Received June 5, 2019, accepted July 13, 2019, date of publication July 23, 2019, date of current version August 12, 2019.

Digital Object Identifier 10.1109/ACCESS.2019.2930578

A Comparative Study of Deep Learning-Based Vulnerability Detection System

ZHEN LI[1,2,4], DEQING ZOU[1,3,5], JING TANG[1,2], ZHIHAO ZHANG[2], MINGQIAN SUN[2], AND HAI JIN[1,2], (Fellow, IEEE)

[1]National Engineering Research Center for Big Data Technology and System, Services Computing Technology and System Lab, Cluster and Grid Computing Lab, Big Data Security Engineering Research Center, Huazhong University of Science and Technology, Wuhan 430074, China
[2]School of Computer Science and Technology, Huazhong University of Science and Technology, Wuhan 430074, China
[3]School of Cyberspace Security, Huazhong University of Science and Technology, Wuhan 430074, China
[4]School of Cyber Security and Computer, Hebei University, Baoding 071002, China
[5]Shenzhen Huazhong University of Science and Technology Research Institute, Shenzhen 518057, China

Corresponding author: Deqing Zou (deqingzou@hust.edu.cn)

This wok was supported in part by the National Key Research and Development (R&D) Plan of China under Grant 2017YFB0802205, in part by the National Natural Science Foundation of China under Grant 61672249 and Grant 61802106, and in part by the Shenzhen Fundamental Research Program under Grant JCYJ20170413114215614.

ABSTRACT Source code static analysis has been widely used to detect vulnerabilities in the development of software products. The vulnerability patterns purely based on human experts are laborious and error prone, which has motivated the use of machine learning for vulnerability detection. In order to relieve human experts of defining vulnerability rules or features, a recent study shows the feasibility of leveraging deep learning to detect vulnerabilities automatically. However, the impact of different factors on the effectiveness of vulnerability detection is unknown. In this paper, we collect two datasets from the programs involving 126 types of vulnerabilities, on which we conduct the first comparative study to quantitatively evaluate the impact of different factors on the effectiveness of vulnerability detection. The experimental results show that accommodating control dependency can increase the overall effectiveness of vulnerability detection F1-measure by 20.3%; the imbalanced data processing methods are not effective for the dataset we create; bidirectional *recurrent neural networks* (RNNs) are more effective than unidirectional RNNs and convolutional neural network, which in turn are more effective than multi-layer perception; using the last output corresponding to the time step for the *bidirectional long short-term memory* (BLSTM) can reduce the false negative rate by 2.0% at the price of increasing the false positive rate by 0.5%.

INDEX TERMS Vulnerability detection, deep learning, source code, comparative study.

I. INTRODUCTION

Nowadays, software vulnerabilities have led to many cyber-attacks. Although numerous approaches have been proposed for software vulnerability detection, the number of vulnerabilities reported in the *Common Vulnerabilities and Exposures* (CVE) [1] increases each year. The problem of software vulnerability prevalence will still exist for a long time.

Source code static analysis has been widely used to detect vulnerabilities in the development of software products. There are mainly two types of approaches: code similarity-based approaches and pattern-based approaches. *Code similarity-based* approaches are limited to detecting vulnerabilities caused by code clone [2], [3], while *pattern-based* approaches are more general and widely used to detect vulnerabilities with various causes. For *pattern-based* approaches, vulnerabilities are detected through matching vulnerability patterns which can be generated by human experts or machine learning techniques. This types of approaches involve three categories: rule-based methods, traditional machine learning-based methods, and deep learning-based methods. Rule-based methods [4]–[8] and traditional machine learning-based methods [9]–[12] typically require human experts to define rules or features to generate vulnerability patterns. As a result, they need many manual efforts and are difficult to characterize the vulnerabilities accurately, thus achieve high false positives or high false negatives. Deep learning-based methods [13]–[17], which do not need human experts to define features and can learn

The associate editor coordinating the review of this manuscript and approving it for publication was Hong-Mei Zhang.

vulnerability patterns automatically, have become a new trend in software vulnerability detection.

The recently proposed VulDeePecker [16] is the first to use deep learning to detect vulnerabilities at the slice level (i.e., multiple lines of code that are semantically related to each other in terms of e.g., data dependency or control dependency), while noting that other studies on using deep learning for vulnerability detection are at a coarser granularity (e.g., function level) [13]–[15]. VulDeePecker demonstrates the feasibility of using deep learning to detect vulnerabilities in a finer granularity, while the quantitative impact of different factors on the effectiveness of vulnerability detection is unknown, such as the following:

- VulDeePecker adopts data dependency as the semantic information of programs. This makes one wonder whether or not other semantic information (e.g., control dependency) can improve the effectiveness of vulnerability detection.
- VulDeePecker does not involve any imbalanced data processing, although the number of vulnerable samples is much smaller than the number of samples without vulnerabilities. This makes one wonder whether or not the imbalanced data processing can improve the effectiveness of vulnerability detection.
- VulDeePecker uses the *Bidirectional Long Short-Term Memory* (BLSTM) neural network. This leaves one wonder whether or not other neural networks can improve the effectiveness of vulnerability detection.

These questions inspire us to make a comparative study while answering these questions as a piggyback.

Our Contributions: In this paper, we conduct the first comparative study to evaluate the quantitative impact of different factors on the effectiveness of deep learning-based vulnerability detection. Specifically, the main contributions of this paper are as follows.

First, in order to experimentally show the effectiveness of vulnerability detection, we collect two datasets from the programs involving 126 types of vulnerabilities, while noting that the dataset published by [16] only involves two types of vulnerabilities (i.e., buffer error vulnerabilities and resource management error vulnerabilities) and only accommodates data dependency as the semantic information. One dataset contains 68,353 code gadgets (i.e., a number of statements that are semantically related to each other) with data dependency and control dependency and the other dataset contains 98,262 code gadgets with only data dependency. We have made the datasets publicly available at https://github.com/VulDeePecker/Comparative_Study so that other researchers can use them for their own studies.

Second, we evaluate the quantitative impact of different factors on the effectiveness of vulnerability detection on the datasets. Some of the experimental findings are highlighted as follows: (i) Accommodating control dependency can increase the overall effectiveness of vulnerability detection F1-measure by 20.3%. (ii) The imbalanced data processing methods are not effective for the dataset we create, and the over-sampling method (e.g., SMOTE) is better than other imbalanced data processing methods for the dataset. (iii) Bidirectional *Recurrent Neural Networks* (RNNs) are more effective than unidirectional RNNs and convolutional neural network, which in turn are more effective than multi-layer perception. (iv) Using the last output corresponding to the time step for the BLSTM can reduce the false negative rate by 2.0% at the price of increasing the false positive rate by 0.5%.

Third, we implement the deep learning-based vulnerability detection system based on an extended open source parser *Joern* [18] for the comparative study, while noting that VulDeePecker [16] is implemented based on the commercial tool Checkmarx [5] which cannot accommodate new semantic information of programs. In addition, we identify important code elements in the code gadgets for vulnerability detection, which can help understand what features the deep learning model has automatically learned.

Paper Organization: Section II describes the preliminaries, Section III presents the comparative study methodology. Section IV discusses the experiments and results. Section V reviews the related prior work. Section VI concludes the present paper and discusses the future work.

II. PRELIMINARIES

Deep learning techniques are recently studied and used to detect software vulnerabilities in source code [13]–[17]. VulDeePecker [16] is the first to use deep learning to detect software vulnerabilities at the slice level, a finer granularity than the function level. In this section, we give a brief review on VulDeePecker, then introduce the BLSTM neural network, and finally compare our study with VulDeePecker.

A. A BRIEF REVIEW ON VULDEEPECKER

1) CODE GADGET

A *code gadget* consists of a number of (not necessarily consecutive) program statements which are related to each other semantically [16]. Specifically, for a vulnerability related to library/API function call, the code gadget is composed of the program statements semantically associated to the arguments of library/API function call. The semantic information can accommodate data dependency and control dependency which are widely used for vulnerability detection [18]–[20]. The present design of VulDeePecker only accommodates data dependency in the code gadgets.

2) DEEP LEARNING-BASED VULNERABILITY DETECTION

VulDeePecker [16] mainly focuses on detecting vulnerabilities related to library/API function calls. It has two phases: a learning phase and a detection phase. In the *learning phase*, the input is plenty of training programs, some of which are vulnerable and others are not. The output is vulnerability patterns which are coded into a neural network. VulDeePecker takes advantage of the commercial tool Checkmarx [5] to collect code gadgets, each of which consists of a number of

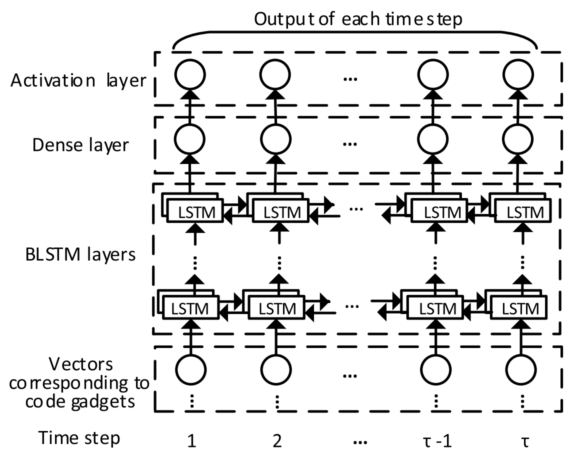


FIGURE 1. Structure of BLSTM.

program statements that are data-dependent on a library/API function call. These code gadgets are labeled vulnerable or not, and then transformed into vectors as the input to the BLSTM. Finally, the vulnerability patterns, represented as the BLSTM with fine-tuned parameters, are as the output of the learning phase.

In the *detection phase*, the input is target programs and the trained neural network from the learning phase, and the output is vulnerable code gadgets. The code gadgets related to the library/API function calls are first collected from target programs based on the commercial tool Checkmarx and transformed into vectors. Then VulDeePecker uses the learned vulnerability patterns to determine whether the code gadgets from the target programs are vulnerable or not and if so, output the vulnerable code gadgets.

B. BLSTM

BLSTM, as a kind of widely used bidirectional recurrent neural network, is effective in coping with sequential data involving context. Figure 1 shows the structure of the BLSTM whose input is the vectors corresponding to code gadgets. The BLSTM consists of several BLSTM layers, a dense layer, and an activation layer. The BLSTM layers have forward and backward directions and contain *Long Short-Term Memory* (LSTM) cells; the dense layer reduces the number of dimensions of the vectors; and the activation layer uses an activation function to generate the output corresponding to each time step. The output of each time step (e.g., the last time step) can be selected in the back propagation for parameter tuning. Finally, a BLSTM with fine-tuned parameters is output after training.

C. COMPARISON WITH VULDEEPECKER

We stress that this paper is not a simple incremental work over VulDeePecker [16] for the following four reasons: (i) Though VulDeePecker demonstrates the feasibility of using deep learning to detect vulnerabilities in a finer granularity, the impact of different factors on the effectiveness of vulnerability detection is unknown. In this paper, we make a comparative study which corresponds to three steps of deep learning-based vulnerability detection system (see Section III-A for details). (ii) In order to evaluate the effectiveness, we collect two datasets from the programs involving 126 types of vulnerabilities, because the dataset published by [16] only involves two types of vulnerabilities and only accommodates data dependency as the semantic information. (iii) Our system is a completely new implementation using an extended open source parser *Joern* [18], because a straightforward extension of VulDeePecker, which is based on the commercial tool Checkmarx [5], cannot accommodate new semantic information in the code gadgets. (iv) We identify important code elements in the code gadgets for vulnerability detection, which can be used to speculate what features the deep learning model has automatically learned.

III. COMPARATIVE STUDY METHODOLOGY

Our goal is to evaluate the quantitative impact of different factors on the effectiveness of deep learning-based vulnerability detection. We focus on vulnerabilities related to library/API function calls in C/C++ programs, while leaving the extension to accommodating other vulnerabilities to future work. In this section, we describe the design of deep learning-based vulnerability detection system to achieve the goal.

A. OVERVIEW

As highlighted in Figure 2, the input to the deep learning-based vulnerability detection system is the source code of a large number of training programs and some target programs, and the output is vulnerable code gadgets. The process of deep learning-based vulnerability detection has six steps: generating code gadgets (Step I), generating ground truth labels for code gadgets (Step II), transforming code gadgets into vectors (Step III), data processing with imbalanced techniques (Step IV), training a neural network (Step V), and applying the trained neural network to classify the code gadgets (Step VI). Among these steps, Steps II, III, and VI are standard, so there are no important factors that can be chosen for comparative study. In this paper, we make a comparative study from the following three aspects which correspond to Steps I, IV, and V highlighted with bold border in Figure 2: generating code gadgets with different semantic information, data processing with different imbalanced techniques, and training a neural network with hard negative mining.

There are two phases: a learning (i.e., training) phase and a detection phase. In the learning phase, the input is the source code of training programs which is used to train the deep neural network for vulnerability detection. Specifically, the learning phase has the following 5 steps.

- Step I: generating code gadgets. Considering code gadgets in which the statements are related to each other only by control dependency involves little information about vulnerabilities, this step generates two types of code gadgets: code gadgets with data dependency and

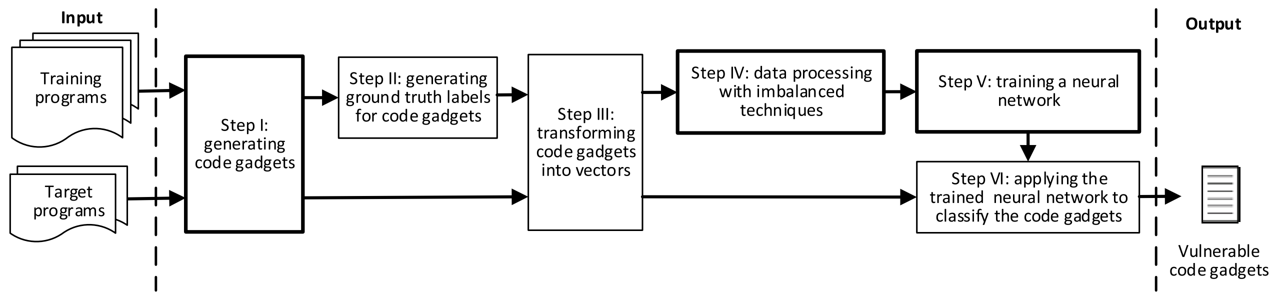

Input
Training programs
Target programs
Step I: generating code gadgets
Step II: generating ground truth labels for code gadgets
Step III: transforming code gadgets into vectors
Step IV: data processing with imbalanced techniques
Step V: training a neural network
Step VI: applying the trained neural network to classify the code gadgets
Output
Vulnerable code gadgets

FIGURE 2. **Overview of deep learning-based vulnerability detection system: Steps I, IV, and V which involve the comparative study are highlighted with bold border.**

code gadgets with both data dependency and control dependency. The process of this step will be elaborated in Section III-B.

- Step II: generating ground truth labels for code gadgets. According to the data sources of known vulnerabilities, such as the *National Vulnerability Database* (NVD) [21] and the *Software Assurance Reference Dataset* (SARD) [22], this step labels each vulnerable code gadget as "1", and labels each code gadget which is not vulnerable as "0". Specifically, the code gadgets are labeled automatically as follows: if the code gadget contains at least one vulnerable statement in the program in question, it is labeled as "1"; otherwise, it is labeled as "0". Note that for the code gadgets which are mislabeled with high probability (see Section III-D for details), we manually check them and correct the mislabeled ones in Step V.
- Step III: transforming code gadgets into vectors. The code gadgets have to be encoded into vectors for input to deep neural networks. This step maps the variable names and the user-defined function names to symbolic names to generate the symbolic representation (e.g., "strcpy(dest, source);" is mapped to "strcpy(V1, V2);"), divides the symbolic representation into a sequence of symbols (e.g., "strcpy", "(", "V1", ",", "V2", ")", and ";"), then transforms each symbol into a fixed-length vector, and finally obtains a vector for each symbolic representation by concatenating the vectors of symbols.
- Step IV: data processing with imbalanced techniques. This step adopts no imbalanced techniques and several imbalanced techniques respectively to process the vectors obtained from Step III. The process of this step will be elaborated in Section III-C.
- Step V: training a neural network. This step first uses k-fold cross validation [23] to identify the code gadgets which are mislabeled with high probability, check them manually, and correct the mislabeled ones. Then the vectors corresponding to code gadgets from the training programs and their labels are input to the neural network. This step leverages hard negative mining for imbalanced data processing, six neural networks, and two strategies of outputs for recurrent neural networks. The trained deep neural network with fine-tuned parameters is finally obtained. The process of this step will be elaborated in Section III-D.

In the detection phase, the input is the source code of target programs which goes through Steps I and III (the same as the steps in the learning phase), then uses the following step to detect the vulnerable code gadgets in target programs.

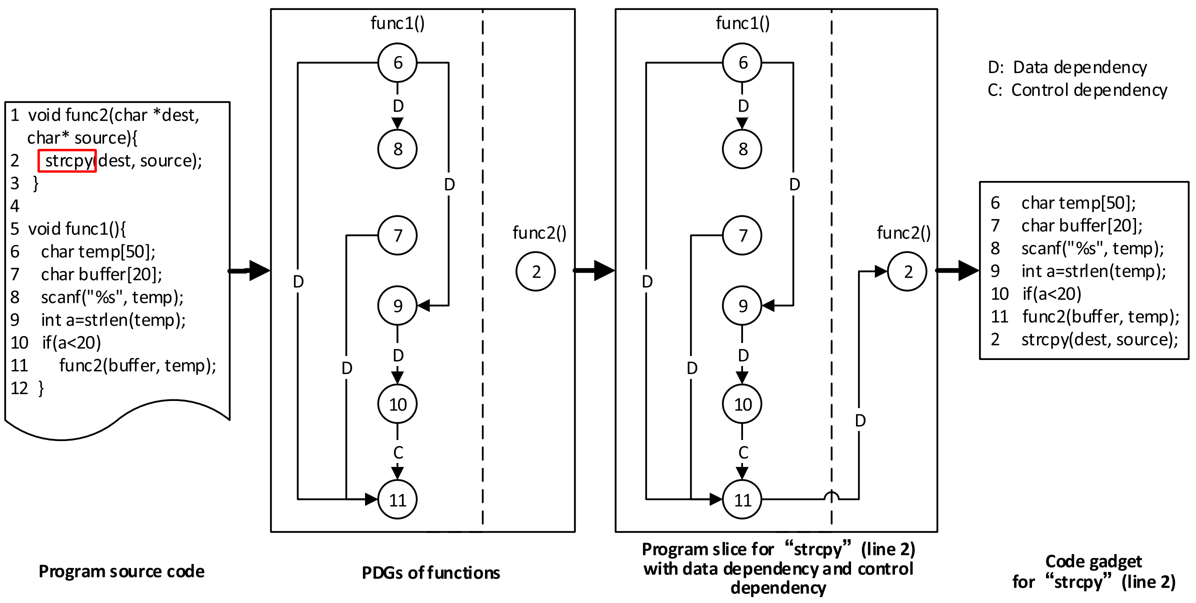
- Step VI: applying the trained neural network to classify the code gadgets from target programs. For the vectors corresponding to the code gadgets from target programs, this step uses the trained neural network from Step V in the learning phase to detect vulnerabilities in target programs, identifies important code elements for vulnerability detection, and outputs the vulnerable code gadgets. The process of this step will be elaborated in Section III-E.

In what follows, we respectively elaborate Steps I, IV, and V which involve the comparative study, and Step VI which involves identifying important code elements for vulnerability detection. For the details of Steps II and III, please refer to [16].

B. GENERATING CODE GADGETS

Source code involves much semantic information among which data dependency and control dependency are widely used for vulnerability detection [18]–[20]. We generate the code gadgets with different semantic information based on the open source parser *Joern* [18]. We do not adopt the commercial tool Checkmarx [5] to generate code gadgets as VulDeePecker [16] does, because Checkmarx can be used to obtain the code gadgets with only data dependency.

We select 811 C/C++ library/API function calls related to security according to all kinds of known vulnerabilities. The list of these security-related C/C++ library/API function calls is deferred to Table 9 in Appendix VI-A. In what follows, library/API function calls refer to these 811 security-related C/C++ library/API function calls. We first use *Joern* [18] to extract the abstract syntax tree for each function in programs, and traverse the abstract syntax trees to identify all library/API function calls. Then we generate a code gadget corresponding to each library/API function call. There are mainly three steps.



```
1  void func2(char *dest,
     char* source){
2     strcpy(dest, source);
3  }
4
5  void func1(){
6    char temp[50];
7    char buffer[20];
8    scanf("%s", temp);
9    int a=strlen(temp);
10   if(a<20)
11      func2(buffer, temp);
12 }
```

```
6   char temp[50];
7   char buffer[20];
8   scanf("%s", temp);
9   int a=strlen(temp);
10  if(a<20)
11  func2(buffer, temp);
2   strcpy(dest, source);
```

func1()
func2()
D
C
6
7
8
9
10
11
2
D: Data dependency
C: Control dependency
Program source code
PDGs of functions
Program slice for "strcpy" (line 2) with data dependency and control dependency
Code gadget for "strcpy" (line 2)

FIGURE 3. **Generating code gadgets with data dependency and control dependency for library/API function call *strcpy* (line 2).**

First, we generate a *Program Dependence Graph* (PDG) for each function using open source tool *Joern* [18]. In the PDG, each node represents a statement or a control predicate, and each edge represents a data dependency or a control dependency between two nodes. As an example shown in Figure 3, the second column illustrates the PDGs of functions *func*1 and *func*2, where the numbers in the nodes represent the corresponding line numbers of statements.

Second, we generate the program slice for each library/API function call based on the PDGs. Specifically, the program slice is composed of an interprocedural forward slice and an interprocedural backward slice which are merged at the library/API function call. For a library/API function call, the forward slice in a function involves the statements from all paths in the PDG starting at the library/API function call; the backward slice in a function involves the statements from all paths in the PDG ending at the library/API function call. We can obtain the forward slice and the backward slice in a function using *Joern* [18]. In order to generate the interprocedural slice (i.e., slice that can cross function boundaries), we extend *Joern* to generate the interprocedural forward slice and the interprocedural backward slice by going beyond function boundaries which are caused by user-defined function calls.

In order to show the impact of different semantic information for code gadgets on the effectiveness of vulnerability detection, we adopt two types of program slices based on PDGs, which are then used to generate the corresponding two types of code gadgets in the third step.

- **Data dependency**: This type of program slices involves the interprocedural forward slice and the interprocedural backward slice with only data dependency in the PDGs.
- **Data dependency and control dependency**: This type of program slices involves the interprocedural forward slice with only data dependency, and the interprocedural backward slice with both data dependency and control dependency in the PDGs.

It is worth mentioning that control dependency does not involve in the interprocedural forward slice for the second type of program slices, because the statements, which are affected by the library/API function calls only according to control dependency, are not vulnerable. As an example shown in Figure 3, the third column illustrates the program slice for library function call "*strcpy*" (line 2) with data dependency and control dependency. Because function *func*1 calls function *func*2 at line 11, an edge between line 11 and line 2 is added, which represents the data dependency between the parameters of function *func*1 and function *func*2 in the interprocedural slice.

Third, we transform program slices to code gadgets. For the statements from the same function in a program slice, the order of the statements is preserved. As an example shown in Figure 3, lines {6,7,8,9,10,11} and line {2} are obtained. For the statements from different functions in a program slice, the statements in the calling functions appear before the statements in the called functions. The final code gadget obtained in Figure 3 is lines {6,7,8,9,10,11,2}.

C. DATA PROCESSING WITH IMBALANCED TECHNIQUES

Since the number of vulnerable code gadgets is much smaller than the number of code gadgets without vulnerabilities, we can process the imbalanced data for vulnerability detection. There are mainly two types of approaches used to reduce the influence of imbalance: *changing the distribution of dataset* [24], [25] and *leveraging hard negative mining* [26]. Because leveraging hard negative mining is applied to the neural network for training, we describe its design in Section III-D.

1) CHANGING THE DISTRIBUTION OF DATASET

By increasing or decreasing some code gadgets (i.e., samples) reasonably to balance the dataset, the adverse effects caused by imbalanced data can be reduced. The main techniques include random data sampling, distance methods, data cleaning approximation, clustering algorithms, evolutionary algorithms, and so on. Among them, the data sampling is the most widely used technique, and it mainly includes under-sampling and over-sampling. *Under-sampling* is mainly used for majority samples to remove noise and redundant data, and *over-sampling* is primarily intended to add the number of minority samples. We choose the following methods to preprocess the input vectors of neural network.

- **None**. No imbalanced data processing techniques are used.
- **NearMiss-2**. NearMiss-2 is an under-sampling method using a *K-Nearest Neighbor* (KNN) classifier [24].
- **SMOTE**. SMOTE is an over-sampling technique, and generates new synthetic samples by interpolation [25].

D. TRAINING A NEURAL NETWORK

Because the labels of code gadgets which are automatically generated in Step II may be mislabeled, in this step we use k-fold cross validation [23] to identify the code gadgets which are mislabeled with high probability and check them manually as per the following steps. First, the data set of code gadgets is divided into k subsets. Second, one of the k subsets is used as the validation set and the other k-1 subsets are put together to form a training set. Third, the trained neural network is used to classify the code gadgets in the validation set. The false positives (i.e., the samples which are not vulnerable and are detected as vulnerable) and false negatives (i.e., the vulnerable samples which are not detected as vulnerable) are considered as the code gadgets which are mislabeled with high probability. We manually check them and correct the mislabeled code gadgets. The second and third steps are repeated k times so that each subset is used as the validation set once.

After correcting the mislabeled code gadgets, we input the vectors corresponding to code gadgets from the training programs and their labels to the neural network. In what follows, we first provide another type of approaches to imbalanced data processing for the neural network (i.e., leveraging hard negative mining), then discuss different kinds of neural networks and the selection strategies of outputs for recurrent neural networks.

1) LEVERAGING HARD NEGATIVE MINING

For highly imbalanced dataset, hard negative mining, also known as bootstrapping, is a common technique introduced in the 1990s by Sung [27] when training face detection models. The basic idea is to gradually increase the background examples collection by selecting the examples that are more likely to cause false positives. This idea is then applied

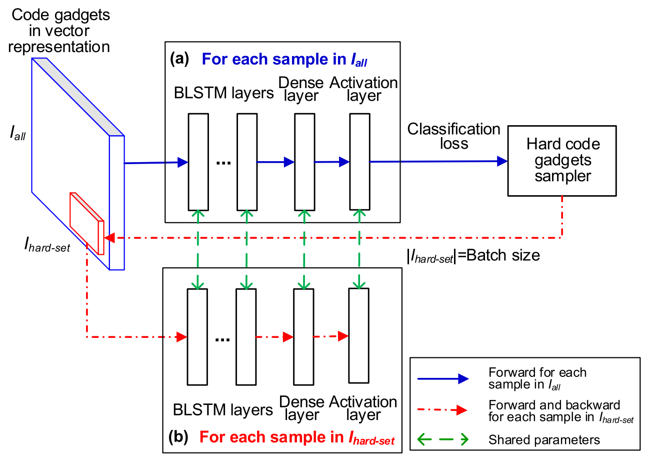

Code gadgets in vector representation
I_{all}
$I_{hard\text{-}set}$
(a) For each sample in I_{all}
BLSTM layers
Dense layer
Activation layer
Classification loss
Hard code gadgets sampler
$|I_{hard\text{-}set}|$=Batch size
BLSTM layers
Dense layer
Activation layer
(b) For each sample in $I_{hard\text{-}set}$
Forward for each sample in I_{all}
Forward and backward for each sample in $I_{hard\text{-}set}$
Shared parameters

FIGURE 4. BLSTM neural network architecture for imbalanced data. In (a), a read-only model is used for forward process for all code gadgets. In (b), a set of hard samples, obtained by hard code gadgets sampler, is used for both forward and backward passes. The adjusted model in (b) is updated to the model in (a) simultaneously.

to the *Online Hard Example Mining* (OHEM) algorithm in an imbalanced dataset for region-based convolution neural network detector in the computer vision [26]. We apply the OHEM algorithm [26] to neural networks for vulnerability detection. In what follows, we take the BLSTM as an example to show the process.

Each time step in the BLSTM corresponds to a symbol in code gadgets obtained in Step III. Figure 4 (a) indicates a BLSTM consisting of several BLSTM layers, a dense layer, and an activation layer, which is a read-only model that does not contain a reverse tuning procedure and is used for the forward process for a set of all code gadgets I_{all}. After the forward process for all code gadgets, the online hard sample mining is used to calculate the loss between the prediction value and the true value. The samples whose loss values are greater than the loss threshold are seen as hard samples.

Considering that if we use all the samples as the input of (a) in Figure 4, the non-hard samples, also called easy samples, will participate in backward pass as well and require memory allocation even if the losses of those easy samples are set to 0. In order to reduce memory consumption and improve the efficiency of computing, we replicate the network directly and only use hard samples for the backward pass training. As can be seen in Figure 4, (b) is a copy of (a), and what makes a difference between them is that the input of (b) is a set of hard samples $I_{hard-set}$ which is used for both forward and backward passes. The samples after the re-sampling phase are the input of (b) to train the model.

2) DIFFERENT NEURAL NETWORKS

We choose three widely-used types of neural networks to detect vulnerabilities: multi-layer perceptions, convolutional neural networks, and recurrent neural networks. They involve six neural networks: a *Multi-Layer Perception* (MLP), a *Convolutional Neural Network* (CNN), an LSTM, a *Gated Recurrent Unit* (GRU), a BLSTM, and a *Bidirectional Gated Recurrent Unit* (BGRU). Among them, LSTM and GRU are unidirectional RNNs; BLSTM and BGRU are *Bidirectional RNNs* (BRNNs).

3) DIFFERENT SELECTION STRATEGIES OF OUTPUTS FOR RNNs

For RNNs, the output of each time step in the activation layer can be selected in the back propagation for parameter tuning. At time step t ($1 \leq t \leq \tau$), the output corresponding to t for the code gadget $\boldsymbol{x}$ is $h_t(\boldsymbol{x})$. We adopt two selection strategies of outputs to show the impact of selection strategies of outputs corresponding to time steps on the effectiveness of deep learning-based vulnerability detection.

- **Last time step**: This strategy uses the output corresponding to the last time step in the activation layer, i.e., $h_\tau(\boldsymbol{x})$.
- **Average of l-max**: This strategy uses the average of l largest outputs corresponding to time steps in the activation layer, i.e.,

$$ave(max_l(h_1(\boldsymbol{x}), h_2(\boldsymbol{x}), \ldots, h_\tau(\boldsymbol{x}))) \quad (1)$$

where function max_l indicates the l largest values among the elements, and function *ave* indicates the average of the elements.

E. APPLYING THE TRAINED NEURAL NETWORK TO CLASSIFY THE CODE GADGETS

In the detection phase, we use the trained neural network to detect whether each code gadget from target programs is vulnerable or not. For RNNs, we can obtain the output corresponding to each symbol in a code gadget. The larger the output value corresponding to a symbol in the code gadget is, the more likely the symbol is vulnerable. The impact of each symbol in the vulnerable code gadgets on the classification can be illustrated by *hot maps*. For each code gadget, the larger output value corresponding to the symbol is represented by darker color, which means the greater probability of vulnerability. In order to explore which code elements (i.e., one or multiple symbols) are more likely to be vulnerable and highlight their usefulness semantically, we manually examined the code elements with darker color in 500 vulnerable code gadgets which were randomly chosen. We found that most of them involved the library/API function calls, the types, the control conditions, and so on. This can be explained by the fact that the misuse of library/API function call names and their parameters, the type conversions of variables, the return types of functions, `if` conditions, `while` conditions, and `for` conditions often lead to vulnerabilities. From the hot maps, we can speculate what features the deep learning model has automatically learned, and further help to explain the deep learning models, which is an interesting future work.

IV. EXPERIMENTS AND RESULTS

According to the vulnerabilities related to library/API function calls, our experiments are geared towards answering the following three Research Questions (RQs).

- RQ1: Can accommodating more semantic information in the code gadgets enhance the effectiveness of vulnerability detection?
- RQ2: Can imbalanced data processing methods enhance the effectiveness of vulnerability detection?
- RQ3: Can different kinds of deep neural networks impact the effectiveness of vulnerability detection?

We implement the neural networks in Python using TensorFlow [28]. The computer running experiments has a NVIDIA GeForce GTX 1080 GPU and an Intel Xeon E5-1620 CPU operating at 3.50GHz.

A. EVALUATION METRICS

The effectiveness of vulnerability detection can be evaluated by 5 widely used metrics [29]:

- *False Positive Rate* (FPR): The ratio of false-positive samples to the total samples that are not vulnerable, where false-positive samples are the samples which are not vulnerable and are detected as vulnerable.
- *False Negative Rate* (FNR): The ratio of false-negative samples to the total samples that are vulnerable, where false-negative samples are the vulnerable samples which are not detected as vulnerable.
- *Accuracy* (A): The ratio of true-positive and true-negative samples to the total samples, where true-positive samples are the vulnerable samples which are detected as vulnerable, and true-negative samples are the samples which are not vulnerable and are not detected as vulnerable.
- *Precision* (P): The ratio of true-positive samples to the total samples that are detected as vulnerable.
- *F1-measure* (F1): The overall effectiveness that considers both precision and false negative rate. $F1 = \frac{2 \cdot P \cdot (1-FNR)}{P+(1-FNR)}$.

The values of above five metrics vary from 0 to 1. For FPR and FNR, the closer they are to 0, the better; for A, P and F1, the closer they are to 1, the better.

B. PREPARING THE DATASET

The programs we use are from two widely used vulnerability data sources: NVD [21] and SARD [22], where NVD reports the vulnerabilities in software products, and SARD involves large numbers of programs (i.e., test cases) with production, synthetic, and academic security flaws or vulnerabilities. In the NVD, for each vulnerability of open source products, the source code of vulnerable program and its corresponding patched program can be obtained by means of *diff* file, which describes the difference between the vulnerable pieces of code and their patched versions [30]. In the SARD, there are three types of programs: "good" programs which have no vulnerabilities, "bad" programs which have vulnerabilities,

TABLE 1. Datasets for experiments: a dataset of code gadgets with data dependency and control dependency (DDCD dataset for short) and a dataset of code gadgets with data dependency (DD dataset for short).

Dataset	#Code gadgets	#Vulnerable code gadgets	#Not vulnerable code gadgets
DDCD	68,353	13,686	54,667
DD	98,262	15,110	83,152

and "mixed" programs which have vulnerabilities and their patched versions.

We focus on 19 popular C/C++ open source products as VulDeePecker [16] selects, and collect all kinds of vulnerabilities which diff files are available in these products. We also collect C/C++ programs in the SARD with all kinds of vulnerabilities. Note that we do not use the dataset of VulDeePecker [16], because (i) the types of vulnerabilities are limited to two types of vulnerabilities (i.e., buffer error and resource management error) and (ii) the code gadgets generated by means of commercial tool Checkmarx [5] only involve data dependency. In total, we collect 368 open source programs related to CVEs from the NVD and 14,000 programs from the SARD. It is worth mentioning that a program in the NVD consists of one or several files which contain a vulnerability or its patched version and a program in the SARD is a test case. These programs contain 126 types of vulnerabilities, where each type is uniquely identified by a Common Weakness Enumeration IDentifier (CWE ID) [31]. The 126 CWE IDs are listed in Appendix VI-B. In our experiments, we randomly select 80% of programs from the NVD and the SARD we collect respectively as training programs, and the rest of 20% as target programs.

We generate code gadgets according to the methods proposed in Section III for both training programs and target programs, and delete the duplicated code gadgets. We obtain a dataset involving 68,353 different code gadgets with data dependency and control dependency (DDCD dataset for short), in which 55,334 code gadgets are generated from training programs and 13,019 code gadgets are generated from target programs; and we obtain a dataset involving 98,262 different code gadgets with data dependency (DD dataset for short) in which 78,558 code gadgets are generated from training programs and 19,704 code gadgets are generated from target programs. For the NVD, each statement that is deleted or modified according to the *diff* file is considered as the vulnerable statement. For the SARD, vulnerable statements are clearly marked in each vulnerable program. The number of vulnerable code gadgets and the number of code gadgets that are not vulnerable are shown in the third column and fourth column of Table 1 respectively. The datasets are publicly available at https://github.com/VulDeePecker/Comparative_Study.

C. EXPERIMENTAL RESULTS FOR RQ1

In order to answer whether accommodating more semantic information in the code gadgets can enhance the effectiveness of detecting vulnerabilities related to library/API function calls, we take control dependency as the additional semantic information and compare the effectiveness for the code gadgets with only data dependency and the code gadgets with both data dependency and control dependency. We use no imbalanced data processing and use BLSTM neural network with the last time step as the selection strategy of outputs as VulDeePecker [16] does.

TABLE 2. F1-measure with different values of dropout for BLSTM.

Dropout	FPR (%)	FNR (%)	A (%)	P (%)	F1 (%)
0.1	3.0	10.3	95.5	89.1	89.4
0.2	3.0	10.4	95.4	89.1	89.3
0.4	3.4	8.0	95.6	87.8	89.9
0.6	3.3	13.5	94.5	87.7	87.1
0.8	3.3	13.5	94.5	87.7	87.1

We use the cross validation to train the deep learning model to make sure its generalization. Selecting an appropriate value of k in the k-fold cross validation is an important research question, since it depends on the sample size, the number of parameters, the structure of data, etc [32]. Considering the tradeoff between the training overhead and the generalization of the model, we use a 5-fold cross validation to train a BLSTM and choose the values of hyper-parameters that lead to the highest F1-measure (i.e., the overall effectiveness of vulnerability detection). Specifically, we vary the value of a hyper-parameter and observe the impact on the F1-measure. When we adjust a hyper-parameter, we set other hyper-parameters to their default values or the values that are widely used by the deep learning community. The main hyper-parameters we use are as follows: the output dimension is 128, the batch size is 32, the minibatch stochastic gradient descent together with ADAMAX [33] is used, the learning rate is 0.01, the number of epochs is 4, the number of hidden layers is 1, the dimension of hidden vectors is 500, and the length of the vector corresponding to a symbol is 40. Table 2 shows the F1-measure with different values of dropout. We observe that the F1-measure reaches the maximum when dropout is 0.4 and declines with larger dropout. Therefore, the best value of dropout is 0.4. The other hyper-parameters of BLSTM are tuned in a similar fashion.

As illustrated in Table 3, using code gadgets with data dependency and control dependency outperforms using code gadgets with only data dependency in terms of most metrics. Specifically, compared with using code gadgets with data dependency, the FNR for using code gadgets with data dependency and control dependency is reduced by 31.2% at the price of increasing the FPR by only 0.5%. As a result, the overall effectiveness F1-measure for data dependency and control dependency increases 20.3%. This can be explained by the fact that control dependency carries extra information that can be used to distinguish the vulnerable code from the code that are not vulnerable, especially for the vulnerabilities whose patches involve condition statements or loop statements such as "if", "for", and "while". For example, if a vulnerable statement in an "if-else" structure is not data dependent on the "if" condition statement, then the "if"

TABLE 3. Effectiveness of BLSTM using different semantic information (with no imbalanced data processing and with the last time step as the selection strategy of output).

Type of semantic information	FPR (%)	FNR (%)	A (%)	P (%)	F1 (%)
Data dependency (VulDeePecker [16])	2.0	37.4	92.4	84.9	72.1
Data dependency and control dependency	2.5	6.2	96.7	90.9	92.4

condition statement is not involved in the code gadget with data dependency. However, the "if" condition statement as a context affects the vulnerable statement by control dependency. When the control dependency is used together with the data dependency, the effectiveness of vulnerability detection can be greatly improved. In what follows, we focus on using the code gadgets with data dependency and control dependency since it is more effective.

In summary, we answer RQ1 with the following:

Insight 1: Accommodating control dependency in the code gadgets can increase the overall effectiveness of vulnerability detection F1-measure by 20.3%.

D. EXPERIMENTAL RESULTS FOR RQ2

In order to test whether imbalanced data processing methods can enhance the effectiveness of deep learning-based vulnerability detection, we test three imbalanced data processing methods involving two methods of changing the distribution of dataset (i.e., NearMiss-2 and SMOTE) and a method of leveraging hard negative mining (i.e., OHEM). We test the BLSTM using code gadgets with data dependency and control dependency and use the last time step as the selection strategy of output.

Table 4 summarizes the comparison of results. Using imbalanced data processing methods does not outperform using no imbalanced data processing in most metrics on the dataset of code gadgets we create. Specifically, we make the following observations. First, using NearMiss-2 method is 1.4% lower in terms of FNR, 38.5% higher in terms of FPR, and 37.4% lower in terms of F1-measure, compared with no imbalanced data processing. This can be explained by the fact that the under-sampling method makes the number of majority samples (i.e., code gadgets that are not vulnerable) become less, thus greatly reduces the amount of data, which causes a markedly higher FPR than other methods. Second, using SMOTE method is 0.7% lower in terms of FNR, 0.6% higher in terms of FPR, and 0.7% lower in terms of F1-measure, compared with no imbalanced data processing. The over-sampling method achieves a little less overall effectiveness than using no imbalanced method by increasing the number of minority samples (i.e., vulnerable code gadgets). In spite of this, using SMOTE method achieves the best overall effectiveness among three imbalanced data processing methods due to the increased number of samples. Third, the FPR of using OHEM method is reduced by only 0.3% at the cost of a substantial increase of FNR

TABLE 4. Effectiveness of BLSTM using different imbalanced data processing methods (with the code gadgets involving data dependency and control dependency and with the last time step as the selection strategy of output).

Imbalanced data processing method	FPR (%)	FNR (%)	A (%)	P (%)	F1 (%)
None	2.5	6.2	96.7	90.9	92.4
NearMiss-2	41.0	4.8	66.7	38.6	55.0
SMOTE	3.1	5.5	96.3	89.0	91.7
OHEM	2.2	45.9	88.5	86.9	66.7
VulDeePecker [16]	2.0	37.4	92.4	84.9	72.1
SySeVR [17] with BGRU	2.2	6.8	96.8	91.9	92.5

(i.e., 39.7%) and a significant decline of F1-measure (i.e., 25.7%), compared with no imbalanced data processing. The hard negative mining method does not change the number of samples. It only selects the samples that are more likely to cause false positives in the back propagation for parameter tuning, which causes a lower FPR but markedly higher FNR than other methods. Fourth, for other deep learning-based methods, VulDeePecker [16] has much higher FNR which causes a much lower F1-measure than SMOTE and no imbalanced data processing methods for code gadgets with data dependency and control dependency, because VulDeePecker only uses data dependency as the semantic information in the code gadget. SySeVR [17] with BGRU is not significantly better than the BLSTM with no imbalanced data processing method, though SySeVR with BGRU has a little higher F1-measure (0.1%); their differences mainly caused by the different neural network.

Summarizing the preceding discussions, we draw:

Insight 2: The imbalanced data processing methods are not effective for the dataset of code gadgets we create using the method in Section IV-B, and the over-sampling method SMOTE is better than other imbalanced data processing methods.

E. EXPERIMENTAL RESULTS FOR RQ3

1) COMPARISON AMONG DIFFERENT NEURAL NETWORKS

In order to show the effectiveness of different neural networks, we adopt a 5-fold cross validation to train an MLP, a CNN, an LSTM, a GRU, a BLSTM, and a BGRU. We test the above six neural networks using code gadgets with data dependency and control dependency, no imbalanced data processing, and the last time step as the selection strategy of output for recurrent neural networks (as VulDeePecker [16] does). We choose the values of hyper-parameters that lead to the highest F1-measure for each neural network. The hyper-parameters we mainly tune are those which have a greater impact on the results according to the deep learning community. We use the minibatch stochastic gradient descent together with ADAMAX with the learning rate of 0.01, and set the number of epochs to 10. The final selected values of hyper-parameters that we tune are listed in Table 5. For other hyper-parameters, we choose their default values.

TABLE 5. Values of hyper-parameters that are chosen for six neural networks.

Hyper-parameter	MLP	CNN	LSTM	GRU	BLSTM	BGRU
Output dimension	512	-	256	256	256	256
Filters	-	32	-	-	-	-
Batch size	16	16	32	16	32	32
Dropout	0.1	0.2	0.1	0.1	0.4	0.2
#Hidden layers	2	5	2	1	3	2

TABLE 6. Comparison among the effectiveness of six neural networks (with the code gadgets involving data dependency and control dependency, with no imbalanced data processing, and with the last time step as the selection strategy of output for RNNs).

Neural network	FPR (%)	FNR (%)	A (%)	P (%)	F1 (%)
MLP	2.5	15.5	94.9	89.3	86.8
CNN	1.8	9.9	96.5	93.3	91.6
LSTM	3.3	9.9	95.3	88.1	89.1
GRU	2.9	5.8	96.5	89.7	91.9
BLSTM	2.5	6.2	96.7	90.9	92.4
BGRU	2.2	6.8	96.8	91.9	92.5

Table 6 summarizes the results of the above six neural networks. We observe that when compared with unidirectional RNNs (i.e., LSTM and GRU), the bidirectional RNNs (i.e., BLSTM and BGRU) can increase the F1-measure by 2.0% on average. This phenomenon might be caused by the following: unidirectional RNNs can only accommodate the information about the statements that appear before or after (not both) the statement in question. In fact, a vulnerable statement may be affected by earlier statements in the program and may be also affected by later statements. Therefore, bidirectional RNNs which can accommodate more information about the statements in two directions are better. The architectures of bidirectional RNNs make them more suitable for coping with sequential data. Moreover, there is no great difference in most metrics between BLSTM and BGRU (i.e., one is not significantly better than the other), though BGRU has a little higher F1-measure than BLSTM. We also observe that the MLP with the F1-measure of 86.8% is worse than other neural networks, which demonstrates that the feed-forward neural network in which neurons in each layer are fully connected to the neurons in the next layer does not do well in learning the semantics of code gadgets. We further observe that CNN, which is good at modeling position-invariant features, is in between bidirectional RNNs and MLP.

We take CNN and BLSTM as examples to show the time complexity of training and detection. It takes 45,204 seconds to train a CNN and 2,410 seconds to detect vulnerabilities, and it takes 106,829 seconds to train a BLSTM and 4,431 seconds to detect vulnerabilities. Not surprisingly, the training time is much larger than the detection time. The CNN spends much shorter time for training and detection than the BLSTM, as the neural networks themselves imply.

TABLE 7. Impact of l on the effectiveness of *average of l-max*.

l	FPR (%)	FNR (%)	A (%)	P (%)	F1 (%)
1	1.6	9.9	96.6	93.7	91.9
2	2.0	8.2	96.7	92.5	92.1
3	2.1	9.5	96.3	92.0	91.2
4	3.1	10.2	95.4	88.7	89.3
5	5.5	43.8	86.3	73.5	63.7

Insight 3: Bidirectional RNNs are more effective than unidirectional RNNs and CNN, which in turn are more effective than MLP.

2) COMPARISON AMONG DIFFERENT SELECTION STRATEGIES OF OUTPUTS

For RNNs, different selection strategies of outputs corresponding to time steps may impact on the effectiveness of deep learning-based vulnerability detection. In order to test the impact of different selection strategies of outputs, we take BLSTM as an example, and test two selection strategies which are used for back propagation to train the neural network without imbalanced data processing, while noting that similar phenomenon is observed for other recurrent neural networks. One is using the output corresponding to the last time step for a code gadget (denoted as ''last time step''); the other one is using the average of l largest outputs corresponding to time steps for a code gadget (denoted as ''average of l-max'').

For the selection strategy of *average of l-max*, Table 7 shows the impact of different values of l on the effectiveness of vulnerability detection. Using the average of two largest outputs corresponding to time steps (i.e., $l = 2$) achieves the highest overall effectiveness F1-measure (i.e., 92.1%) and the lowest FNR (i.e., 8.2%), while using the largest output corresponding to the time step (i.e., $l = 1$) achieves the lowest FPR (i.e., 1.6%). For l ($l \geq 2$), the larger l is, the less effective in terms of all five metrics. In what follows, we select $l = 2$ since it achieves the highest overall effectiveness for the selection strategy of *average of l-max*.

Table 8 summarizes the results for two selection strategies of outputs corresponding to time steps. We observe that the *average of l-max* ($l = 2$) achieves a 0.5% lower FPR and a 1.6% higher precision at the cost of a 2.0% higher FNR and a 0.3% lower F1-measure, compared with the *last time step*. As can be seen from the overall effectiveness F1-measure, the *last time step* is a little better. We speculate the higher FNR of the *average of l-max* ($l = 2$) is caused by the following: the time steps with the l largest outputs may correspond to any intermediate time steps, thus the *average of l-max* cannot better grasp the overall information for all time steps, compared with the last time step. Local information limits the scope of learning vulnerability features, especially for long code gadgets. Therefore, some vulnerability features cannot be learned, which causes a higher FNR. In addition, VulDeePecker [16] with last time step is less effectiveness

TABLE 8. Effectiveness of BLSTM using different selection strategies of outputs corresponding to time steps (with the code gadgets involving data dependency and control dependency and with no imbalanced data processing).

Selection strategy	FPR (%)	FNR (%)	A (%)	P (%)	F1 (%)
Last time step	2.5	6.2	96.7	90.9	92.4
Average of l-max ($l = 2$)	2.0	8.2	96.7	92.5	92.1
VulDeePecker [16]	2.0	37.4	92.4	84.9	72.1
SySeVR [17] with BGRU	2.2	6.8	96.8	91.9	92.5

in most metrics, especially with 31.2% higher FNR and 20.3% lower F1-measure compared with the method using the *last time step*. This is mainly caused by using only data dependency as the semantic information in the code gadget. SySeVR [17] with BGRU uses the last time step and is not significantly better than the BLSTM with last time step, though SySeVR with BGRU has a little higher F1-measure (0.1%). The difference between BLSTM with last time step and SySeVR with BGRU is mainly caused by different neural network. In summary, we draw:

Insight 4: Different selection strategies of outputs corresponding to time steps for recurrent neural networks can influence the effectiveness of vulnerability detection. Specifically, using the last output corresponding to the time step for BLSTM can reduce the FNR by 2.0% at the price of increasing the FPR by 0.5%.

V. RELATED WORK

As discussed in Section I, there are mainly two types of approaches to static vulnerability detection for source code: code similarity-based approaches and pattern-based approaches. The *code similarity-based* approaches detect the vulnerabilities caused by code clone, i.e., the vulnerable pieces of code which are similar to the code of given vulnerabilities. When they are used to detect vulnerabilities that are not caused by code clones, high false negatives occur. The *pattern-based* approaches detect a type of vulnerabilities through matching vulnerability patterns which can be generated manually or automatically. Deep learning-based vulnerability detection belongs to the pattern-based approaches. In this section, we review the prior work on the pattern-based vulnerability detection approaches, which mainly involve three categories: rule-based, traditional machine learning-based, and deep learning-based methods.

A. RULE-BASED METHODS

Rule-based methods are based on vulnerability rules (i.e., patterns) which are manually defined by human experts. The present static vulnerability detection tools, which provide the corresponding rules for each vulnerability type, belong to this group. This category includes the lightweight methods that generate patterns from source code (e.g., open source tools Flawfinder [4], RATS [34], ITS4 [35], and commercial tool Checkmarx [5]) and the more comprehensive methods that generate patterns based on intermediate code (e.g., commercial tools Fortify [6] and Coverity [7]). These tools can report the locations and the types of vulnerabilities, but cannot accurately distinguish between various vulnerable code and non-vulnerable code, resulting in high false positives or high false negatives [16], [36].

B. TRADITIONAL MACHINE LEARNING-BASED METHODS

Traditional machine learning-based methods rely on human experts for defining features to characterize vulnerabilities, and use traditional machine learning techniques [37], such as k-nearest neighbor and support vector machine, to classify vulnerable code and non-vulnerable code. Ghaffarian and Shahriari [38] reviewed the approaches of vulnerability analysis and discovery using machine-learning techniques, most of which are traditional machine learning-based methods. Traditional machine learning-based methods can be characterized by the granularity and attributes. *Granularity* describes the "atomic" unit of code that is treated as a whole. Many granularities have been investigated, including program [9], package [39], component [10], [40], file [11], [41], and function [12], [42], [43]. *Attributes* are the features that are defined to characterize pieces of code, including terms and their occurrence frequencies [10], imports and function calls [40], complexity, code churn, and developer activity [41], dependency relation [39], API symbols and subtrees [12], [42], system calls [9], and the combination of some features above [43]. Therefore, these approaches rely on human experts to manually define features, and cannot pin down the precise locations of vulnerabilities because programs are represented in some coarse-grained granularities.

C. DEEP LEARNING-BASED METHODS

Deep learning-based methods do not need to manually define features to characterize vulnerabilities, and can learn the features of vulnerable code automatically. Lin *et al.* [13] presented an approach to automatic learning high-level representations of functions based on their abstract syntax trees for vulnerability discovery. Russell *et al.* [14] proposed a large-scale function-level vulnerability detection system to learn deep feature representation of source code after lexical analysis. It combined the neural feature representations of function source code with random forest as a classifier. Harer *et al.* [15] used machine learning methods to perform the data-driven vulnerability detection, and compared the effectiveness of using the source code and the compiled code. The granularity of above approaches is function-level involving large numbers of statements which are not related to vulnerabilities. Therefore, they are coarse-grained and cannot pin down the precise locations of vulnerabilities. The recently proposed VulDeePecker [16] is the first to use deep learning to detect vulnerabilities at the slice level (finer than function level), effectively excluding the statements which are not related to vulnerabilities. SySeVR [17] is a

TABLE 9. 811 security-related C/C++ library/API function calls corresponding to all kinds of known vulnerabilities, where "∗" is the wildcard.

StrNCat, getaddrinfo, _ui64toa, fclose, pthread_mutex_lock, gets_s, sleep, _ui64tot, freopen_s, _ui64tow, send, lstrcat, HMAC_Update, __fxstat, StrCatBuff, _mbscat, _mbstok_s, _cprintf_s, ldap_search_init_page, memmove_s, ctime_s, vswprintf, vswprintf_s, _snwprintf, _gmtime_s, _tccpy, ∗RC6∗, _mbslwr_s, random, __wcstof_internal, _wcslwr_s, _ctime32_s, wcsncat∗, MD5_Init, _ultoa, snprintf, memset, syslog, _vsnprintf_s, HeapAlloc, pthread_mutex_destroy, ChangeWindowMessageFilter, _ultot, crypt_r, _strupr_s_l, LoadLibraryExA, _strerror_s, LoadLibraryExW, wvsprintf, MoveFileEx, _strdate_s, SHA1, sprintfW, StrCatNW, _scanf_s_l, pthread_attr_init, _wtmpnam_s, snscanf, _sprintf_s_l, dlopen, sprintfA, timed_mutex, OemToCharA, ldap_delete_ext, sethostid, popen, OemToCharW, _gettws, vfork, _wcsnset_s_l, sendmsg, _mbsncat, wvnsprintfA, HeapFree, _wcserror_s, realloc, _snprintf∗, wcstok, _strncat∗, StrNCpy, _wasctime_s, push∗, _lfind_s, CC_SHA512, ldap_compare_ext_s, wcscat_s, strdup, _chsize_s, sprintf_s, CC_MD4_Init, wcsncpy, _wfreopen_s, _wcsupr_s, _searchenv_s, ldap_modify_ext_s, _wsplitpath, CC_SHA384_Final, MD2, RtlCopyMemory, lstrcatW, MD4, MD5, _wcstok_s_l, _vsnwprintf_s, ldap_modify_s, strerror, _lsearch_s, _mbsnbcat_s, _wsplitpath_s, MD4_Update, _mbccpy_s, _strncpy_s_l, _snprintf_s, CC_SHA512_Init, fwscanf_s, _snwprintf_s, CC_SHA1, swprintf, fprintf, EVP_DigestInit_ex, strlen, SHA1_Init, strncat, _getws_s, CC_MD4_Final, wnsprintfW, lcong48, lrand48, write, HMAC_Init, _wfopen_s, wmemchr, _tmakepath, wnsprintfA, lstrcpynW, scanf_s, _mbsncpy_s_l, _localtime64_s, fstream.open, _wmakepath, Connection.open, _tccat, valloc, setgroups, unlink, fstream.put, wsprintfA, ∗SHA1∗, _wsearchenv_s, ualstrcpyA, CC_MD5_Update, strerror_s, HeapCreate, ualstrcpyW, __xstat, _wmktemp_s, StrCatChainW, ldap_search_st, _mbstowcs_s_l, ldap_modify_ext, _mbsset_s, strncpy_s, move, execle, StrCat, xrealloc, wcsncpy_s, _tcsncpy∗, execlp, RIPEMD160_Final, ldap_search_s, EnterCriticalSection, _wctomb_s_l, fwrite, _gmtime64_s, sscanf_s, wcscat, _strupr_s, wcrtomb_s, VirtualLock, ldap_add_ext_s, _mbscpy, _localtime32_s, lstrcpy, _wcsncpy∗, CC_SHA1_Init, _getts, _wfopen, __xstat64, strcoll, _fwscanf_s_l, _mbslwr_s_l, RegOpenKey, makepath, seed48, CC_SHA256, sendto, execv, CalculateDigest, memchr, _mbscpy_s, _strtime_s, ldap_search_ext_s, _chmod, flock, __fxstat64, _vsntprintf, CC_SHA256_Init, _itoa_s, __wcserror_s, _gcvt_s, fstream.write, sprintf, recursive_mutex, strrchr, gethostbyaddr, _wcsupr_s_l, strcspn, MD5_Final, asprintf, _wcstombs_s_l, _tcstok, free, MD2_Final, asctime_s, _alloca, _wputenv_s, _wcsset_s, _wcslwr_s_l, SHA1_Update, filebuf.sputc, filebuf.sputn, SQLConnect, ldap_compare, mbstowcs_s, HMAC_Final, pthread_condattr_init, _ultow_s, rand, ofstream.put, CC_SHA224_Final, lstrcpynA, bcopy, system, CreateFile∗, wcscpy_s, _mbsnbcpy∗, open, _vsnwprintf, strncpy, getopt_long, CC_SHA512_Final, _vsprintf_s_l, scanf, mkdir, _localtime_s, _snprintf, _mbccpy_s_l, memcmp, final, _ultoa_s, lstrcpyW, LoadModule, _swprintf_s_l, MD5_Update, _mbsnset_s_l, _wstrtime_s, _strnset_s, lstrcpyA, _mbsnbcpy_s, mlock, IsBadHugeWritePtr, copy, _mbsnbcpy_s_l, wnsprintf, wcscpy, ShellExecute, CC_MD4, _ultow, _vsnwprintf_s_l, lstrcpyn, CC_SHA1_Final, vsnprintf, _mbsnbset_s, _i64tow, SHA256_Init, wvnsprintf, RegCreateKey, strtok_s, _wctime32_s, _i64toa, CC_MD5_Final, wmemcpy, WinExec, CreateDirectory∗, CC_SHA256_Update, _vsnprintf_s_l, jrand48, wsprintf, ldap_rename_ext_s, filebuf.open, _wsystem, SHA256_Update, _cwscanf_s, wsprintfW, _sntscanf, _splitpath, fscanf_s, strpbrk, wcstombs_s, wscanf, _mbsnbcat_s_l, strcpynA, pthread_cond_init, wcsrtombs_s, _wsopen_s, CharToOemBuffA, RIPEMD160_Update, _tscanf, HMAC, StrCCpy, Connection.connect, lstrcatn, _mbstok, _mbsncpy, CC_SHA384_Update, create_directories, pthread_mutex_unlock, CFile.Open, connect, _vswprintf_s_l, _snscanf_s_l, fputc, _wscanf_s, _snprintf_s_l, strtok, _strtok_s_l, lstrcatA, snwscanf, pthread_mutex_init, fputs, CC_SHA384_Init, _putenv_s, CharToOemBuffW, pthread_mutex_trylock, __wcstoul_internal, _memccpy, _snwprintf_s_l, _strncpy∗, wmemset, MD4_Init, ∗RC4∗, strcpyW, _ecvt_s, memcpy_s, erand48, IsBadHugeReadPtr, strcpyA, HeapReAlloc, memcpy, ldap_rename_ext, fopen_s, srandom, _cgetws_s, _makepath, SHA256_Final, remove, _mbsupr_s, pthread_mutexattr_init, __wcstold_internal, StrCpy, ldap_delete, wmemmove_s, _mkdir, strcat, _cscanf_s_l, StrCAdd, swprintf_s, _strnset_s_l, close, ldap_delete_ext_s, ldap_modrdn, strchr, _gmtime32_s, _ftcscat, lstrcatnA, _tcsncat, OemToChar, mutex, CharToOem, strcpy_s, lstrcatnW, _wscanf_s_l, __lxstat64, memalign, MD2_Init, StrCatBuffW, StrCpyN, CC_MD5, StrCpyA, StrCatBuffA, StrCpyW, tmpnam_r, _vsnprintf, strcatA, StrCpyNW, _mbsnbset_s_l, EVP_DigestInit, _stscanf, CC_MD2, _tcscat, StrCpyNA, xmalloc, _tcslen, ∗MD4∗, vasprintf, strxfrm, chmod, ldap_add_ext, alloca, _snscanf_s, IsBadWritePtr, swscanf_s, wmemcpy_s, _itoa, _ui64toa_s, EVP_DigestUpdate, __wcstol_internal, _itow, StrNCatW, strncat_s, ualstrcpy, execvp, _mbccat, EVP_MD_CTX_init, assert, ofstream.write, ldap_add, _sscanf_s_l, drand48, CharToOemW, swscanf, _itow_s, RIPEMD160_Init, CopyMemory, initstate, getpwuid, vsprintf, _fcvt_s, CharToOemA, setuid, malloc, StrCatNA, strcat_s, srand, getwd, _controlfp_s, olestrcpy, __wcstod_internal, _mbsnbcat, lstrncat, des_∗, CC_SHA224_Init, set∗, vsprintf_s, SHA1_Final, _umask_s, gets, setstate, wvsprintfW, LoadLibraryEx, ofstream.open, calloc, _mbstrlen, _cgets_s, _sopen_s, IsBadStringPtr, wcsncat_s, add∗, nrand48, create_directory, ldap_search_ext, _i64toa_s, _ltoa_s, _cwscanf_s_l, wmemcmp, __lxstat, lstrlen, pthread_condattr_destroy, _ftcscpy, wcstok_s, __xmknod, pthread_attr_destroy, sethostname, _fscanf_s_l, StrCatN, RegEnumKey, _tcsncpy, strcatW, AfxLoadLibrary, setenv, tmpnam, _mbsncat_s_l, _wstrdate_s, _wctime64_s, _i64tow_s, CC_MD4_Update, ldap_add_s, _umask, CC_SHA1_Update, _wcsset_s_l, _mbsupr_s_l, strstr, _tsplitpath, memmove, _tcscpy, vsnprintf_s, strcmp, wvnsprintfW, tmpfile, ldap_modify, _mbsncat∗, mrand48, sizeof, StrCatA, _ltow_s, ∗desencrypt∗, StrCatW, _mbccpy, CC_MD2_Init, RIPEMD160, ldap_search, CC_SHA224, mbsrtowcs_s, update, ldap_delete_s, getnameinfo, ∗RC5∗, _wcsncat_s_l, DriverManager.getConnection, socket, _cscanf_s, ldap_modrdn_s, _wopen, CC_SHA256_Final, _snwprintf∗, MD2_Update, strcpy, _strncat_s_l, CC_MD5_Init, mbscpy, wmemmove, LoadLibraryW, _mbslen, ∗alloc, _mbsncat_s, LoadLibraryA, fopen, StrLen, delete, _splitpath_s, CreateFileTransacted∗, MD4_Final, _open, CC_SHA384, wcslen, wcsncat, _mktemp_s, pthread_mutexattr_destroy, _snwscanf_s, _strset_s, _wcsncpy_s_l, CC_MD2_Final, _mbstok_s_l, wctomb_s, MySQL_Driver.connect, _snwscanf_s_l, ∗_des_∗, LoadLibrary, _swscanf_s_l, ldap_compare_s, ldap_compare_ext, _strlwr_s, GetEnvironmentVariable, cuserid, _mbscat_s, strspn, _mbsncpy_s, ldap_modrdn2, LeaveCriticalSection, CopyFile, getpwd, sscanf, creat, RegSetValue, ldap_modrdn2_s, CFile.Close, ∗SHA_1∗, pthread_cond_destroy, CC_SHA512_Update, ∗RC2∗, StrNCatA, _mbsnbcpy, _mbsnset_s, crypt, excel, _vstprintf, xstrdup, wvsprintfA, getopt, mkstemp, _wcsnset_s, _stprintf, _sntprintf, tmpfile_s, OpenDocumentFile, _mbsset_s_l, _strset_s_l, _strlwr_s_l, ifstream.open, xcalloc, StrNCpyA, _wctime_s, CC_SHA224_Update, _ctime64_s, MoveFile, chown, StrNCpyW, IsBadReadPtr, _ui64tow_s, IsBadCodePtr, getc, OracleCommand.ExecuteOracleScalar, AccessDataSource.Insert, IDbDataAdapter.FillSchema, IDbDataAdapter.Update, GetWindowText∗, SendMessage, SqlCommand.ExecuteNonQuery, streambuf.sgetc, streambuf.sgetn, OracleCommand.ExecuteScalar, SqlDataSource.Update, _Read_s, IDataAdapter.Fill, _wgetenv, _RecordsetPtr.Open∗, AccessDataSource.Delete, Recordset.Open∗, filebuf.sbumpc, DDX_∗, RegGetValue, fstream.read∗, SqlCeCommand.ExecuteResultSet, SqlCommand.ExecuteXmlReader, main, streambuf.sputbackc, read, m_lpCmdLine, CRichEditCtrl.Get∗, istream.putback, SqlCeCommand.ExecuteXmlReader, SqlCeCommand.BeginExecuteXmlReader, filebuf.sgetn, OdbcDataAdapter.Update, filebuf.sgetc, SQLPutData, recvfrom, OleDbDataAdapter.FillSchema, IDataAdapter.FillSchema, CRichEditCtrl.GetLine, DbDataAdapter.Update, SqlCommand.ExecuteReader, istream.get, ReceiveFrom, _main, fgetc, DbDataAdapter.FillSchema, kbhit, UpdateCommand.Execute∗, Statement.execute, fgets, SelectCommand.Execute∗, getch, OdbcCommand.ExecuteNonQuery, CDaoQueryDef.Execute, fstream.getline, ifstream.getline, SqlDataAdapter.FillSchema, OleDbCommand.ExecuteReader, Statement.execute∗, SqlCeCommand.BeginExecuteNonQuery, OdbcCommand.ExecuteScalar, SqlCeDataAdapter.Update, sendmessage, mysqlpp.DBDriver, fstream.peek, Receive, CDaoRecordset.Open, OdbcDataAdapter.FillSchema, _wgetenv_s, OleDbDataAdapter.Update, readsome, SqlCommand.BeginExecuteXmlReader, recv, ifstream.peek, _Main, _tmain, _Readsome_s, SqlCeCommand.ExecuteReader, OleDbCommand.ExecuteNonQuery, fstream.get, IDbCommand.ExecuteScalar, filebuf.sputbackc, IDataAdapter.Update, streambuf.sbumpc, InsertCommand.Execute∗, RegQueryValue, IDbCommand.ExecuteReader, SqlPipe.ExecuteAndSend, Connection.Execute∗, getdlgtext, ReceiveFromEx, SqlDataAdapter.Update, RegQueryValueEx, SQLExecute, pread, SqlCommand.BeginExecuteReader, AfxWinMain, getchar, istream.getline, SqlCeDataAdapter.Fill, OleDbDataReader.ExecuteReader, SqlDataSource.Insert, istream.peek, SendMessageCallback, ifstream.read∗, SqlDataSource.Select, SqlCommand.ExecuteScalar, SqlDataAdapter.Fill, SqlCommand.BeginExecuteNonQuery, getche, SqlCeCommand.BeginExecuteReader, getenv, streambuf.snextc, Command.Execute∗, _CommandPtr.Execute∗, SendNotifyMessage, OdbcDataAdapter.Fill, AccessDataSource.Update, fscanf, QSqlQuery.execBatch, DbDataAdapter.Fill, cin, DeleteCommand.Execute∗, QSqlQuery.exec, PostMessage, ifstream.get, filebuf.snextc, IDbCommand.ExecuteNonQuery, Winmain, fread, getpass, GetDlgItemTextCCheckListBox.GetCheck, DISP_PROPERTY_EX, pread64, Socket.Receive∗, SACommand.Execute∗, SQLExecDirect, SqlCeDataAdapter.FillSchema, DISP_FUNCTION, OracleCommand.ExecuteNonQuery, CEdit.GetLine, OdbcCommand.ExecuteReader, CEdit.Get∗, AccessDataSource.Select, OracleCommand.ExecuteReader, OCIStmtExecute, getenv_s, DB2Command.Execute∗, OracleDataAdapter.FillSchema, OracleDataAdapter.Fill, CComboBox.Get∗, SqlCeCommand.ExecuteNonQuery, OracleCommand.ExecuteOracleNonQuery, mysqlpp.Query, istream.read∗, CListBox.GetText, SqlCeCommand.ExecuteScalar, ifstream.putback, readlink, CHtmlEditCtrl.GetDHtmlDocument, PostThreadMessage, CListCtrl.GetItemText, OracleDataAdapter.Update, OleDbCommand.ExecuteScalar, stdin, SqlDataSource.Delete, OleDbDataAdapter.Fill, fstream.putback, IDbDataAdapter.Fill, _wspawnl, fwprintf, sem_wait, _unlink, ldap_search_ext_sW, signal, PQclear, PQfinish, PQexec, PQresultStatus.

deep learning-based framework which uses syntax-based, semantics-based, and vector representations to detect various types of vulnerabilities at the slice level. Nevertheless, there is no systematic comparative study to show the quantitative impact of different factors on the effectiveness of vulnerability detection.

The present study follows VulDeePecker and more specifically studies the impact of control dependency in the code gadget, the imbalanced data processing, and the neural networks on the deep learning-based vulnerability detection. As discussed above, the extension is based on a completely new implementation using an extended open source tool *Joern*, because a straightforward extension cannot accommodate new semantic information for VulDeePecker which is based on the commercial tool Checkmarx.

VI. CONCLUSION AND FUTURE WORK

We conduct a comparative study to evaluate the quantitative impact of different factors on the effectiveness of deep learning-based vulnerability detection, involving more semantic information, imbalanced data processing, and different neural networks. For this purpose, we collect a dataset from the programs involving 126 types of vulnerabilities and implement the deep learning-based vulnerability detection system. Experimental results show using the code representation with data dependency and control dependency, no imbalanced data processing, BRNNs, and the last output corresponding to the time step is the best selection for deep learning-based vulnerability detection.

However, the present study has several limitations. First, we focus on detecting the vulnerabilities related to library/API function calls; future research needs to accommodate other kinds of vulnerabilities for comparative study. Second, the present implementation for code gadgets generation focuses on dealing with C/C++ programs; future comparative study will consider introducing the intermediate language to support multiple programming languages. Third, hot map can help identify important code elements in the code gadgets for vulnerability detection and speculate what features the deep learning model has learned; more investigations are needed to explain the deep learning model in detail.

APPENDIX

A. LIST OF C/C++ LIBRARY/API FUNCTION CALLS

Table 9 lists the 811 security-related C/C++ library/API function calls according to all kinds of known vulnerabilities.

B. LIST OF THE 126 TYPES OF VULNERABILITIES

The 126 types of vulnerabilities (i.e., CWE IDs) covered in the programs of present study are as follows:
CWE-015, CWE-020, CWE-022, CWE-023, CWE-036, CWE-078, CWE-080, CWE-088, CWE-089, CWE-090, CWE-114, CWE-119, CWE-120, CWE-121, CWE-122, CWE-123, CWE-124, CWE-126, CWE-127, CWE-129, CWE-134, CWE-170, CWE-176, CWE-188, CWE-190, CWE-191, CWE-194, CWE-195, CWE-196, CWE-197, CWE-222, CWE-223, CWE-242, CWE-244, CWE-252, CWE-253, CWE-256, CWE-259, CWE-272, CWE-284, CWE-319, CWE-321, CWE-325, CWE-327, CWE-338, CWE-345, CWE-362, CWE-363, CWE-364, CWE-366, CWE-367, CWE-369, CWE-377, CWE-398, CWE-400, CWE-401, CWE-404, CWE-412, CWE-414, CWE-415, CWE-416, CWE-426, CWE-427, CWE-457, CWE-459, CWE-464, CWE-467, CWE-468, CWE-469, CWE-475, CWE-476, CWE-479, CWE-489, CWE-506, CWE-510, CWE-526, CWE-534, CWE-535, CWE-543, CWE-562, CWE-571, CWE-587, CWE-588, CWE-590, CWE-591, CWE-605, CWE-606, CWE-609, CWE-617, CWE-620, CWE-663, CWE-665, CWE-666, CWE-672, CWE-674, CWE-675, CWE-680, CWE-681, CWE-682, CWE-685, CWE-688, CWE-690, CWE-704, CWE-758, CWE-761, CWE-762, CWE-765, CWE-771, CWE-773, CWE-774, CWE-775, CWE-780, CWE-785, CWE-789, CWE-805, CWE-806, CWE-821, CWE-822, CWE-824, CWE-828, CWE-831, CWE-833, CWE-834, CWE-835, CWE-839, CWE-843.

REFERENCES

[1] (2019). *Common Vulnerabilities Exposures (CVE)*. [Online]. Available: http://cve.mitre.org/
[2] S. Kim, S. Woo, H. Lee, and H. Oh, "VUDDY: A scalable approach for vulnerable code clone discovery," in *Proc. IEEE Symp. Secur. Privacy (SP)*, San Jose, CA, USA, May 2017, pp. 595–614.
[3] Z. Li, D. Zou, S. Xu, H. Jin, H. Qi, and J. Hu, "VulPecker: An automated vulnerability detection system based on code similarity analysis," in *Proc. 32nd Annu. Conf. Comput. Secur. Appl.*, Los Angeles, CA, USA, Dec. 2016, pp. 201–213.
[4] (2019). *FlawFinder*. [Online]. Available: http://www.dwheeler.com/flawfinder
[5] (2019). *Checkmarx*. [Online]. Available: https://www.checkmarx.com/
[6] (2019). *HP Fortify*. [Online]. Available: https://www.hpfod.com/
[7] (2019). *Coverity*. [Online]. Available: https://scan.coverity.com/
[8] Z. Fang, Q. Liu, Y. Zhang, K. Wang, Z. Wang, and Q. Wu, "A static technique for detecting input validation vulnerabilities in Android apps," *Sci. China Inf. Sci.*, vol. 60, no. 5, May 2017, Art. no. 052111.
[9] G. Grieco, G. L. Grinblat, L. Uzal, S. Rawat, J. Feist, and L. Mounier, "Toward large-scale vulnerability discovery using machine learning," in *Proc. 6th ACM Conf. Data Appl. Secur. Privacy*, New Orleans, LA, USA, Mar. 2016, pp. 85–96.
[10] R. Scandariato, J. Walden, A. Hovsepyan, and W. Joosen, "Predicting vulnerable software components via text mining," *IEEE Trans. Softw. Eng.*, vol. 40, no. 10, pp. 993–1006, Oct. 2014.
[11] S. Moshtari and A. Sami, "Evaluating and comparing complexity, coupling and a new proposed set of coupling metrics in cross-project vulnerability prediction," in *Proc. 31st Annu. ACM Symp. Appl. Comput.*, Pisa, Italy, Apr. 2016, pp. 1415–1421.
[12] F. Yamaguchi, M. Lottmann, and K. Rieck, "Generalized vulnerability extrapolation using abstract syntax trees," in *Proc. 28th Annu. Comput. Secur. Appl. Conf.*, Orlando, FL, USA, Dec. 2012, pp. 359–368.
[13] G. Lin, J. Zhang, W. Luo, L. Pan, and Y. Xiang, "POSTER: Vulnerability discovery with function representation learning from unlabeled projects," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, Dallas, TX, USA, 2017, pp. 2539–2541.
[14] R. Russell, L. Kim, L. Hamilton, T. Lazovich, J. Harer, O. Ozdemir, P. Ellingwood, and M. W. McConley, "Automated vulnerability detection in source code using deep representation learning," in *Proc. 17th IEEE Int. Conf. Mach. Learn. Appl. (ICMLA)*, Orlando, FL, USA, Dec. 2018, pp. 757–762.
[15] J. A. Harer, L. Y. Kim, R. L. Russell, O. Ozdemir, L. R. Kosta, A. Rangamani, L. H. Hamilton, G. I. Centeno, J. R. Key, P. M. Ellingwood, E. Antelman, A. Mackay, M. W. McConley, J. M. Opper, S. P. Chin, and T. Lazovich, "Automated software vulnerability detection with machine learning," Aug. 2018, *arXiv:1803.04497*. [Online]. Available: https://arxiv.org/abs/1803.04497
[16] Z. Li, D. Zou, S. Xu, X. Ou, H. Jin, S. Wang, Z. Deng, and Y. Zhong, "VulDeePecker: A deep learning-based system for vulnerability detection," in *Proc. NDSS*, San Diego, CA, USA, 2018, pp. 1–15.
[17] Z. Li, D. Zou, S. Xu, H. Jin, Y. Zhu, and Z. Chen, "SySeVR: A framework for using deep learning to detect software vulnerabilities," Sep. 2018, *arXiv:1807.06756*. [Online]. Available: https://arxiv.org/abs/1807.06756
[18] F. Yamaguchi, N. Golde, D. Arp, and K. Rieck, "Modeling and discovering vulnerabilities with code property graphs," in *Proc. IEEE Symp. Secur. Privacy*, Berkeley, CA, USA, May 2014, pp. 590–604.
[19] J. Li and M. D. Ernst, "CBCD: Cloned buggy code detector," in *Proc. 34th Int. Conf. Softw. Eng.*, Zurich, Switzerland, Jun. 2012, pp. 310–320.
[20] N. H. Pham, T. T. Nguyen, H. A. Nguyen, and T. N. Nguyen, "Detection of recurring software vulnerabilities," in *Proc. IEEE/ACM Int. Conf. Automated Softw. Eng.*, Antwerp, Belgium, Sep. 2010, pp. 447–456.
[21] (2019). *National Vulnerability Database*. [Online]. Available: https://nvd.nist.gov/

[22] (2019). *Software Assurance Reference Dataset*. [Online]. Available: https://samate.nist.gov/SRD/index.php

[23] T. Fushiki, "Estimation of prediction error by using κ-fold cross-validation," *Statist. Comput.*, vol. 21, no. 2, pp. 137–146, 2011.

[24] I. Mani and I. Zhang, "kNN approach to unbalanced data distributions: A case study involving information extraction," in *Proc. Workshop Learn. Imbalanced Datasets*, Aug. 2003, pp. 42–48.

[25] N. V. Chawla, K. W. Bowyer, L. O. Hall, and W. P. Kegelmeyer, "SMOTE: Synthetic minority over-sampling technique," *J. Artif. Intell. Res.*, vol. 16, no. 1, pp. 321–357, 2002.

[26] A. Shrivastava, A. Gupta, and R. Girshick, "Training region-based object detectors with online hard example mining," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit. (CVPR)*, Las Vegas, NV, USA, Jun. 2016, pp. 761–769.

[27] K. K. Sung, "Learning and example selection for object and pattern detection," Ph.D. dissertation, Dept. Elect. Eng. Comput. Sci., MIT, Cambridge, MA, USA, 1996.

[28] M. Abadi *et al.*, "TensorFlow: A system for large-scale machine learning," in *Proc. 12th USENIX Conf. Operating Syst. Design Implement.*, Savannah, GA, USA, Nov. 2016, pp. 265–283.

[29] M. Pendleton, R. Garcia-Lebron, J.-H. Cho, and S. Xu, "A survey on systems security metrics," *ACM Comput. Surv.*, vol. 49, p. 62, Dec. 2016.

[30] W. Qiang, Y. Liao, G. Sun, L. T. Yang, D. Zou, and H. Jin, "Patch-related vulnerability detection based on symbolic execution," *IEEE Access*, vol. 5, pp. 20777–20784, 2017.

[31] (2018). *Common Weakness Enumeration*. [Online]. Available: http://cwe.mitre.org/

[32] Y. Jung, "Multiple predicting κ-fold cross-validation for model selection," *J. Nonparam. Statist.*, vol. 30, no. 1, pp. 197–215, 2018.

[33] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," Dec. 2014, *arXiv:1412.6980*. [Online]. Available: https://arxiv.org/abs/1412.6980

[34] (2014). *Rough Audit Tool for Security*. [Online]. Available: https://code.google.com/archive/p/rough-auditing-tool-for-security/

[35] J. Viega, J. T. Bloch, Y. Kohno, and G. McGraw, "ITS4: A static vulnerability scanner for C and C++ code," in *Proc. 16th Annu. Comput. Secur. Appl. Conf.*, New Orleans, LA, USA, Dec. 2000, pp. 257–267.

[36] F. Yamaguchi, "Pattern-based vulnerability discovery," Ph.D. dissertation, Dept. Comput. Sci., Univ. Göttingen, Göttingen, Germany, 2015.

[37] Y. Cao, Y. Zou, Y. Luo, B. Xie, and J. Zhao, "Toward accurate link between code and software documentation," *Sci. China Inf. Sci.*, vol. 61, no. 5, May 2018, Art. no. 050105.

[38] S. M. Ghaffarian and H. R. Shahriari, "Software vulnerability analysis and discovery using machine-learning and data-mining techniques: A survey," *ACM Comput. Surv.*, vol. 50, no. 4, p. 56, Nov. 2017.

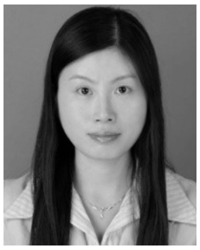
[39] S. Neuhaus and T. Zimmermann, "The beauty and the beast: Vulnerabilities in red hat's packages," in *Proc. USENIX ATC*, San Diego, CA, USA, 2009, pp. 1–14.

[40] S. Neuhaus, T. Zimmermann, C. Holler, and A. Zeller, "Predicting vulnerable software components," in *Proc. 14th ACM Conf. Comput. Commun. Secur.*, Alexandria, VA, USA, 2007, pp. 529–540.

[41] Y. Shin, A. Meneely, L. Williams, and J. A. Osborne, "Evaluating complexity, code churn, and developer activity metrics as indicators of software vulnerabilities," *IEEE Trans. Softw. Eng.*, vol. 37, no. 6, pp. 772–787, Dec. 2011.

[42] F. Yamaguchi, F. F. Lindner, and K. Rieck, "Vulnerability extrapolation: Assisted discovery of vulnerabilities using machine learning," in *Proc. WOOT*, San Francisco, CA, USA, 2011, p. 13.

[43] B. Chernis and R. Verma, "Machine learning methods for software vulnerability detection," in *Proc. 4th ACM Int. Workshop Secur. Privacy Anal.*, Tempe, AZ, USA, Mar. 2018, pp. 31–39.



ZHEN LI received the B.E. and M.E. degrees in computer science and technology from Hebei University, Baoding, China, in 2003 and 2006, respectively. She is currently pursuing the Ph.D. degree in cyberspace security with the Huazhong University of Science and Technology, Wuhan, China. Her research interests include vulnerability detection and software security.

DEQING ZOU received the Ph.D. degree from the Huazhong University of Science and Technology, Wuhan, China, in 2004. He is currently a Professor of cyberspace security with the Huazhong University of Science and Technology. He has been the leader of one "863" project of China and three National Natural Science Foundation of China (NSFC) projects, and core member of several important national projects, such as National 973 Basic Research Program of China. His current research interests include system security, trusted computing, virtualization, and cloud security.

JING TANG received the B.E. degree in intelligence science and technology from Central South University, Changsha, China, in 2018. She is currently pursuing the M.E. degree in cyberspace security with the Huazhong University of Science and Technology. Her current research interests include vulnerability detection and machine learning.

ZHIHAO ZHANG is currently pursuing the bachelor's degree in cyberspace security with the Huazhong University of Science and Technology, Wuhan, China. His current research interests include vulnerability detection and machine learning.

MINGQIAN SUN is currently pursuing the bachelor's degree in cyberspace security with the Huazhong University of Science and Technology, Wuhan, China. His current research interest includes software security.

HAI JIN received the Ph.D. degree in computer engineering from the Huazhong University of Science and Technology (HUST), Wuhan, China, in 1994, where he is currently a Cheung Kung Scholars Chair Professor of computer science and engineering. He is also the Chief Scientist of ChinaGrid, the largest grid computing project in China, and the Chief Scientists of National 973 Basic Research Program Project of Virtualization Technology of Computing System and Cloud Security. He has coauthored 22 books and published more than 700 research papers. His research interests include computer architecture, virtualization technology, cluster computing and cloud computing, peer-to-peer computing, network storage, and network security. He is a Fellow of the CCF and a member of the ACM. He received the Excellent Youth Award from the National Science Foundation of China, in 2001.