Measurement 152 (2020) 107139

Contents lists available at ScienceDirect

Measurement

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

A deep learning based static taint analysis approach for IoT software vulnerability location ☆

Weina Niu [a], Xiaosong Zhang [b,c,*], Xiaojiang Du [d], Lingyuan Zhao [b], Rong Cao [b], Mohsen Guizani [e]

[a] College of Cybersecurity, Sichuan University, Chengdu, Sichuan 610065, China
[b] Institute for Cyber Security, School of Computer Science and Engineering, University of Electronic Science and Technology of China, Chengdu, Sichuan 611731, China
[c] Cyberspace Security Research Center, Peng Cheng Laboratory, Shenzhen 518040, China
[d] Department of Computer and Information Sciences, Temple University, Philadelphia, PA 19122, USA
[e] College of Engineering, Qatar University, Doha 2713, Qatar

ARTICLE INFO

Article history:
Received 27 May 2019
Received in revised form 9 September 2019
Accepted 7 October 2019
Available online 16 October 2019

MSC 2010:
00-01
99-00

Keywords:
IoT software vulnerability location
Deep learning
Software patching
Static taint analysis

ABSTRACT

Computer system vulnerabilities, computer viruses, and cyber attacks are rooted in software vulnerabilities. Reducing software defects, improving software reliability and security are urgent problems in the development of software. The core content is the discovery and location of software vulnerability. However, traditional human experts-based approaches are labor-consuming and time-consuming. Thus, some automatic detection approaches are proposed to solve the problem. But, they have a high false negative rate. In this paper, a deep learning based static taint analysis approach is proposed to automatically locate Internet of Things (IoT) software vulnerability, which can relieve tedious manual analysis and improve detection accuracy. Deep learning is used to detect vulnerability since it considers the program context. Firstly, the taint from the difference file between the source program and its patched program selection rules are designed. Secondly, the taint propagation paths are got using static taint analysis. Finally, the detection model based on two-stage Bidirectional Long Short Term Memory (BLSTM) is applied to discover and locate software vulnerabilities. The Code Gadget Database is used to evaluate the proposed approach, which includes two types of vulnerabilities in C/C++ programs, buffer error vulnerability (CWE-119) and resource management error vulnerability (CWE-399). Experimental results show that our proposed approach can achieve an accuracy of 0.9732 for CWE-119 and 0.9721 for CWE-399, which is higher than that of the other three models (the accuracy of RNN, LSTM, and BLSTM is under than 0.97) and achieve a lower false negative rate and false positive rate than the other approaches.

1. Introduction

Over the next four years, 10 billion Internet of Things (IoT) and connected devices will be deployed worldwide according to the report from Strategy Analytics [1]. The popularity and rapid development of IoT technology have also brought new security and privacy risks, like IoT botnet, cryptocurrency mining and ransomware attack. Several papers (e.g., [2–5]) have studied related security issues. IoT devices are closer to the user state than traditional PC devices or server devices, and their associated privacy data or property data has a larger number than that of traditional devices, which makes attackers favor IoT devices. These attacks may shorten the lifetime of the battery in IoT network or ruin the energy supply system, which would influence the basic functions of the devices and even cause huge economic losses [6]. For example, the notorious Mirai botnet [7] exploits login vulnerabilities in unsecured IoT devices such as webcams and home routers has launched the largest DDoS attack known to date. Moreover, the autonomous working mechanism and limitations on energy resources of IoT devices make them vulnerable to energy resources exhaustion (ERE) attacks [8,9]. On the base of the attack method analysis, many research studies have been carried out on this topic. For example, Boubiche et al. [10] analyzed Sleep Deprivation, Barrage, Collision and Synchronization attacks at the intersection of the physical and data link layers. Goudar et al. [11] discussed

☆ Fully documented templates are available in the elsarticle package on CTAN.

* Corresponding author at: Institute for Cyber Security, School of Computer Science and Engineering, University of Electronic Science and Technology of China, Chengdu, Sichuan 611731, China.
E-mail address: johnsonzxs@uestc.edu.cn (X. Zhang).

https://doi.org/10.1016/j.measurement.2019.107139
0263-2241/

Denial-of-Sleep attacks in WSN, which were caused by manipulating network packets.

Applications, network protocols, operating systems, and cryptographic algorithms ultimately exist in the form of software. However, it is difficult to guarantee reliable and safe software development since the design and development of computer software requires high-intensity mental work and rich experience. Moreover, the number of software vulnerabilities registered in the Common Vulnerabilities and Exposures (CVE) [12] continues to grow since 1999 and reaches 14,714 in 2017, which is shown in Fig. 1. In addition, the US Department of Defense's Advanced Research Projects Agency (DARPA) has hosted the Cyber Grand Challenge (CGC) [13] since 2015 to improve the capabilities of a new generation of fully automated cyberspace defense systems. The five aspects of the CGC emphasize that all competitions are automated [13]. Therefore, the fully automated method is the solution to future cyber warfare.

Detecting and locating software vulnerabilities is the foundation for building a cyber defense system [14]. Many software vulnerability detection methods have been proposed, which can be divided into two categories, one is pattern-based [15] and the other is code similarity-based [16,17]. The first group of methods relies on human experts for building vulnerability feature database. Therefore, they are labor-consuming and sometimes error-prone. Moreover, they cannot discover the precise locations of vulnerabilities due to independent program representation. The second group of methods can solve the problem by representing each software in the abstract level, and these methods consider contextual information as well. However, they have a high false negative rate and false positive rate.

Deep learning has dramatically changed the way that computing devices handle human-centric content such as image, video, and audio [18]. The widespread use of IoT and Network Physics Systems (CPS) in the industry can benefit from the introduction of deep learning models. For example, images of production vehicles in the assembly line and their annotations are input into a deep learning system such as AlexNet, GoogLeNet, etc. to achieve visual inspection. Deep learning can also produce next-generation applications on IoT devices, which can perform complex sensing and recognition tasks [19,20].

However, to the best of our knowledge, it is not common to use deep learning to detect software vulnerabilities. Deep learning is mainly used for software defect prediction, like software language modeling [21], code cloning detection [22], API learning [23], binary function boundary recognition [24], and malicious URLs, file paths detection and registry keys detection [25], which is different from software vulnerabilities detection. Li et al.[26] used deep learning model BLSTM to detect software vulnerabilities and can achieve an accuracy of 0.949. The work is also used as a comparative experiment of our proposed method.

In response to the above reality and to reduce false positive rates and false negative rates, deep learning based static taint analysis approach is proposed in this paper. The designed IoT software vulnerability location system can enhance the automation level and accuracy of software vulnerability discovery and location, which uses the patch-based taint propagation path method and deep learning-based vulnerability discovery and location method.

Our contributions. This paper introduces deep learning based static taint analysis approach for IoT software vulnerability. Three are three of our main contributions as follows.

First, we propose three taint selection principles to determine the original taints. The first is to select the variables shared by the deleted and added lines in the diff file, the second is to select the parameters of the known vulnerability function or the ordinary function, and the third is to select the restricted variable in the if conditional statement.

Second, we propose the taint weight calculation method to select taint with high weight. A large number of initial taints are generated using the three taint selection strategies, but many initial taints do not actually trigger the vulnerabilities. In order to further improve the accuracy of the selected taints, further taint screening is performed in combination with the frequency of real taints.

Third, we develop the deep learning-based IoT software vulnerability location system and evaluate its effectiveness using the Code Gadget Database.

Paper organization. The rest of the paper is organized as follows. Section 2 introduces related work about software vulnerability detection. Section 3 presents some preliminaries. Section 4 elaborates the proposed deep learning-based vulnerability location approach in detail. Section 5 describes our experimental evaluation and results. Section 6 concludes the present paper.

2. Related work

In this section, how traditional software complexity and quality metrics (such as entropy) contribute to software vulnerability analysis and studies related to software vulnerability detection and location are discussed. Some latest development in related fields is also tracked.

There are three parameters to decide the software complexity, which include overall complexity, input and output complexity,

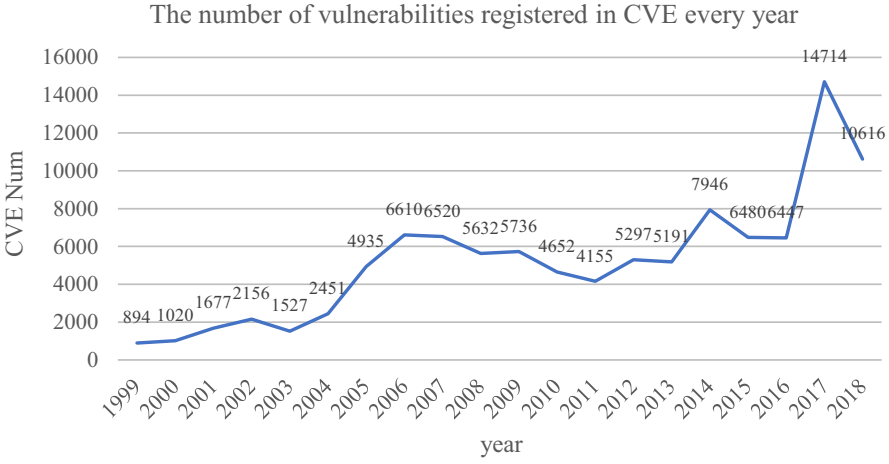

Fig. 1. The number of vulnerabilities registered in CVE per year.

and rectification complexity. The indicators of overall complexity include the number of code lines, the number of functions, the number of code lines declaring functions and variables, the complexity of some key algorithms, the complexity of circles and the number of recursive calls layers. The input and output complexity contains metrics such as global variables used by functions, parameters of functions, heaps and stacks of function calls. Intuitively, the rectification complexity is the number of code lines that are annotated.

The quality indicators of the software are as follows:

1) The number of bugs in each code segment/module/time period. Coverity and Checkmarx use this indicator to judge their ability to detect vulnerabilities.
2) Code coverage (the proportion and extent to which the source code was tested). HP Fortify uses this indicator to judge its ability to detect vulnerabilities.
3) Designing/Development constraints (number of methods/properties in a class).
4) Software complexity.

In summary, the software complexity and quality indicators contribute to the vulnerability analysis in three main points: prejudging and cost controlling for the limitations of vulnerability recurrence, adjusting and improving the vulnerability tracking solution, and evaluating and repairing of the vulnerability repair strategy.

Flawfinder [27] is an open-source code analysis tool, which primarily makes simple text pattern matching with a built-in database of C/C++ functions to discover well-known problems. However, Flawfinder has high false negative rates since it doesn't do control flow or data flow analysis. In order to improve universality, Rough Auditing Tool for Security (RATS) [28] provides a list of potential trouble spots with C, C++, Perl, PHP, and Python source code. It checks for risky built-in/library function calls by the rules of RATS. Unfortunately, RATS has high false negative rates and false positive rates since it performs only a rough analysis of source code. Moreover, manual inspection is still necessary under the aid of RATS. To support interactive programming environments in real-time, ITS4 [29] uses a parse tree generated with a context-free parser to represent the program. It breaks a non-preprocessed file into a series of lexical tokens and then matches them to the vulnerability database. However, ITS4 has high false positive rates since it cannot understand the program context.

CxSAST from Checkmarx [30] is an accurate and flexible source code analysis solution, which is fluent in all major languages. Checkmarx uses a unique lexical analysis technique and CxQL patent query technology to perform static analysis. However, Checkmarx has high false negative rates. In order to improve detection accuracy and reduce cost, Coverity [31] offers integrations with key development tools and CI/CD systems. Moreover, Coverity supports multiple programming languages and frameworks. Compared to other static analysis tools, Coverity has the following characteristics: providing deep, full path coverage accuracy; using interprocedural analysis. But, it also has high positive rates. For example, Coverity may report a risk when the pointer *pNext* does *pNext*++ operation without being assigned or assigned a value of *NULL*. However, if the *pNext* pointer is assigned in a while loop below $pNext = NULL$, this report can be ignored. To reduce time-consuming and effort-consuming, HP Fortify [32] statically analyzes the source code through the built-in five main analysis engines: data flow, semantics, structure, control flow, configuration flow, etc. Unfortunately, it cannot effectively locate the location of the vulnerability. The comparison of mainstream commercial static analysis tools is shown in Table 1.

Neuhaus et al. [33] have developed Vulture, which is used to mine a vulnerability database, a version archive and a code base, and map past vulnerabilities to components. Vulture [33] is able to predict vulnerabilities of new components based on their imports and function calls [34]. However, such fine-grained relationships of Vulture is still at the component level. Based on the empirical study of 3,241 Red Hat packages, Neuhaus et al. [35] used support vector machines on Red Hat dependency data to predict vulnerable packages [36]. To further reduce the vulnerability analysis granularity, Yamaguchi et al. [37] embedded code in a vector space and automatically determined API usage patterns using the machine learning. However, the false negative rate also exists. Yamaguchi et al. [38] extracted abstract syntax trees from the source code and searched for vulnerabilities based on the idea of vulnerability extrapolation, but they cannot identify vulnerabilities automatically. Grieco et al. [15] proposed a machine-learning-based approach to discover software vulnerability through lightweight static and dynamic features. Unfortunately, the test prediction error is high as well. Li et al. [39] presented an automatic software vulnerability detection system, Vulnerability Pecker (VulPecker). VulPecker [39] generated the signature of the target program and then detected vulnerability using code-similarity algorithms. However, the effectiveness of the approach needs to be further improved due to its some heuristics. Kim et al. [16] proposed a scalable approach for vulnerable code clone discovery, VUDDY, which leverages function-level granularity and a length-filtering technique to reduce the number of signature comparisons. However, VUDDY [16] focuses on the vulnerable discovery of code clone.

The academic community also has a lot of articles about vulnerability detection and location, which promotes the development of this field. Huang et al. [40] proposed a new automatic vulnerability classification model (TFI-DNN). The proposed TFI-DNN model outperformed others in accuracy, precision, and F1-score and performed well in recall rate. It was also superior to SVM, Na've Bayes and KNN on comprehensive evaluation indexes. Jurn et al. [41] proposed a Hybrid Fuzzing method based on a binary complexity analysis and introduced an automatic patch technique modifying the PLT/GOT table to translate vulnerable functions into safe functions. The experimental results showed the proposed model has good performance in open-source binaries. Spanos et al. [42] proposed a model combined with text analysis and multi-target classification techniques to estimate the vulnerability

Table 1
The comparison of mainstream commercial static analysis tools.

Tool name	Checkmarx [30]	Coverity [31]	HP Fortify [32]
Platform	Windows	Multi-platform	Multi-platform
Program language	Multi-language	C/C++, Java	Multi-language
Vulnerability types	Multiple	Multiple	Multiple
Development agency	Checkmarx	Coverity	HP
Release time	2003	2002	2012
Key technology	Lexical Analysis and CxQL Patent Query	SAT engine and software DNA map	Data flow, semantics, structure, control flow, configuration flow, etc.

characteristics. They considered the vulnerability characteristics as a vector of six targets and estimated these characteristics using multi-target classification. Experimental results showed that the proposed methodology could achieve comparable results. Aakanshi et al. [43] proposed a mathematical model to predict the bad smells using the Information Theory. Bad smells were collected using the detection tool from sub-components of the Apache Abdera project, and different measures of entropy (Shannon, Rényi, and Tsallis entropy) were used to identify bad smells. The experimental results showed that all three entropy approaches are sufficient to predict the bad smells in software. Madhu et al. [44] proposed bug dependency-based mathematical models by considering the summary description of bugs and comments submitted by users in terms of the entropy-based measures. The models mainly followed exponential, S-shaped or mixtures of both types of curves. But some improvement work could be done in the area of the summary_entropy and comment_entropymetric-based models using other project data to make it general.

In addition to the above scholars, there are still some people who have made their own contributions in this field or similar fields. StarWarsIV, Ali Hassan et al. [45] proposed Hybrid Adaptive Bandwidth and Power Algorithm, and Delay-tolerant Streaming Algorithm to significantly optimizes power drain, battery lifetime, standard deviation. Ali et al. [46] proposed an optimization scheme aiming at achieving the customer experience quality of vehicle Internet. Abdul et al. [47] evaluated the quality of service computing in health care applications, proposed AQCA algorithm which is more suitable for the quality of service computing, and analyzed the impact of each QoE parameter on medical data processing by estimating QoE perception. Sandeep et al. [48] improved and managed M-QoS by prioritizing telemedicine services using a decisive and intelligent tool called Analytic Hierarchy Process (AHP). Hina et al. [49] presented a detailed survey about how 5G has revolutionized medical healthcare with the help of IoT for enhancing quality and efficiency of the wearable devices. Also, state-of-the-art 5G-based sensor node architecture was proposed for the health monitoring of the patients with ease and comfort. Ali Hassan et al. [50] proposed a novel joint transmission power control (TPC) and duty-cycle adaptation based framework, adaptive energy-efficient transmission power control (AETPC) algorithm a Feedback Control-based duty-cycle algorithm and system-level battery and energy harvesting models to minimize charge and energy depletions of the wearable devices. Ali Hassan et al. [51] proposed a forward central dynamic and available approach (FCDAA), a system-level battery model, a data reliability model for edge AI-based IoT devices over hybrid TPC and duty-cycle network to use resources appropriately. Ali Hassan et al. [52] proposed a novel energy-efficient adaptive power control (APC) algorithm to overcome the problem that a constant transmission power and a typical conventional transmission power control (TPC) methods are not suitable choices for WBAN for the large temporal variations in the wireless channel. Muhammad et al. [53] put forward the Wireless Body Sensor Networks which provides ways to monitor individual activity in a variety of scenarios. YezhiLin et al. [54] developed an efficient, simple and unified way to increase the potential speed of the multicore-system. Chandio et al. [55] proposed a system, which is named integration of inter-connectivity of information system (i3) based on service-oriented architecture (SOA) with web services, to monitor and exchange student's information. Lodro et al. [56] proposed the channel modeling of 5G mmWave cellular communication for urban microcell which was simulated in LOS condition at operating frequency of 28 GHz with multiple antenna elements at transmitter and receiver. Different parameters affecting the channel had been considered in simulation using NYUSIM software.

3. Preliminaries

In this section, related preliminaries about our deep-learning-based IoT software vulnerability location approach are illustrated.

3.1. Static taint analysis

Static taint propagation analysis [57] refers to determining whether data can be transmitted from a tainted source to a taint convergence point by analyzing data dependencies between program variables without running and modifying the code.

The object of static taint analysis is usually the source code or intermediate representation of the program. The workflow of static taint analysis are described as follows: first, a call graph (CG) is constructed according to the function call relationship in the program; then, specific data stream propagation analysis is performed within the function or between functions according to different program characteristics. Common explicit stream taint propagation methods include direct assignment propagation, propagation through function (procedure) calls, and propagation through aliases (pointers).

In recent years, researchers have developed a number of tools to conduct taint analysis on other languages like java, but there are only a few tools available for C/C++. Some famous open-source tools like Saint [58] proposed in 2015 and Tanalysis [59], built as a plugin for the Frama-C platform, now are no longer available. Tools are still available such as Marcelo [60], which modifies the clang static analyzer to perform static taint analysis, but clang has disadvantages of not being able to analyze multiple source files, and it does not have access to the LLVM which can help with analysis. Lacking of an extensible and configurable static taint analysis tool is an open opportunity ignored by academia.

3.2. CNN-BLSTM

Neural networks [61] have achieved great success in image processing [62], speech recognition, and NLP [63], but they are rarely used in vulnerability detection. It means that many neural network models may not be suitable for vulnerability detection. Therefore, some principles are needed to guide the selection of neural network models for vulnerability detection. Whether the vulnerability is included in the code is determined by the context, so a neural network that is able to handle the context can be used for vulnerability detection [64]. The neural network used for NLP also needs to consider the context. It is feasible to use the deep learning model to conduct software vulnerability discovery and location [26]. The structure of Neural networks includes convolution layer and pooling layer.

Convolution layer: the input to each node in the convolutional layer is just a small piece of the upper layer of the neural network, which we usually call the kernel. The convolutional layer attempts to further analyze each small block in the neural network to obtain more abstract features. Assuming that $w^i_{x,y,z}$ is used to represent the i_{th} node in the output unit node matrix, $a_{x,y,z}$ is used to indicate the weight of filter input node (x,y,z), b^i is used to represent the offset term coefficient corresponding to the i_{th} output node, then the value of the i_{th} node in the identity matrix g(i) is defined as for formula (1).

$$g(i) = f\left(\sum_{x=1}^{a}\sum_{y=1}^{b}\sum_{z=1}^{c} a_{x,y,z} \times w^i_{x,y,z} + b^i\right) \tag{1}$$

Pooling layer: it can effectively reduce the size of the matrix, thus reducing the parameters in the final full connected laye. Using the

pooling layer can both speed up the calculation and prevent overfitting problems. The calculation of the pooling layer in the filter is not a weighted sum of nodes, but a simpler maximum or average operation.

Recurrent Neural Network (RNN) [65] is used to mine the time series information in the data and the deep representation of semantic information. It is often used in speech recognition, language modeling, machine translation, and timing analysis. RNN differs from ordinary fully connected neural networks in that the nodes between the hidden layers of the RNN are connected. The input of the hidden layer includes not only the output of the input layer but also the output of the hidden layer at the previous moment.

Long Short Term Memory (LSTM) [66] is a special type of RNN, which can learn long-term dependency information. LSTM is different from standard RNN, which has four different structures that interact in a very special way. LSTM is a special network structure with three "gate" structures. Bidirectional Long Short Term Memory (BLSTM) [67] uses a two-way structured LSTM model, taking into account the impact of context on the structure.

In addition, the RNN model has a Vanishing Gradient problem [68], which may lead to invalid model training. The Vanishing Gradient problem is solved with the idea of memory cells into RNNs (including LSTM and GRU), but LSTM is one-way and is not enough to detect software vulnerability (function parameters may be affected by the previous statement may also be affected by the following statements). Therefore, it is feasible to use the BLSM model to conduct software vulnerability discovery and location. To further improve detection accuracy, CNN-BLSTM neural network is applied in the paper. The input data size is $100 * 150$. After the convolution layer and the pooling layer, the data size is $9 * 128$. After the LSTM layer, the data size is 64. Finally, the data is classified by the fully connected layer. Fig. 2. shows the structure of CNN-BLSTM neural network, which has a convolution layer, a max pooling layer, a number of BLSTM layers, a fully connected (FC) layer and a softmax layer.

4. Deep-learning-based IoT software vulnerability location approach

The section introduces the proposed deep-learning-based IoT software vulnerability location approach. As is highlighted in Fig. 3, the proposed approach includes four components: patching comparison, static taint analysis, taint propagation paths transforming, and IoT software vulnerability location. Fig. 4 shows the specific technical process. There are six steps in our patching-based approach: using difflib to obtain the Diff file between the source code and the patched code; labeling taints according to the taint selection principles; generating taint propagation paths using static taint propagation; transforming taint propagation paths into symbolic representations; encoding the symbolic representations into vectors; applying the trained CNN-BLSTM neural network to locate two common types of IoT software vulnerability.

4.1. Patching comparison

The patching comparison obtains a Diff file with different marks by comparing the source code of the vulnerable software with the patched software. The Diff file is input to the static taint analysis module. The existing source code comparison tools include DiffMerge [69], Textdiff [70], Meld [71], Git diff [72] and so on. Most of these open-source tools have a graphical interface. If we call them directly in the source code, it will take a lot of time to operate manually. In addition, the source code of most tools has runtime errors, which is not easy to use. As the difflib package of python can achieve the same functions as these tools, this package is directly used in the paper to obtain Diff file with different marks.

4.2. Static taint analysis

As is highlighted in Fig. 5, the function of the static taint analysis module is to generate a taint propagation path by performing lexical analysis and grammar analysis on the Diff file. The module includes the following two tasks: (1) determining the taint according to taint selection principles (see below); (2) generating the taint propagation paths according to tainted sources.

The principles of taint selection are described as follows:

1. Selecting the common variables in the deleted and added rows;
2. Selecting the parameters of the known vulnerability function or the ordinary function;
3. Selecting the restricted variable in the if conditional statement.

According to the principles of taint selection, the appropriate taints are initially selected, but many initial taints do not actually trigger the vulnerabilities. In order to further improve the accuracy of the selected taints, we then rank taints based on the taint weight calculation method, which is as follows: 1) If a taint is a parameter of the CWE-119 or CWE-339 vulnerability correlation function, the tainted weight is 1; 2) If a taint is a parameter of the ordinary function, the tainted weight is 2; 3) If a taint is bound by an *if* statement, the tainted weight is 3; 4) Otherwise, the tainted weight is 4. Finally, we generate taint propagation paths based on static taint analysis and the line number at which the taint first appears.

4.3. Taint propagation paths transforming

There are two steps in this module. The first step is transforming taint propagation paths into symbolic representation, and the second step is encoding the symbolic representation into vectors. The output of the module is the input of IoT software vulnerability location module.

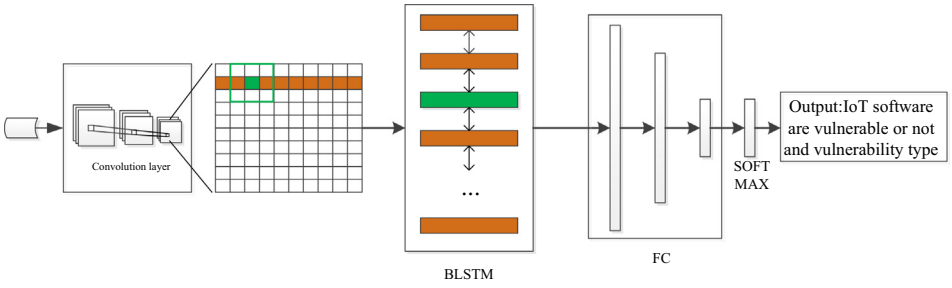

Fig. 2. A brief review of CNN-BLSM neural network.

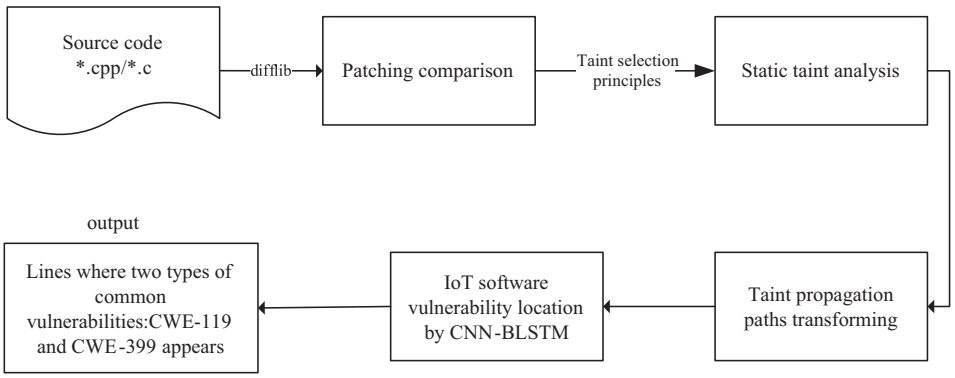

Fig. 3. Overview of our proposed approach.

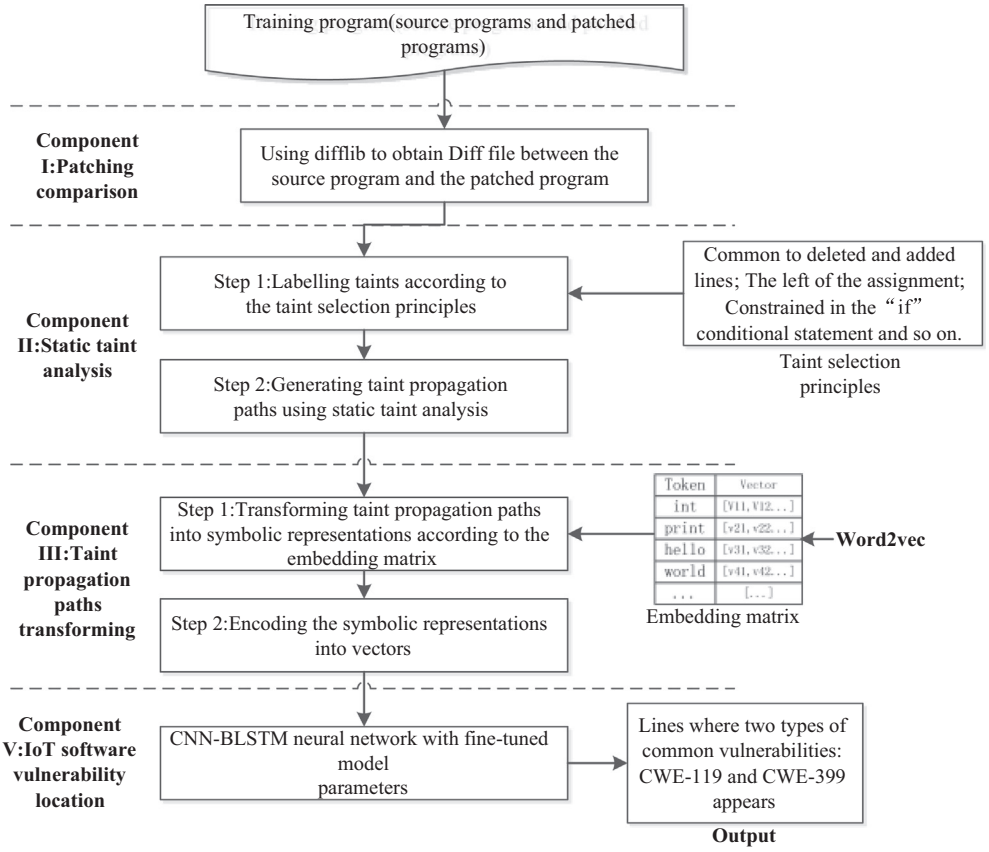

Fig. 4. Technique flow chart of our proposed approach.

4.3.1. Transforming symbolic representation

Processing the code segment is analogized to the Natural Language Processing (NLP) problem, which is necessary to segment the training data. The purpose of the word segmentation is to convert the text into a sequence of words that the model can read. Generating a word sequence requires two steps: 1: using the data set to generate a participle dictionary; 2. replacing each word in the original code snippet with the number of the word corresponding to the dictionary. The detailed steps are shown as follow.

Step 1, after filtering special symbols such as "!#$%&()* +,-./:;<=>?@[]^'{|}~ \t\n" from all the files, all words in the dataset are segmented and all words are extracted.

Step 2, generate a word breaker: the word frequency of all words in the data set is counted, and each word is assigned a number according to the word frequency order. Finally, a dictionary (vocabulary-word frequency order number) is formed.

Step 3, use the generation dictionary to replace words in all code snippets in the dataset with word numbers for model input. Take *print*(*hello world*) for example, whose result of filtering special symbols is *print hello world*. The input of model is [2,40,66], which is converted according to the dictionary {"int":1, "print":2, ..., "hello":40, ..., "world":66, ...}.

The purpose of this module is to transform the code block in the form of text type into a numeric type and populate it. *keras.preprocessing.text.tokenizer* is used to segment the input data and the input data is represented by a subscript sequence. Here are the implementation details of tokenizer: calculating the number of times each word appears in the program, sorting words according to the number of times the word appears from large to small, the first one is 1, and this is recursive. A numerical representation of the dataset sample is got by using *keras.preprocessing.text.text_to_word_sequence* participle. The spe-

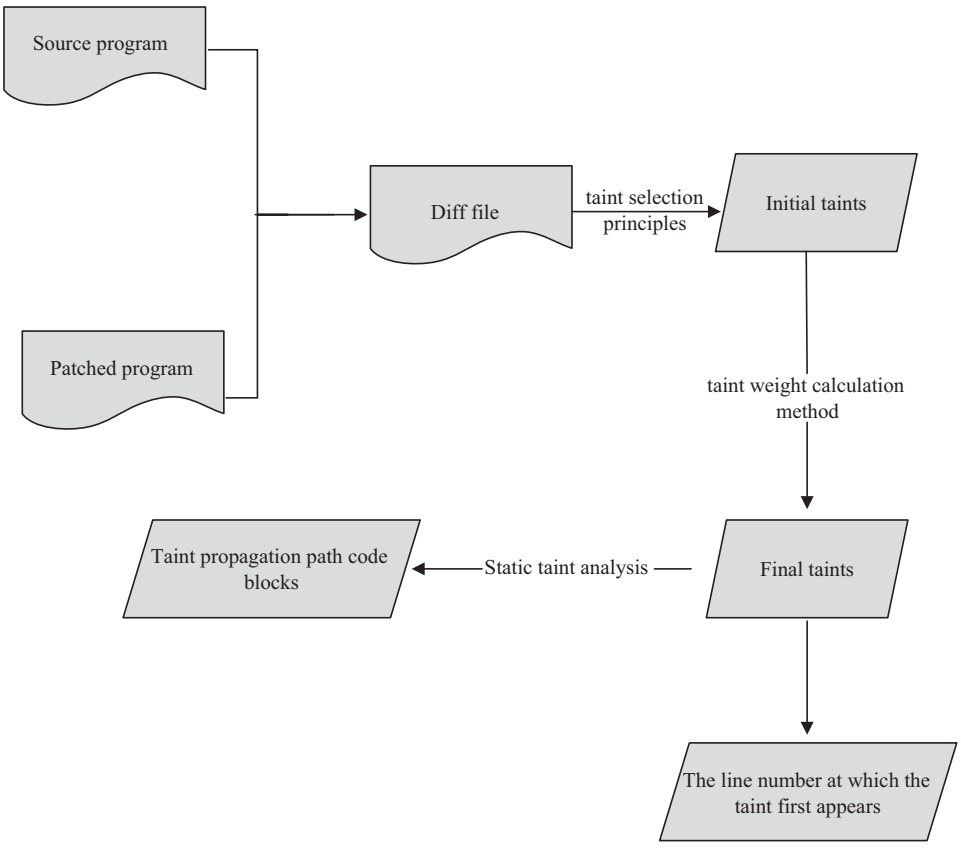

Fig. 5. The work flow of the module of Static taint analysis.

cial symbol in the participle process should be ignored, such as "!#$%&()*+,-./:;<=>?@[]^_'{|}~ \t\n".

4.3.2. Encoding vectors

The goal of the module is generating a word vector set using Word2Vec in the gensim package and embedding the processed data as a vector form. Word2Vec is an efficient tool to characterize words as real-value vectors, which includes two models: CBOW and SkipGram. CBOW is used in this paper, which uses distributed representation to map each word into a K-dimensional real number vector. The implementation of word2vec is essentially a two-layer deep neural network (DNN) that predicts adjacent words with input words.

Word2Vec is a model for learning semantic knowledge in an unsupervised way from a large amount of text corpus, which is widely used in NLP. Word2Vec actually uses the word vector to represent the semantic information of the word by learning the text. That is, the semantically similar words are close to each other in the space through an embedded space. Embedding is actually a mapping that maps words from the original space to the new multidimensional space. In other words, space, where the original word is located, is embedded in a new space. The working principle of embedding is shown in Fig. 6. As vectors encoded using the one-hot method are high-dimensional and sparse, embedding is chosen to solve the problem. Suppose we encounter a dictionary containing 2,000 words in NLP. When using one-hot encoding, each word is represented by a vector containing 2,000 integers, which not only takes up a lot of storage space but also cannot express the similarity between words and words. The embedding is to express the word "deep" with a fixed length vector [32,48,21,56,15]. However, not every word is replaced by a one-hot vector but instead is used to find the index of the vector in the embedded matrix. The following is an example. The training sample is a computational diagram of (input word: "ants", output word: "car").

4.4. IoT software vulnerability location

This module has two parts, the first part is the training phase and the second part is the testing phase.1) The training phase: Generating Diff file between the source program and patched pro-

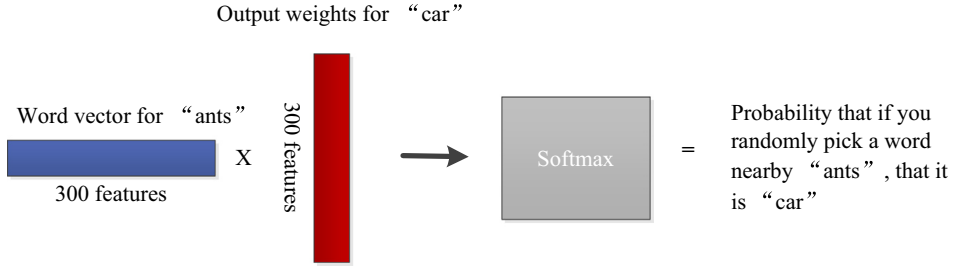

Fig. 6. The working principle of embedding [73].

gram, selecting taint according to taint selection principles and taint weight calculation method; getting taint propagation paths using static taint analysis; transforming taint propagation paths into certain symbolic representations; encoding taint propagation paths in the symbolic representation into vectors; training a CNN-BLSTM neural network. The trained CNN-BLSTM neural network is shown in Fig. 7.2) The detection phase: Given one or multiple target programs, Diff files are generated between source programs and patched programs. Taints are labeled using the taint selection principles and taint weight calculation method, and taint propagation paths are obtained based on the taints labeled in the previous step. Taint propagation path is transformed into symbolic representation and encoded by Word2Vec. At last, the lines of code where the vulnerability exists are located by applying the trained CNN-BLSM model.

The pseudo_code of our proposed method is as follows:

```
pseudo_code of our proposed method
Get the diff_file between the source file and the file after
   patching through difflib
for diff in diff_file:
   Remove comments (// &/*...*/) and import header files
   (beginning with #)
   Select variables that are shared in the difference row;
   Remove keywords
   if there are taints in the function line:
      if taint in sensitive function:
         for taint in sensitive function from VulDeePecker [26]:
            The vulnerability exists in the line
            Extract the taint propagation path of the variable
         break
      else:
         for taint in normal function
            Extracting the taint propagation path of the variable
         break
   elif taint in if statement:
      for taint in taint which in if statement
         Extracting the taint propagation path of the variable
      break
   else
      for taint in taints
         Extracting the taint propagation path of the variable
      break
if training
   Establish a CNN-BLSTM network, and use the extracted
   taint propagation path to train the network to obtain a
   model
elif testing
   Using the obtained model to detect whether the taint
   propagation path is a vulnerability and a related type;
```

5. Experimental evaluation and results

In this section, the dataset is illustrated to verify the validity of the proposed approach. The experimental setting and evaluation metrics are given, and then the experimental results are analyzed.

5.1. Experimental dataset

The proposed IoT software vulnerability location system focuses on two types of common vulnerabilities: buffer error (i.e., CWE-119) and resource management error (i.e., CWE-399) in this paper. Open source software programs in the experimental dataset come from the National Vulnerability Database (NVD) and the NIST Software Assurance Reference Dataset (SARD) project. The distribution of taint propagation paths from software programs in our experimental dataset is listed in Table 2, where TPP represents taint propagation paths. A taint propagation path is a number of lines of code with semantic correlation.

5.2. Experimental setting

The deep-learning-based IoT software vulnerability location system has been implemented in Python 3.6.5, Numpy 1.14.3, TensorFlow 1.8.0, Keras 2.1.6, Joblib 0.11, Gensim 3.4.0, Scikit-learn 0.19.1, Nltk 3.3, Reportlab 3.4.0, and all experiments are made using an off-the-shelf computer with Intel Core i7 at 3.7 GHz and 32 GB of RAM, GeForce GTX 1080. In order to evaluate the true positive rates and false positive rates of our software vulnerability location approach, we use the parameters in the evaluating experiment, which is shown in Table 3. In the experiment, the ratio of the training set to the test set is $4:1$.

The minimum length of taint propagation paths in the training set is 2, the largest length is 2,698, and the average length is 48. The reasonable parameter value of *max_sequence_len* is 100. For English text, the embedding length is usually 150. Moreover, the larger the embedding length is, the larger the computational overhead is. Therefore, the value of parameter *embedding_dim* is 150. The value of dropout is normally set to 0.5. However, because the number of training is 5, the result shows that there will be no overfitting. Thus, the value of the parameter *dropout* is set as 0.2. Regarding the parameter return_sequences, by setting this parameter to True, the result is output at each time step in the LSTM, and finally, all the outputs are stitched together. The final result contains each time step information, and the test result also indicates that it's better to set the value of the parameter True. Since the task is a two-category task, the output of the final model should use the sigmoid activation function, and the corresponding loss function should be binary_crossentropy. What's more, commonly used optimization algorithms such as sgd are prone to get into minimum values, so the adam optimization algorithm is used, and the speed is faster and the gradient descent process is smoother.

5.3. Evaluation metrics and experiment analysis

The following evaluation metrics are chosen to evaluate the proposed IoT software vulnerability location systems based on patching comparison. *TP* is the number of normal programs correctly labeled as normal, *FP* is the number of programs with vulnerabilities labeled as normal programs, *FN* is the number of normal programs labeled as programs with vulnerabilities, and *TN* is the number of programs with vulnerabilities correctly detected. FP_O and FN_O indicate the FP and FN of other deep learning models, like RNN, LSTM, and BLSTM. TP_{CB} and FN_{CB} indicate the FP and FN of CNN-BLSTM model, respectively.

$Accuracy = (TP + TN)/(TP + TN + FP + FN)$

$FP_I = (FP_O - TP_{CB})/TP_{CB}$

$FN_I = (FN_O - FN_{CB})/FN_{CB}$

In general, the higher the *Accuracy*, the better the recognition effect. The values of FP_I and FN_I are positive, indicating that CNN-BLSTM model is better; otherwise other models are better.

Some experiments are performed to evaluate the performance of the proposed approach for detecting IoT software CEW-399/CWE-119 vulnerabilities. The experiments mainly include the comparison of CNN-BLSTM with other deep learning models, such as RNN, LSTM, and BLSTM.

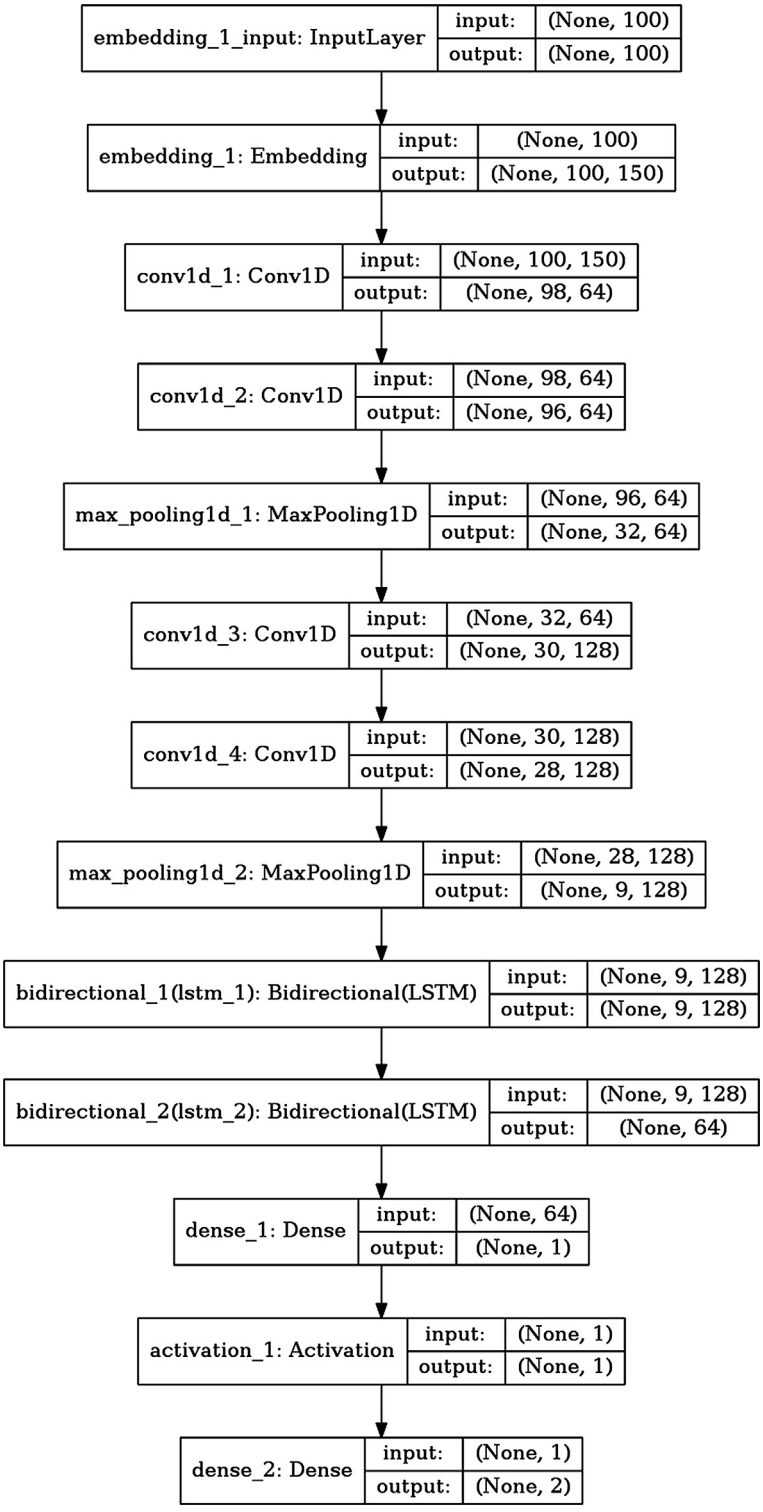

Fig. 7. The trained CNN-BLSTM neural network.

5.3.1. Experimental results of different models for CWE-399 identification

Table 4 shows the experimental results of different deep learning models for locating software vulnerability with CWE-399. Experimental results show that CNN-BLSTM-based classifier is more effective in software vulnerability with CWE-399 identification, whose value of *Accuracy* is 0.9721. Firstly, the convolutional neural network (CNN) is used to train the morphological character-level vector of code gadgets, and the word vector with semantic feature information is obtained from the large-scale

Table 2
The distribution of software programs in the experiment.

Category	Normal TPP	TPP with CWE-119	TPP with CWE-399
Number	43,913	10,439	7,285

Table 3
Experimental parameters settings.

Parameter	Description	Value
max_num_len	Size of the program word dictionary	20000
max_sequence_len	Maximum length of taint propagation path fragment	100
Word2Vec : *size*	Word vector dimension	150
fit : *batch_size*	The size of batch	32
LSTM : *dropout*&*Dense* : *dropout*	Parameters used to prevent overfitting	0.2
pool_size	Size of pool window	3
kernel_size	Size of convolution window	3
fit : *nb_epoch*	The frequency of training	5
LSTM : *return_sequences*	Whether data is returned at each time step	*True*
loss	Loss_function	*binary_crossentropy*
optimizer	Optimization function	*adam*

Table 4
Experimental results of different models for CWE-399 identification.

Model	RNN	LSTM	BLSTM	CNN-BLSTM
TN	1239	1362	1419	1416
FP	210	87	30	33
FN	159	76	107	89
TP	2769	2582	2821	2839
Accuracy	0.9157	0.9628	0.9687	0.9721
FP_I	5.364	1.636	−0.09	1
FN_I	0.786	−0.146	0.2	1

Table 5
Experimental results of different models for CWE-119 identification.

Model	RNN	LSTM	BLSTM	CNN-BLSTM
TN	5170	5763	5833	5954
FP	1183	482	454	306
FN	969	314	274	334
TP	16529	17292	17290	17257
Accuracy	0.9098	0.9666	0.9695	0.9732
FP_I	2.866	0.575	0.484	1
FN_I	1.9	−0.06	−0.18	1

background corpus training. Then the two are combined as input and constructed. BLSTM deep neural network model is suitable for software vulnerability identification, which can further improve recognition performance. The *Accuracy* of the RNN is 0.9157. RNN has the worst recognition effect because of its gradient disappearance problem. LSTM performs worse than BLSTM, mainly because the network structure of BLSTM constitutes an acyclic graph, and the output is obtained by taking into consideration the factors before and after. What's more, as seen from FP_I and FN_I, CNN-BLSTM has been greatly enhanced on RNN. Although there are negative numbers in FP_I of BLSTM and FN_I of LSTM, it is still improved overall.

5.3.2. Experimental results of different models for CWE-119 identification

Table 5 shows the experimental results of different deep learning models for locating software vulnerability with CWE-119. Experimental results show that CNN-BLSTM-based classifier has a better performance in software vulnerability with CWE-119 identification, whose value of *Accuracy* is 0.9732. The *Accuracy* of the RNN is 0.9098. RNN has the worst recognition effect because of its gradient disappearance problem. LSTM performs worse than BLSTM, mainly because BLSTM considers the factors before and after. As can be seen from Table 5, the CNN-BLSTM model has a lower false positive rate than other deep learning models, but its false negative rate is indeed higher than the LSTM and BLSTM models. What's more, as seen from FP_I and FN_I, CNN-BLSTM has been greatly enhanced on RNN. Although there are negative numbers in FN_I of LSTM and FN_I of BLSTM, the improvement in FP_I is greater, and the CNN-LSTM model is still better overall.

Finally, we trained a total of 31,802 samples using CNN-BLSTM and spent 3097.2 seconds. And we tested a total of 7950 samples using CNN-BLSTM and spent 34.2 seconds.

6. Conclusion

In recent years, various kinds of commercial software have been frequently exposing vulnerabilities, which seriously affect the enterprise's security. Thus, the security of third-party applications has received much attention. Existing dynamic detection methods consuming a lot of CPU resources, and the level of automation is low. This work uses static analysis and deep learning algorithms to automatically locate vulnerabilities. The proposed approach generates the Diff file between source code and patched program, labels taint sources according to the designed taint selection principles, obtains the lines where taints first appear and taint propagation paths using static taint analysis, transforms taint propagation paths into symbolics, encodes symbolic into vectors, discovers CWE-119/CWE-399 vulnerabilities based on trained CNN-BLSM model and finds their lines. The vulnerability locator based on deep learning is evaluated on a dataset consisting of

17,725 programs with vulnerabilities and 43,913 benign programs. Experimental results show that the proposed approach can achieve an accuracy of 0.9732 for CWE-119 and 0.9721 for CWE-399, which is higher than that of the other three models (the accuracy of RNN, LSTM, and BLSTM is under than 0.97).

In the future, our work can be applied to industrial control security, smart car security, smart home security, and other fields to ensure the safety of the Internet of Things equipment. Our work can be used as a detection system to detect these devices before they leave the factory, or as a chip embedded in the Internet of Things device to detect.

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

We thank the anonymous reviewers for their comments that helped us improve the paper. This work was supported in part by the National Key R&D Plan under Grant CNS 2016QY06X1205, in part by the Basic Research Business Fees of Central Colleges under Grant CNS 20826041B4252, in part by the National Natural Science Foundation (NSFC) under Grant CNS 61572115, and in part by the Science and Technology Project of State Grid Corporation of China under Grant CNS 522722180007. Any opinions, findings, conclusions or recommendations expressed in this material are those of the authors and do not reflect the views of the funding agencies.

References

[1] D. Mercer, Smart home will drive internet of things to 50 billion devices, says strategy analytics, 2017.https://www.strategyanalytics.com/strategy-analytics/news/strategy-analytics-press-releases/2017/10/26/smart-home-will-drive-internet-of-things-to-50-billion-devices-says-strategy-analytics..
[2] X. Du, M. Guizani, Y. Xiao, H.-H. Chen, Transactions papers a routing-driven elliptic curve cryptography based key management scheme for heterogeneous sensor networks, IEEE Trans. Wireless Commun. 8 (3) (2009) 1223–1229.
[3] Y. Xiao, V.K. Rayi, B. Sun, X. Du, F. Hu, M. Galloway, A survey of key management schemes in wireless sensor networks, Comput. Commun. 30 (11–12) (2007) 2314–2341.
[4] X. Du, Y. Xiao, M. Guizani, H.-H. Chen, An effective key management scheme for heterogeneous sensor networks, Ad Hoc Netw. 5 (1) (2007) 24–34.
[5] X. Du, H.-H. Chen, Security in wireless sensor networks, IEEE Wirel. Commun. 15 (4) (2008) 60–66.
[6] A. Laszka, A. Dubey, M. Walker, D. Schmidt, Providing privacy, safety, and security in IOT-based transactive energy systems using distributed ledgers, in: Proceedings of the Seventh International Conference on the Internet of Things, ACM, 2017, p. 13.
[7] C. Kolias, G. Kambourakis, A. Stavrou, J. Voas, Ddos in the IOT: Mirai and other botnets, Computer 50 (7) (2017) 80–84.
[8] A.T. Capossele, V. Cervo, C. Petrioli, D. Spenza, Counteracting denial-of-sleep attacks in wake-up-radio-based sensing systems, in: 2016 13th Annual IEEE International Conference on Sensing, Communication, and Networking (SECON), IEEE, 2016, pp. 1–9.
[9] T. Farzana, A. Babu, A light weight plgp based method for mitigating vampire attacks in wireless sensor networks, Int. J. Eng. Comput. Sci. 3 (7) (2014).
[10] D.E. Boubiche, A. Bilami, A defense strategy against energy exhausting attacks in wireless sensor networks, J. Emerging Technol. Web Intell. 5 (1) (2013) 18–27.
[11] C. Goudar, S. Kulkarni, Mechanisms for detecting and preventing denial of sleep attacks and strengthening signals in wireless sensor networks, Int. J. Emerg. Res. Manage. Technol. 4 (6) (2013).
[12] CVE, Cve details, 2017.https://www.cvedetails.com/browse-by-date.php..
[13] D. Fraze, Cyber grand challenge (CGC), 2017.https://www.darpa.mil/program/cyber-grand-challenge..
[14] W.-C. Lin, S.-W. Ke, C.-F. Tsai, CANN: an intrusion detection system based on combining cluster centers and nearest neighbors, Knowl.-based Syst. 78 (2015) 13–21.
[15] G. Grieco, G.L. Grinblat, L. Uzal, S. Rawat, J. Feist, L. Mounier, Toward large-scale vulnerability discovery using machine learning, in, in: Proceedings of the Sixth ACM Conference on Data and Application Security and Privacy, ACM, 2016, pp. 85–96.
[16] S. Kim, S. Woo, H. Lee, H. Oh, VUDDY: a scalable approach for vulnerable code clone discovery, in: IEEE Symposium on Security and Privacy (SP), IEEE, 2017, pp. 595–614.
[17] H. Sajnani, V. Saini, J. Svajlenko, C.K. Roy, C.V. Lopes, Sourcerercc: scaling code clone detection to big-code, in: 2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE), IEEE, 2016, pp. 1157–1168.
[18] F. Wu, J. Wang, J. Liu, W. Wang, Vulnerability detection with deep learning, in: 2017 3rd IEEE International Conference on Computer and Communications (ICCC), IEEE, 2017, pp. 1298–1302.
[19] M. Mohammadi, A. Al-Fuqaha, S. Sorour, M. Guizani, Deep learning for IOT big data and streaming analytics: a survey, IEEE Commun. Surv. Tutorials 20 (4) (2018) 2923–2960.
[20] H. Li, K. Ota, M. Dong, Learning IOT in edge: deep learning for the internet of things with edge computing, IEEE Network 32 (1) (2018) 96–101.
[21] M. White, C. Vendome, M. Linares-Vásquez, D. Poshyvanyk, Toward deep learning software repositories, in: Proceedings of the 12th Working Conference on Mining Software Repositories, IEEE Press, 2015, pp. 334–345.
[22] M. White, M. Tufano, C. Vendome, D. Poshyvanyk, Deep learning code fragments for code clone detection, in: Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, ACM, 2016, pp. 87–98.
[23] X. Gu, H. Zhang, D. Zhang, S. Kim, Deep api learning, in: Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, ACM, 2016, pp. 631–642.
[24] E.C.R. Shin, D. Song, R. Moazzezi, Recognizing functions in binaries with neural networks, in: 24th {USENIX} Security Symposium ({USENIX} Security 15), 2015, pp. 611–626.
[25] J. Saxe, K. Berlin, expose: a character-level convolutional neural network with embeddings for detecting malicious URLS, file paths and registry keys, arXiv preprint arXiv:1702.08568 (2017) 1–18..
[26] Z. Li, D. Zou, S. Xu, X. Ou, H. Jin, S. Wang, Z. Deng, Y. Zhong, Vuldeepecker: A deep learning-based system for vulnerability detection, arXiv preprint arXiv:1801.01681 (2018) 1–15..
[27] D.A. Wheeler, Flawfinder, 2017.https://www.dwheeler.com/flawfinder/..
[28] S.S. Solutions, Rough auditing tool for security (rats), 2017. https://code.google.com/archive/p/rough-auditing-tool-for-security/..
[29] J. Viega, J.-T. Bloch, Y. Kohno, G. McGraw, ITS4: a static vulnerability scanner for C and C++ code, in: Proceedings 16th Annual Computer Security Applications Conference (ACSAC'00), IEEE, 2000, pp. 257–267.
[30] Checkmarx, Securing uncompiled code with cxsast, 2017.https://www.checkmarx.com/products/static-application-security-testing/..
[31] Coverity, Coverity scan static analysis, 2017.https://scan.coverity.com/..
[32] HP, Fortify static code analyzer: Static application security testing, 2017. https://software.microfocus.com/es-es/products/static-code-analysis-sast/overview..
[33] S. Neuhaus, T. Zimmermann, The beauty and the beast: vulnerabilities in red hat's packages, USENIX Annual Technical Conference (2009).
[34] G. Abaei, A. Selamat, H. Fujita, An empirical study based on semi-supervised hybrid self-organizing map for software fault prediction, Knowl.-Based Syst. 74 (2015) 28–39.
[35] S. Neuhaus, T. Zimmermann, C. Holler, A. Zeller, Predicting vulnerable software components, in: ACM Conference on computer and communications security, Citeseer, 2007, pp. 529–540.
[36] S.S. Rathore, S. Kumar, Linear and non-linear heterogeneous ensemble methods to predict the number of faults in software systems, Knowl.-Based Syst. 119 (2017) 232–256.
[37] F. Yamaguchi, F. Lindner, K. Rieck, Vulnerability extrapolation: assisted discovery of vulnerabilities using machine learning, in, in: Proceedings of the 5th USENIX conference on Offensive technologies, USENIX Association, 2011, pp. 13–13.
[38] F. Yamaguchi, M. Lottmann, K. Rieck, Generalized vulnerability extrapolation using abstract syntax trees, in: Proceedings of the 28th Annual Computer Security Applications Conference, ACM, 2012, pp. 359–368.
[39] J. Li, M.D. Ernst, CBCD: cloned buggy code detector, in: Proceedings of the 34th International Conference on Software Engineering, IEEE Press, 2012, pp. 310–320.
[40] G. Huang, Y. Li, Q. Wang, J. Ren, Y. Cheng, X. Zhao, Automatic classification method for software vulnerability based on deep neural network, IEEE Access 7 (2019) 28291–28298.
[41] J. Jurn, T. Kim, H. Kim, An automated vulnerability detection and remediation method for software security, Sustainability 10 (5) (2018) 1652.
[42] G. Spanos, L. Angelis, A multi-target approach to estimate software vulnerability characteristics and severity scores, J. Syst. Softw. 146 (2018) 152–166.
[43] A. Gupta, B. Suri, V. Kumar, S. Misra, T. Blažauskas, R. Damaševičius, Software code smell prediction model using Shannon, Rényi and Tsallis entropies, Entropy 20 (5) (2018) 372.
[44] M. Kumari, A. Misra, S. Misra, L. Fernandez Sanz, R. Damasevicius, V. Singh, Quantitative quality evaluation of software products by considering summary and comments entropy of a reported bug, Entropy 21 (1) (2019) 91.
[45] A.H. Sodhro, S. Pirbhulal, Z. Luo, V.H.C. de Albuquerque, Towards an optimal resource management for IOT based green and sustainable smart cities, J. Cleaner Prod. 220 (2019) 1167–1179.
[46] A.H. Sodhro, Z. Luo, G.H. Sodhro, M. Muzamal, J.J. Rodrigues, V.H.C. de Albuquerque, Artificial intelligence based QOS optimization for multimedia

communication in IOV systems, Future Gener. Comput. Syst. 95 (2019) 667–680.
[47] A.H. Sodhro, A.S. Malokani, G.H. Sodhro, M. Muzammal, L. Zongwei, An adaptive QOS computation for medical data processing in intelligent healthcare applications, Neural Comput. Appl. (2019) 1–12.
[48] A.H. Sodhro, F.K. Shaikh, S. Pirbhulal, M.M. Lodro, M.A. Shah, Medical-qos based telemedicine service selection using analytic hierarchy process, in: Handbook of Large-Scale Distributed Computing in Smart Healthcare, Springer, 2017, pp. 589–609.
[49] H. Magsi, A.H. Sodhro, F.A. Chachar, S.A.K. Abro, G.H. Sodhro, S. Pirbhulal, Evolution of 5g in internet of medical things, in: 2018 International Conference on Computing, Mathematics and Engineering Technologies (iCoMET), IEEE, 2018, pp. 1–7.
[50] A.H. Sodhro, S. Pirbhulal, G.H. Sodhro, A. Gurtov, M. Muzammal, Z. Luo, A joint transmission power control and duty-cycle approach for smart healthcare system, IEEE Sens. J. (2018), 1–1.
[51] A.H. Sodhro, S. Pirbhulal, V.H.C. de Albuquerque, Artificial intelligence driven mechanism for edge computing based industrial applications, IEEE Trans. Industr. Inf. (2019) 4235–4243.
[52] A.H. Sodhro, Y. Li, M.A. Shah, Energy-efficient adaptive transmission power control for wireless body area networks, IET Commun. 10 (1) (2016) 81–90.
[53] M. Muzammal, R. Talat, A.H. Sodhro, S. Pirbhulal, A multi-sensor data fusion enabled ensemble approach for medical data from body sensor networks, Inf. Fusion 53 (2020) 155–164.
[54] Y. Lin, X. Jin, J. Chen, A.H. Sodhro, Z. Pan, An analytic computation-driven algorithm for decentralized multicore systems, Future Gener. Comput. Syst. 96 (2019) 101–110.
[55] A.A. Chandio, D. Zhu, A.H. Sodhro, M.U. Syed, An implementation of web services for inter-connectivity of information systems, arXiv preprint arXiv:1407.8320 (2014) 1–7..
[56] M.M. Lodro, N. Majeed, A.A. Khuwaja, A.H. Sodhro, S. Greedy, Statistical channel modelling of 5G mmWave MIMO wireless communication, in: 2018 International Conference on Computing, Mathematics and Engineering Technologies (iCoMET), IEEE, 2018, pp. 1–5.
[57] I. Medeiros, N. Neves, M. Correia, Detecting and removing web application vulnerabilities with static analysis and data mining, IEEE Trans. Reliab. 65 (1) (2016) 54–69.
[58] X.N. Noundou, Saint: Simple static taint analysis tool users manual, 2015. https://archive.org/details/saint..
[59] djn3m0, tanalysis, 2015.https://github.com/djn3m0/tanalysis..
[60] M. Arroyo, F. Chiotta, F. Bavera, An user configurable clang static analyzer taint checker, in: 35th International Conference of the Chilean Computer Science Society (SCCC), IEEE, 2016, pp. 1–12.
[61] J. Schmidhuber, Deep learning in neural networks: an overview, Neural Networks 61 (2015) 85–117.
[62] H. Xu, C. Huang, D. Wang, Enhancing semantic image retrieval with limited labeled examples via deep learning, Knowl.-Based Syst. 163 (2019) 252–266.
[63] H. Peng, Y. Ma, Y. Li, E. Cambria, Learning multi-grained aspect target sequence for chinese sentiment analysis, Knowl.-Based Syst. 148 (2018) 167–176.
[64] J. Leng, Q. Chen, N. Mao, P. Jiang, Combining granular computing technique with deep learning for service planning under social manufacturing contexts, Knowl.-Based Syst. 143 (2018) 295–306.
[65] H. Liu, B. Lang, M. Liu, H. Yan, CNN and RNN based payload classification methods for attack detection, Knowl.-Based Syst. (2018), S0950705118304325.
[66] G. Lin, J. Zhang, W. Luo, L. Pan, Y. Xiang, Poster Vulnerability discovery with function representation learning from unlabeled projects, in: Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, ACM, 2017, pp. 2539–2541.
[67] Z. Li, D. Zou, S. Xu, H. Jin, Y. Zhu, Z. Chen, S. Wang, J. Wang, Sysevr: a framework for using deep learning to detect software vulnerabilities, arXiv preprint arXiv:1807.06756 (2018) 1–13..
[68] R. Jozefowicz, W. Zaremba, I. Sutskever, An empirical exploration of recurrent network architectures, in: International Conference on Machine Learning, 2015, pp. 2342–2350.
[69] L. SourceGear, Diffmerge,https://sourcegear.com/diffmerge/..
[70] Textdiff,http://www.angusj.com/delphi/textdiff.html..
[71] Textdiff,http://meldmerge.org/..
[72] Git diff,https://www.atlassian.com/git/tutorials/saving-changes/git-diff..
[73] C. McCormick, Word2vec tutorial – the skip-gram model, 2017.http://mccormickml.com/2016/04/19/word2vec-tutorial-the-skip-gram-model/..