developers can trace back to the project source code to do more research. This setting can guarantee the authenticity of the source code. However, not all codes from SARD are from real-world projects, especially those with a label of *mixed*. This part of the codes are sets of test cases, which are used for static code analysis. This means that for one vulnerability and its patch, SARD generates test cases by combining them with different control flows and data flows. This will lead to the high similarity between test cases, and these test cases will eventually become datasets for experiments. Fig. 3 shows two samples of code from SARD, which were generated by different control flows. Both codes are patches of the same vulnerability. It can be clearly seen that except for the “if” statement, the rest of the code is very similar. Datasets with such codes can easily affect the experimental results of machine learning.

In the process of building a machine learning model, if the testing set and the training set are similar in content, the experimental results of the model must be excellent. But this departs from the original intention of vulnerability detection. Evaluating a machine learning-based vulnerability detection framework should not only use metrics for machine learning models. We should use more real-world projects to verify and assess the quality of the framework.

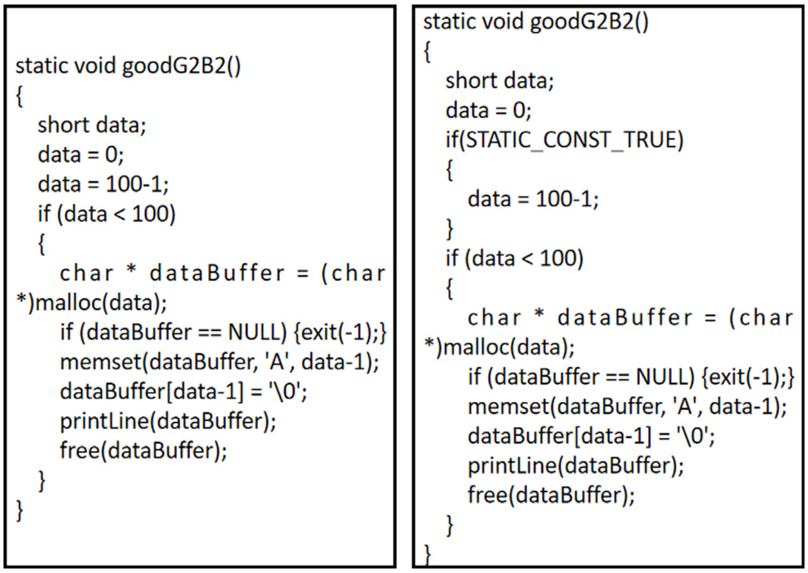
In summary, the answer for RQ5 is that as the proportion of code from SARD increases, the performance of machine learning-based vulnerability detection gets better.

5.6. *Threats to validity*

There are two possible threats to the effectiveness of our comparative study in this article.

Internal validity: Statistical biases are major threats to internal effectiveness. There are many uncertain issues in the machine learning process, such as sample imbalance and hyperparameter adjustment. As the size of the dataset increases, more and more factors need to be controlled in the experiment. It is easy to cause accidental experimental results. To alleviate this potential threat, we used a method that randomly shuffles the dataset and repeats iterative runs. This ensures that the results of the machine learning models can converge to stable values. However, the number of iterations still depends on the size and quality of the dataset. We also need to ensure that significantly different training sets are generated in each round. In this way, we try to eliminate the impact of uncertainty as much as possible.

External validity: The dataset itself also threatens the external validity. For example, the dataset may have some default values



```
static void goodG2B2()
{
    short data;
    data = 0;
    data = 100-1;
    if (data < 100)
    {
        char * dataBuffer = (char
*)malloc(data);
        if (dataBuffer == NULL) {exit(-1);}
        memset(dataBuffer, 'A', data-1);
        dataBuffer[data-1] = '\0';
        printLine(dataBuffer);
        free(dataBuffer);
    }
}
```

```
static void goodG2B2()
{
    short data;
    data = 0;
    if(STATIC_CONST_TRUE)
    {
        data = 100-1;
    }
    if (data < 100)
    {
        char * dataBuffer = (char
*)malloc(data);
        if (dataBuffer == NULL) {exit(-1);}
        memset(dataBuffer, 'A', data-1);
        dataBuffer[data-1] = '\0';
        printLine(dataBuffer);
        free(dataBuffer);
    }
}
```

Fig. 3. Samples of code from SARD.

and wrong items. In fact, the high similarity of the samples mentioned in RQ5 also threatens the external validity. Due to the large size, the dataset itself must have defects that cannot be directly observed. Therefore, it is important to evaluate the quality of datasets. Another problem is the accuracy of the label. Incorrect labels will directly affect the results of the machine learning model. In this article, we collected datasets from NVD and SARD based on VulDeePecker (Li et al., 2018b) and we used a new labeling method to label data from SARD. This step reduces the probability of mislabeling.

6. Conclusions and future work

In this article, we study the factors for machine learning-based vulnerability detection. The main findings in our study include: (1) The deep learning models have a better performance than the traditional machine learning models, BLSTM is the best model for vulnerability detection. (2) Using CountVectorizer can improve the performance of machine learning models. (3) Features extracted from code datasets are similar to the human experience and will be affected by the source proportion and preprocess methods. (4) User-defined variable and function name replacement improves the number of system-related features, but they not helpful for vulnerability detection. (5) As the proportion of code from SARD increases, the performance of machine learning-based vulnerability detection will get better.

However, there is still room for improvement in our comparative study. First, we only focus on three types of vulnerabilities. Future research should be conducted by considering more vulnerabilities and more datasets because different types of vulnerabilities have different analysis methods and characteristics. Second, from the experiment result, we find that codes from NVD and SARD show different characteristics and experimental results. Each dataset itself still has many factors worth exploring. How to fairly evaluate the performance of vulnerability detection is also a topic worth discussing. Third, future research will use some more accurate and stable evaluation models. Finally, we will also investigate other advanced modeling methods (Chen et al., 2018; Ni et al., 2019; Xu et al., 2019) and data preprocessing methods (Liu et al., 2015; Ni et al., 2017).

CRediT authorship contribution statement

Wei Zheng: Conceptualization, Funding acquisition, Project administration. **Jialiang Gao:** Investigation, Software. **Xiaoxue Wu:** Conceptualization, Investigation, Validation. **Fengyu Liu:** Software. **Yuxing Xun:** Software. **Guoliang Liu:** Software. **Xiang Chen:** Supervision.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgment

The work described in this paper was partially supported by the Key Laboratory of Advanced Perception and Intelligent Control of High-end Equipment, Ministry of Education, Anhui Polytechnic University (Grant No.GDSC202006), Innovation Foundation for Doctor Dissertation of Northwestern Polytechnical University (Grant No.CX202067).

References

Abdullah, M.S., Zainal, A., Maarof, M.A., Nizam Kassim, M., 2018. Cyber-Attack Features for Detecting Cyber Threat Incidents from Online News. In: Proceedings of 2018 Cyber Resilience Conference, pp. 1–4.
Ahmadi, M., Ulyanov, D., Semenov, S., Trofimov, M., Giacinto, G., 2016. Novel feature extraction, selection and fusion for effective malware family classification. In: Proceedings of the Sixth ACM Conference on Data and Application Security and Privacy. In: CODASPY '16, ACM, New York, NY, USA, pp. 183–194.
Asquith, M., 2015. Extremely scalable storage and clustering of malware metadata. J. Comput. Virol. Hacking Tech. 12.
Avidan, E., Feitelson, D.G., 2017. Effects of variable names on comprehension: An empirical study. In: 2017 IEEE/ACM 25th International Conference on Program Comprehension (ICPC). pp. 55–65.
Chen, Q., Bao, L., Li, L., Xia, X., Cai, L., 2018. Categorizing and predicting invalid vulnerabilities on common vulnerabilities and exposures. In: 2018 25th Asia-Pacific Software Engineering Conference (APSEC). pp. 345–354.
Chen, D., Chen, X., Li, H., Xie, J., Mu, Y., 2019a. Deepcpdp: Deep learning based cross-project defect prediction. IEEE Access 7, 184832–184848.
Chen, X., Chen, C., Zhang, D., Xing, Z., 2019b. Sethesaurus: Wordnet in software engineering. IEEE Trans. Softw. Eng..

Chen, Z., Roussopoulos, M., Liang, Z., Zhang, Y., Chen, Z., Delis, A., 2012. Malware characteristics and threats on the internet ecosystem. J. Syst. Softw. 85 (7), 1650–1672, Software Ecosystems.
Chen, X., Zhao, Y., Cui, Z., Meng, G., Liu, Y., Wang, Z., 2020. Large-scale empirical studies on effort-aware security vulnerability prediction methods. IEEE Trans. Reliab. 69 (1), 70–87.
Chen, X., Zhao, Y., Wang, Q., Yuan, Z., 2018. Multi: Multi-objective effort-aware just-in-time software defect prediction. Inf. Softw. Technol. 93, 1–13.
Clang, 0000. https://clang-analyzer.llvm.org/.
Common Vulnerabilities Exposures (CVE), 0000. http://cve.mitre.org/.
Common Weakness Enumeration (CWE), 0000. http://cwe.mitre.org/index.html.
CountVectorizer, 0000. https://scikit-learn.org/stable/modules/generated/.
Cppcheck, 0000. http://cppcheck.wiki.sourceforge.net/.
Dahl, G.E., Stokes, J.W., Deng, L., Yu, D., 2013. Large-scale malware classification using random projections and neural networks. In: Proceedings of 2013 IEEE International Conference on Acoustics, Speech and Signal Processing, pp. 3422–3426.
Feng, Z., Xiong, S., Cao, D., Deng, X., Wang, X., Yang, Y., Zhou, X., Huang, Y., Wu, G., 2015. HRS: A hybrid framework for malware detection. In: Proceedings of the 2015 ACM International Workshop on International Workshop on Security and Privacy Analytics. In: IWSPA '15, ACM, New York, NY, USA, pp. 19–26.
FlawFinder, 0000. http://www.dwheeler.com/flawfinder/.
Graziano, M., Canali, D., Bilge, L., Lanzi, A., Balzarotti, D., 2015. Needles in a haystack: Mining information from public dynamic analysis sandboxes for malware intelligence. In: Proceedings of the 24th USENIX Conference on Security Symposium. In: SEC'15, USENIX Association, Berkeley, CA, USA, pp. 1057–1072.
Gu, J., Wang, Z., Kuen, J., Ma, L., Shahroudy, A., Shuai, B., Liu, T., Wang, X., Wang, G., 2015. Recent advances in convolutional neural networks. Pattern Recognit..
Kawaguchi, N., Omote, K., 2015. Malware function classification using APIs in initial behavior. In: Proceedings of 2015 10th Asia Joint Conference on Information Security, pp. 138–144.
Khodamoradi, P., Fazlali, M., Mardukhi, F., Nosrati, M., 2015. Heuristic metamorphic malware detection based on statistics of assembly instructions using classification algorithms. In: Proceedings of 2015 18th CSI International Symposium on Computer Architecture and Digital Systems, pp. 1–6.
Kwon, B.J., Mondal, J., Jang, J., Bilge, L., Dumitraş, T., 2015. The dropper effect: Insights into malware distribution with downloader graph analytics. In: Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security. In: CCS '15, ACM, New York, NY, USA, pp. 1118–1129.
LeDoux, C., Lakhotia, A., 2015. Malware and machine learning. In: Yager, R.R., Reformat, M.Z., Alajlan, N. (Eds.), Intelligent Methods for Cyber Warfare. Springer International Publishing, Cham, pp. 1–42.
Li, H., 2012. Dynamic analysis of Object-Oriented software complexity. In: Proceedings of 2012 2nd International Conference on Consumer Electronics, Communications and Networks, pp. 1791–1794.
Li, Z., Zou, D., Tang, J., Zhang, Z., Sun, M., Jin, H., 2019. A comparative study of deep learning-based vulnerability detection system. IEEE Access 7, 103184–103197.
Li, Z., Zou, D., Xu, S., Jin, H., Zhu, Y., Chen, Z., Wang, S., Wang, J., 2018a. Sysevr: A framework for using deep learning to detect software vulnerabilities. CoRR abs/1807.06756 arXiv:1807.06756.
Li, Z., Zou, D., Xu, S., Ou, X., Jin, H., Wang, S., Deng, Z., Zhong, Y., 2018b. Vuldeepecker: A deep learning-based system for vulnerability detection. ArXiv Preprint ArXiv:1801.01681.
Liu, W., Liu, S., Gu, Q., Chen, J., Chen, X., Chen, D., 2015. Empirical studies of a two-stage data preprocessing approach for software fault prediction. IEEE Trans. Reliab. 65 (1), 38–53.
Malik, Y., Campos, C.R.S., Jaafar, F., 2019. Detecting android security vulnerabilities using machine learning and system calls analysis. In: Proceedings of 2019 IEEE 19th International Conference on Software Quality, Reliability and Security Companion, pp. 109–113.
Mohaisen, A., Alrawi, O., Mohaisen, M., 2015. AMAL: High-fidelity, behavior-based automated malware analysis and classification. Comput. Secur. 52, 251–266.
Ndichu, S., Kim, S., Ozawa, S., Misu, T., Makishima, K., 2019. A machine learning approach to detection of javascript-based attacks using ast features and paragraph vectors. Appl. Soft Comput. 84, 105721. http://dx.doi.org/10.1016/j.asoc.2019.105721.
Ni, C., Chen, X., Xia, X., Gu, Q., Zhao, Y., 2019. Multi-task defect prediction. J. Softw.: Evol. Process.
Ni, C., Liu, W.-S., Chen, X., Gu, Q., Chen, D.-X., Huang, Q.-G., 2017. A cluster based feature selection method for cross-project software defect prediction. J. Comput. Sci. Tech. 32 (6), 1090–1107.
NVD, 0000. https://nvd.nist.gov/.
Raff, E., Nicholas, C., 2017. An alternative to NCD for large sequences, lempel-ziv jaccard distance. In: Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining. In: KDD '17, ACM, New York, NY, USA, pp. 1007–1015.
Saccente, N., Dehlinger, J., Deng, L., Chakraborty, S., Xiong, Y., 2019. Project achilles: A prototype tool for static method-level vulnerability detection of java source code using a recurrent neural network. In: 2019 34th IEEE/ACM International Conference on Automated Software Engineering Workshop (ASEW). pp. 114–121.
Sexton, J., Storlie, C., Anderson, B., 2016. Subroutine based detection of apt malware. J. Comput. Virol. Hacking Tech. http://dx.doi.org/10.1007/s11416-015-0258-7.
Shabtai, A., Moskovitch, R., Elovici, Y., Glezer, C., 2009. Detection of malicious code by applying machine learning classifiers on static features: A state-of-the-art survey. Inf. Secur. Tech. Rep. (ISSN: 1363-4127) 14 (1), 16–29, Malware.
Software Assurance Reference Dataset, 0000. https://samate.nist.gov/SRD/index.php.
Souri, A., Hosseini, R., 2018. A state-of-the-art survey of malware detection approaches using data mining techniques. Hum.-centric Comput. Inf. Sci. 8 (1), 1–22.
Tommy, R., Sundeep, G., Jose, H., 2017. Automatic detection and correction of vulnerabilities using machine learning. In: Proceedings of 2017 International Conference on Current Trends in Computer, Electrical, Electronics and Communication, pp. 1062–1065.
Wan, Z., Xia, X., Lo, D., Murphy, G.C., 2019. How does machine learning change software development practices?. IEEE Trans. Softw. Eng. 1.
Wolf, L., Hanani, Y., Bar, K., Dershowitz, N., 2014. Joint word2vec networks for bilingual semantic representations. Int. J. Comput. Linguist. Appl. 5 (1), 27–42.
Word2vec, 0000. http://radimrehurek.com/gensim/models/word2vec.html.
Xu, Z., Li, S., Xu, J., Liu, J., Luo, X., Zhang, Y., Zhang, T., Keung, J., Tang, Y., 2019. Ldfr: Learning deep feature representation for software defect prediction. J. Syst. Softw. 158, 110402. http://dx.doi.org/10.1016/j.jss.2019.110402, URL http://www.sciencedirect.com/science/article/pii/S0164121219301761.
Xu, Z., Liu, J., Luo, X., Yang, Z., Zhang, Y., Yuan, P., Tang, Y., Zhang, T., 2018. Software defect prediction based on kernel PCA and weighted extreme learning machine. Inf. Softw. Technol. 106, http://dx.doi.org/10.1016/j.infsof.2018.10.004.
Xu, Z., Zhang, T., Zhang, Y., Tang, Y., Liu, J., Luo, X., Keung, J., Cui, X., 2019. Identifying crashing fault residence based on cross project model. In: 2019 IEEE 30th International Symposium on Software Reliability Engineering (ISSRE). pp. 183–194.
Zheng, W., Bai, Y., Che, H., 2018. A computer-assisted instructional method based on machine learning in software testing class. Comput. Appl. Eng. Educ. 26 (5), 1150–1158.
Zheng, W., Feng, C., Yu, T., Yang, X., Wu, X., 2019. Towards understanding bugs in an open source cloud management stack: An empirical study of openstack software bugs. J. Syst. Softw. 151, 210–223.