Hindawi
Security and Communication Networks
Volume 2020, Article ID 8858010, 16 pages
https://doi.org/10.1155/2020/8858010

Review Article

A Survey of Automatic Software Vulnerability Detection, Program Repair, and Defect Prediction Techniques

Zhidong Shen and Si Chen

*Key Laboratory of Aerospace Information Security and Trusted Computing, Ministry of Education,
School of Cyber Science and Engineering, Wuhan University, Wuhan, Hubei 430079, China*

Correspondence should be addressed to Zhidong Shen; shenzd@whu.edu.cn

Received 19 June 2020; Revised 13 August 2020; Accepted 11 September 2020; Published 30 September 2020

Academic Editor: Luigi Coppolino

Open source software has been widely used in various industries due to its openness and flexibility, but it also brings potential software security problems. Together with the large-scale increase in the number of software and the increase in complexity, the traditional manual methods to deal with these security issues are inefficient and cannot meet the current cyberspace security requirements. Therefore, it is an important research topic for researchers in the field of software security to develop more intelligent technologies to apply to potential security issues in software. The development of deep learning technology has brought new opportunities for the study of potential security issues in software, and researchers have successively proposed many automation methods. In this paper, these automation technologies are evaluated and analysed in detail from three aspects: software vulnerability detection, software program repair, and software defect prediction. At the same time, we point out some problems of these research methods, give corresponding solutions, and finally look forward to the application prospect of deep learning technology in automated software vulnerability detection, automated program repair, and automated defect prediction.

1. Introduction

With the rapid development of information technology, software is playing an important role in various aspects all over the world, such as the economy, military, and society. At the same time, potential security issues in software are becoming an emerging worldwide challenge. Software vulnerabilities are one of the root causes of security problems. High-skilled hackers can exploit the software vulnerabilities to do a lot of harmful things according to their own wishes, such as stealing the private information of users and halting the crucial equipment.

In today's multiuser continuous interaction environment, if a hacker uses a certain time node to launch an attack whose loss and cost are difficult to predict. According to the statistics released by the Common Vulnerabilities and Exposures (CVE) organization [1], the number of software vulnerabilities discovered in 2000 was less than 4600 while the number of vulnerabilities currently covered almost nearly 20000. It can be seen from Figure 1 that the number of vulnerabilities has reached its peak at the past three years, which undoubtedly increases the threats faced by many computer users using network services. Therefore, it is crucial to security researchers to discover and fix problems of the software in a timely manner.

Potential security problems of software are not only security problems faced by software developers but also related to the development of national network security. In the past few years, the phenomenon of cyber attacks on the industrial and commercial has continued to increase, and the security threats faced by enterprises and individuals have also increased. The ransomware "WannaCry," which appeared in 2017, is the cyberattack that has the worst impact, the widest coverage, and the most serious consequences in recent years. Therefore, the need for automated, scalable, machine-speed vulnerability detection, program

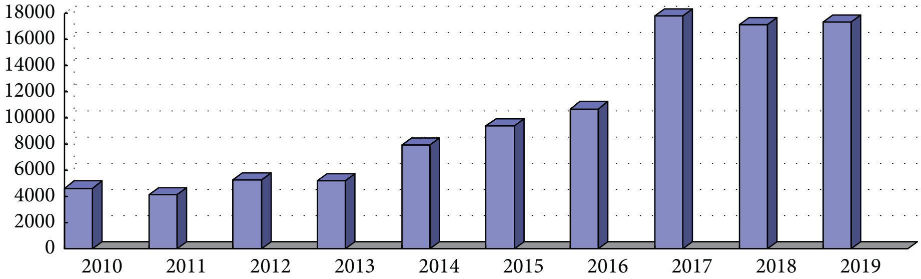

Figure 1: Number of CVE vulnerabilities over the years.

patching, and defect prediction techniques are becoming urgent when facing severe challenges posed by network security issues.

In 2016, the CGC held by DARPA left a profound impact on us. Judging from the CGC conference, machines can indeed replace part of the work of human white hats, and even in some aspects, such as operating speed, machines have a natural advantage over humans. Especially, in recent years, the rapid development of deep learning technology has given us more opportunities for machine to solve software security problems more intelligently.

In this article, we review the application of deep learning techniques in software security research, discuss the efforts of academia and related researchers in software security research, and look forward to the opportunities and challenges that deep learning technology faces in the field of software security.

To summarize the work of this paper, the key contribution is three-folds:

> Firstly, we review the latest research progress of deep learning technology in software vulnerability detection, program patching, and defect prediction. Tables 1 and 2 give an overview of these techniques.
>
> Secondly, we focus on the advantages and disadvantages of each technology from the aspects of software vulnerability detection, program patching, and defect prediction and propose ideas and solutions for future research.

At last, we look forward to the opportunities and challenges faced by existing automatic vulnerability detection, automatic program patching, and automatic defect prediction technologies and provide some reference value of future researchers.

2. Automatic Software Vulnerability Detection

Traditional vulnerability research methods often require security researchers to have professional knowledge and rich practical experience, which is less versatile and has limited efficiency. At present, the application of deep learning and natural language processing technologies can intelligently process vulnerability information to assist security vulnerability research and improve the efficiency of security vulnerability mining. The binary vulnerability detection method has high detection accuracy and wide practicability, but is difficult to trace the structure information and type information about the upper level code of the program. At present, the binary code vulnerability detection uses disassembly operations to disassemble binary code into assembly instruction codes, from which we use program analysis technology to extract key vulnerability code information that are vectorized to input into neural network for training. According to the static analysis technology, the vulnerability processing method is further divided into a vulnerability detection method based on code similarity and a vulnerability detection method based on code pattern.

2.1. Code Similarity-Based Vulnerability Detection. Code similarity vulnerability detection is also called clone detection, whose core idea is that similar program code has similar software vulnerabilities. In the process of continuous development of code clone detection technology, researchers have defined four common clone types: Type-1 (T1), Type-2 (T2), Type-3 (T3), and Type-4 (T4) [2]. Type-1 (T1) clones are identical code fragments which may contain variations in whitespace, layouts, or comments. Type-2 (T2) clones are code fragments allowing variations in identifiers, literals, types, whitespace, layouts, and comments. Compared with Type-2 clones, Type-3 (T3) clones allow extra modifications such as changed, added, or removed statements. Type-4 (T4) clones further include semantically equivalent but syntactically different code fragments.

2.2. Grammar-Based Clone Detection Methods. In the process of clone detection, the representation of source code determines the upper limit of information extraction, which further limits the model design and the selection of similarity measurement algorithms and ultimately affects the detection effect. The traditional text-based representation method, process data only involving the removal of comments and spaces in the program code, is mainly based on text similarity measurement methods to detect text-level clones. However, the tree-based representation method analyses the program code through the interpreter to process data, whose measurement algorithm takes the more program structures information into account to detect code clones from the grammatical level.

At present, many scholars have described the development of code clone detection technology in the field of software security from the dimensions of code

Table 1: Summary of deep learning technologies on automatic vulnerability detection, automatic program patching, and automatic defect prediction.

	Method type	Advantage	Disadvantage
Automatic vulnerability detection	Code similarity-based vulnerability detection	Source code-based detects multiple clone types; binary code-based achieves higher detection accuracy	False negative rate high (source); analytical complexity (binary)
	Code pattern-based vulnerability detection	Static method achieves higher code; coverage dynamic method detects faster	Lack run-time information (static); low code coverage (dynamic)
Automatic program patching	Grammar-based program patching	Token-based method error analysis; simple text-based method generates higher quality patches	Poor patches interpretability (token); unacceptable program behavior (tex)
	Semantic-based program patching	Static method achieves good repair effect dynamic method and accurately captures program behavior	Limited type of repair (static); high cost (dynamic)
Automatic defect prediction	Within-project defect prediction	End-to-end implementation, accurate prediction of defective program modules	Poor expansion
	Crossproject defect prediction	Effectively integrates dataset resources to better promote new project development practices	Excessive code feature extraction granularity
	Just-in-time defect prediction	Earlier identifies faulty modules and fine-grained analysis, efficiently identifies the number of defects	Lack of extensive training data to train the model

Table 2: Various feature parameters selected of deep learning technology on software vulnerability detection, program repair, and defect prediction.

Method type	CNN network	RNN network	DNN network
Code similarity vul detection	Iteration, dropout, hidden_layer, gradient_rate	hidden_layer, network depths, dropout, batch_size, iteration	hidden_layer, iteration, dropout, layer_size, learning_rate
Code pattern vul detection	filter_size, filter_num, hidden_layer	Dropout, batch_size, iteration, learning_rate, vector_dimension	—
Grammer program patching	—	learning_rate, batch_size, hidden_layer, token_length, hidden_unit, iteration, embedding_size, gradient_optimizer	—
Semantic program patching	—	hidden_unit, gradient_optimizer, vector_dimension, learning_rate, iteration, dropout, hidden_dimension	—
Within-project defect prediction	filter_num, filter_size, hidden_node, batch_size, iteration, embedding_dimension	Dropout, vector_dimension, hidden_layer, hidden_node, batch_size, iteration, learning_rate	hidden_layer, hidden_node, iteration, learning_rate, gradient_optimizer, batch_size
Crossproject defect prediction	filter_size, filter_num, learning_rate, vector_size	Dropout, hidden_layer, hidden_node, batch_size	Iteration, hidden_layer, hidden_node
Crossproject defect prediction	—	—	hidden_layer, hidden_node, activation_function

representation, model design, similarity measurement algorithms, and performance metrics. However, these works mainly summarize the early code clone detection methods and rarely involve the application of deep learning technology in clone detection technology. This article mainly introduces the research progress of deep learning technology in code clone detection technology in recent years and elaborates the related technology from the grammatical level and the semantic level.

The code representation on the syntax level mainly considers the syntax rules of the program source code. Usually, an abstract syntax tree (AST) is constructed to hierarchically display the structure information of the program, and then further processing for detecting. At present, the method based on deep learning intercepts key node information by constructing abstract syntax tree, transforms it into feature vector or hash values, then selects a suitable neural network model for training, and finally uses the model to achieve clone comparison.

Table 3 shows the technical implementation and technical characteristics of deep learning in grammar cloning detection. The current mainstream method is to characterize program code based on AST. Marastoni et al. have implemented T4 type code clone detection based on the image.

Literature [3] proposed a sentence-based deep learning code clone detection method and defined eight token types. First, use ANTLR [7] to parse each method and extract C1–C3, and use the Eclipse ASTParser [8] to create an abstract syntax tree for each method to further extract C4–C8 tokens. Then, calculate the frequency of each type of

Table 3: Technical characteristics of grammar-based clone detection.

System/writer	Data preprocessing	Code representation	Network	Clone types	Classification object
CCLearner [3]	Use ANTLR and ASRParser to parse each method	Token AST	DNN	T1–T3	Method pairs
CLDH [4]	Parse each code fragment to AST	AST	LSTM	T1–T4	code fragments
White et al. [5]	Use ANTLR to tokenize code	AST FBT olive trees	RtvNN	T1–T4	Method/file level
Marastoni et al. [6]	Leverage tigress C to obfuscate dataset	Binary image	CNN	T4	Image level

token in each method and the similarity between the method pairs, which is input to the classifier for training to detect clones of the given code base. To assess CCLearner's sensitivity to the parameter settings in DNN and the selected features, we experimented with different configurations of DNN and investigated different feature sets. It is observed that CCLearner worked best with 2 hidden layers and 300 training iterations in DNN. Since the token only reveals the code relationship of the grammatical level, the ability of this method to measure the similarity between code functions is limited.

CCLearner uses only lexical information about the source code and does not consider code structure information, which makes it difficult to detect functional clone pairs. Literature [4] proposes an end-to-end deep learning framework called CDLH, which captures lexical and syntactic information on the functional behaviour of code fragments in a supervised manner and hashes to further convert real-valued representations into binary hash codes. Compared with CCLearner, the learned features are more compact, which significantly improves detection efficiency and saves storage space. At the same time, writer studies the influence of different lengths of hash code on clone detection performance of CDLH measured by $F1$ value, and $F1$ values of CDLH with respect to different code lengths ranging from 8 to 48 are not sensitive to hash code length in this range.

Literature [5] comprehensively considers code structure and identifier, which are simultaneously used to model. Different from CCLearner's work based on token frequency statistics, this article uses a greedy way to transform between multiple tree structures to characterize the code, and the RtNN model is used to map similar terms in the code fragments to similar continuous value vectors. In addition, this model is based on the RvNN model to learn code fragments of different levels of granularity. In the experiment set, ANTLR is used to tokenize the source code and the RNNLM Toolkit [9] to train several RtNNs for each system, varying hidden layer sizes and depths. Then, using perplexity [10] as a proxy for evaluating the quality of model, the higher quality the model, the better the accuracy.

Literature [6] proposes a novel program similarity detection method, which converts binary code into image data that is input to CNN network of training model. At the same time, the author uses Tigress C obfuscator [11] iterative transformation grammar to expand the original dataset to fully train the model. Since the CNN network needs to view the entire image extracted from the binary file to complete the classification of each file, the size of the binary executable file limits the performance of the system.

In the experiment phase, all the weights in the CNN are initialized with random values taken from a normal distribution with standard deviation set at 0.1 to avoid running into local minima at early stages, and the following parameters are further fine tuned to improve the performance of the model: test_ratio, gradient_rate, limit, random_seed, and norm.

In order to prove the validity of the combination of various code characterization methods of similarity task detection, literature [12] implements four code representations: Identifiers, Abstract Syntax Trees, Control Flow Graphs, and Bytecode, and trains a model for each representation. The experimental results show that, by integrating different models, the accuracy of code clone detection can be further improved.

Compared with the method of text and token, the grammar-based codes representation method considers the structural information about the program code, which is more fault tolerant for the sequence conversion to program statements and minor modifications. However, the tree-based method still faces some shortcomings. With the enhancement of the logic structure of the program, the complexity of constructing the program abstract syntax tree increases, which further leads to an increase in the computational overhead of extracting key codes and vector transformation operations.

2.3. Semantic-Based Clone Detection Methods. The code representation of the semantic level not only considers the syntax of the source code but also uses the control flow and data flow information about the program code to solve the code cloning detection problem of the code function level. Data flow graphs, control flow graphs, and program dependency graphs generated by programs are currently the mainstream representation methods.

Table 4 shows the technical implementation and technical characteristics of deep learning in semantic cloning detection. Program code is characterized by a program dependency graph of control flow and data flow. Well-known detection technologies are DeepSim, which uses CFG and DFG node information; ZEEK uses code symbolization and hash techniques; CCDLC and Sheneamer et al. use PDG to consider program semantic information on a deeper level.

The program has a well-defined syntax, and the abstract syntax tree can be well represented. AST has shown excellent results from syntax-level cloning detection, but it cannot effectively detect discontinuous code clones. Literature [13] implements a novel clustering function, which effectively

Table 4: Technical characteristics of semantic-based clone detection.

System/writer	Preprocessing	Representation	Network	Clone types	Classification object
CCDLC [13]	Trimme, normalize method blocks	BDG PDG AST	CNN	T3-T4	Method blocks
Sheneamer et al. [14]	Trimme, nirmalize method blocks	AST PDG BDG	—	T1–T4	Method blocks
ZEEK [15]	Split the procedure to basic blocks	Hashes	NN	T4	Code block
DeepSim [16]	Use WALA to analysis bytecode	DFG CFG	DNN′	T4	Method level

combines the high-level features extracted from PDG with the low-level features extracted from BDG, closely reflecting the relationship between the data in the program code. As for model training, the CNN network is implemented with dropout regularization and Rectified Linear Units, which is run with multithreaded mini-batch descent, and the experimental result shows that CCDLC achieves good results in code obfuscation and semantic cloning detection. Compared with [14, 17], the CCDLC system further improves the data preprocessing and feature vector conversion process.

In order to better perform the embedding learning of word vectors, literature [15] introduces a new vector representation method proc2vec. First, split the program assembly code into basic blocks and use the strands that compose a code section as feature set, transforming strands to numbers and assembling those numbers to form a vector that represents the corresponding code. Compared with direct token symbol numerical conversion, the numerical vector matrix is more sparse, which can greatly reduce the computational cost of similarity. The neural network model is a common four-layer structure, which is trained using crossentropy cost function, dropout regularization of 0.1, batch size of 32, and 3 passes over the data.

Literature [16] analyses the relationship between various variables and basic code blocks based on control flow and data flow and considers three kinds of features for encoding the control flow and data flow information (variable features, basic block features, and relationship features between variables and basic blocks). Then, a high-dimensional sparse binary feature vector semantic matrix is generated through the predefined code rules. Semantic feature matrix reduces the problem of finding isomorphic subgraphs to detecting similar patterns and makes it easy to use for the later processes. The feature parameters of model training are considered from layer size, epoch, learning rate, dropout, and L2 regularization, whose appropriate values are set from referencing the parameter settings in the classic model. Compared with the existing state-of-the-art techniques, DeepSim significantly outperforms better in recall, precision, and time performance.

Compared with the general natural language text characteristics, the semantic-based method fully considers the program structure, sequence, and special grammar information and deepens the semantic features. However, semantic-based representation technology requires the use of generators to generate program dependency graphs for each programming language, which has poor scalability, and seldom talks about the selection and optimization of neural network models.

In the existing code similarity detection methods, a four-layer network structure model is commonly used. RNN and CNN have their own advantages in data relationship processing and feature extraction, whose advantages can be considered to design a new network structure. For the selection of feature parameters, we can learn from the training method of k-fold crossvalidation.

3. Code Pattern-Based Vulnerability Detection

The code pattern-based vulnerability detection technology mainly involves two stages. In the training phase, control flow and data flow technology are used to extract key codes in the program, which are transformed into vector, with the help of current mainstream tool (such as word2vec), which is input them into appropriate neural networks for supervised training. In the detection phase, the same data processing is performed on the new software program, which is used to detect the existing vulnerabilities through the learned model. According to whether the program needs to be run, the code pattern-based vulnerability detection methods are divided into static detection methods and dynamic detection methods, whose network structures currently used for model training include CNN, RNN, and LSTM [18–20].

3.1. Static Detection Methods. Static analysis refers to the process of program analysis by constructing abstract syntax trees and program dependency graphs, without running the software, whose analysed object usually refer to source code or executable code. Compared with executable code, source code analysis can obtain more semantic information, comprehensively considers the information on the execution path, therefore finding more vulnerabilities and improving the hit rate.

Figure 2 shows the principle of code static analysis and neural network training. The whole process includes the following steps: sample code construction, feature extraction, word vector generation, and neural network model training and classification. Among them, vulnerability feature extraction mainly involves how to select appropriate granularity to represent software programs and vulnerability detection. Since deep learning or neural networks take vectors as input, we need to represent programs as vectors that are semantically meaningful for vulnerability detection. We should use some intermediate representation as a “bridge” between a program and its vector representation, which is the actual input to deep learning. Vulnerability feature extraction is to transform programs into some intermediate representation that can preserve (some of) the

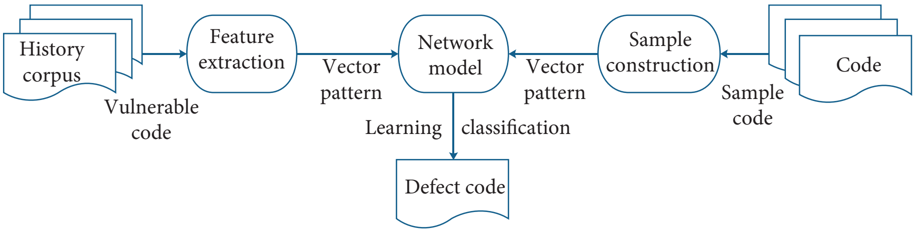

Figure 2: Principle of code static analysis and neural network training.

semantic relationships between the programs' elements (e.g., data dependency and control dependency) through CFG and PDG technology. Word vector generation is based on feature extraction, applying the most mainstream word vector generation technology so that intermediate representation can be transformed into a vector representation, that is, the actual input to neural networks. Neural network training classification involves two stages of training and detection. The training phase takes the code vector representation extracted from the historical code base as input, whose output is neural network of fine-tuned model parameters. In the detection phase, the code vector representation extracted from the new software program is taken as input, and the output is the classification result.

Table 5 shows the technical implementation and technical characteristics of deep learning in static code vulnerability detection. VulDeePecker for the first time demonstrated the potential of deep learning technology in vulnerability detection; SySeVR and CPGVA started from PDG and made better use of program control flow graphs; Lee et al. started from the binary level of the program to study the issue of assembly instruction-level code detection.

Traditional vulnerability detection methods require human experts to manually define the characteristics of the vulnerability, which is heavy and tedious. In addition, each person's perception of the characteristics of the vulnerability and the level of experience are different, making it difficult to achieve the desired results.

In order to tap the potential of deep learning in vulnerability detection, literature [21] proposed VulDeePecker, a vulnerability detection system based on deep learning, which proposes the concept of code gadgets that is some intermediate representation that can preserve (some of) the semantic relationships between the programs' elements (e.g., data dependency and control dependency). Then, the intermediate representation can be transformed into a vector representation that is the actual input to neural networks. Considering the relationship between variables and statements in the program code, the author uses the BLSTM network structure. This is because the arguments of a program function call may be affected by earlier statements of the program and may be also affected by the later statements. During model training, article adopts a 10-fold crossvalidation to train a BLSTM neural network and varies the number of hidden layers for each BLSTM neural network to observe the influence on the resulting $F1$-measure.

VulDeePecker uses data flow analysis to generate code gadgets, failing to implement the control flow analysis process, and the types of vulnerabilities detected are limited. In addition, the feature parameters selected for model training are mainly selected based on the experience in the NLP task, and how to select more appropriate feature parameters requires further research.

Aiming at the existing shortcomings of VulDeePecker, the literature [22] further considers the program control flow and proposes a grammar-based and semantic and vector-based vulnerability detection system framework. Similar to VulDeePecker extracting code gadgets, data flow analysis is used to extract and generate SyVCs from the program that then are used to generate program slices through the PDG and converted into SeVCs. The main parameters for learning BLSTM are dropout is 0.2, batch size is 16, number of epochs is 20, output dimension is 256, default learning rate of 0.002, dimension of hidden vectors is 500, and number of hidden layers is 2. Compared with VulDeePecker, SySeVR proposed its own word vector generation algorithm that avoids the problems of the word2vec model and optimized the selection of feature parameters of the neural network model.

As mentioned earlier, it is difficult to fully discover multiple vulnerabilities by considering program structure, data flow, and control flow dependencies. Literature [23] proposes a novel source code representation method—code attribute graph—which is a novel representation of source code and merged concepts of classic program analysis, namely, abstract syntax trees, control/data flow graphs, and program dependence graphs into a joint data structure. In the process of program preprocessing, CPGVA uses Stream extraction to consider the sequence of program points that may be taken by various flow graphs of the program, avoiding program dependencies that cannot be considered in traditional PDG. The article does not discuss in detail the impact of network structure feature parameters on the performance of model detection, which mainly compared the state–of-art source code review methods related to deep learning the results of SARD [25] dataset, depending on CNN (AlexNet, Lenet, Tcnn, etc.) or RNN (LSTM, BLSTM, GRU, etc.).

In order to better evaluate the performance of deep feature learning vulnerability detection technology, the literature [26] compares three kinds of vulnerability detection methods. The experimental results are not good in cross-project and class imbalance problems of software vulnerability detection, but good research ideas for future research.

In addition, literature [24] proposes the Instruction2vec model to vectorize assembly code in the research process of

Table 5: Technical characteristics of deep learning in static code vulnerability detection.

System/writer	Analysis object	Vulnerability types	Vector techniques	Network	Dataset
VulDeePeckke r [21]	Code gadgets	Library/API function	word2vec	BLSTM	NVD SARD
SySeVR [22]	Code gadgets	126 types of vulnerability	Custom algorithm	6 deep neural networks	NVD SARD
CPGVA [23]	Code stream	CWE-78 CWE-90	word2vec	CNN, RNN (variant net)	SARD
Lee et al. [24]	Function	CWE-121	instruction2vec	Text-CNN	Juliet

binary program vulnerability detection. Compared to the method of generating word vectors based on the word2vec model, the process of Instruction2vec utilizes the word2vec results for all words in the assembly code. At the same time, most instructions use one opcode and two operands, which also have a fixed length. NLP uses a high-dimensional vector to represent a large number of word vectors, but the assembly code does not need a high-dimensional vector because the number of words is small, instruction2vec better retains the potential relationships of opcodes, operands, and registers. This article explores the impact of different convolution kernel sizes and the number of hidden layers on model performance; Text-CNN uses 9 types of filters, each 128 in count, to maximize performance. The experimental results can be seen that increasing the number of filters in the CNN network of a certain range can significantly improve the detection accuracy.

Deep learning-based static vulnerability detection methods analyse program code dependencies on the source code level and preprocess the vulnerable code based on the idea of program slicing. In the existing research, the labelling of each sample code needs to be carried out manually, and the model is only for specific languages and types of vulnerabilities and is not suitable for extension to large projects. At the same time, the influence of the selection of different feature parameters on the neural network model is not explained too much in the existing methods of the article.

3.2. Dynamic Detection Methods. Dynamic analysis is the process of verifying or discovering the vulnerability of the software by running a specific program and obtaining information such as the output or internal state-of-the program, whose object of analysis is executable code. Compared with the static methods, the dynamic methods analyse vulnerabilities to obtain specific operating information, so the analysed vulnerabilities are generally more accurate and have a lower false alarm rate.

Static analysis tools have proven to be particularly effective in specific application areas, such as embedded systems or aerospace applications. However, it is much more difficult to use these techniques on more common software. Literature [27] proposes a deep learning-based dynamic analysis software vulnerability detection method. The final execution state was determined through the three steps of binary executed, events hooked, and events collected and the VDiscover tool [28]. Then, use the zzuf tool to implement data labelling. Finally, the sequence of 9872 function calls is analysed as a feature to represent the mode of the binary program during execution. Limited by the amount of data and the length of the sequence in data preprocessing, some specific types of vulnerabilities may be missed in actual detection.

Literature [29] starts with the generation and selection of intelligent seeds to reduce the exploration of useless paths. The solution proposed in this article is NeuFuzz, which uses deep neural networks to learn hidden vulnerability patterns from a large number of vulnerable and clean program execution paths. During online guided fuzzing, use the prediction model to determine whether an unseen path is vulnerable, and the seed is marked according to the prediction result and then added to the seed queue. At last, vulnerable seeds will be prioritized and assigned more mutation energy in the next seed selection process and seed mutation process.

Deep learning technology can reduce human feature engineering, which is expected to replace traditional vulnerability detection methods in the future research process and significantly improve vulnerability detection performance. However, deep learning in vulnerability detection methods face many shortcomings, where the model is often trained based on the program source code, which is not available in most cases. In addition, detection granularity and limited models further limit the detection effect.

In order to further advance the development of automated vulnerability detection technology, in the future research process, abstract modelling of programming languages with similar grammar and syntax can be considered to achieve software vulnerability detection between multiple languages with the same model. At the same time, transfer learning can be considered to implement the learning process of small pieces of data to solve the lack of large-scale vulnerability datasets. In addition, the optimization of neural network structure and the selection of feature parameters have always been problems faced by relevant practitioners [30]. Existing code vulnerability detection technologies are often based on 10-fold cross-validation training methods to obtain optimal parameters, the way of which is often compromised of training time and model performance for large-scale project data.

4. Automatic Software Program Repair

Early automated patching technology was mainly used to prevent the spread of worms, and automated patching technology slowly penetrated into all aspects of computer software security with the development of technology. The automatic program repair technology can assist in the automatic repair of some defects in the software program, thereby effectively reducing the program debugging time of the software developer.

4.1. Patching Process. The automated patching technology is divided into the following three phases: the software fault location phase, the patch generation phase, and the patch evaluation phase.

Software fault location is a prerequisite for automatic program repair and is mainly used to identify the location of potential defects or vulnerabilities in the program. At present, the commonly used fault location technologies are divided into two categories: static fault location technology and dynamic fault location technology. The static fault location technology [31–33] mainly obtains the control dependence and data dependency relationship between the tested program codes through program analysis technology, thereby confirming and locating the fault location. The dynamic fault location technology [34] obtains the execution information of the program by executing the preselected test case and locates the position of the defect statement in the tested program by analysing the execution flow of the program.

The build patch phase defines some operational rules by examining the program code structure and analysing the sample code submission and modification information, and then modifying the defect statement through the defined operations. Two patch generation methods are commonly used based on search and semantics. The search-based patch generation method is a process of finding a patch in a search space and verifying it, often improving the patch generation efficiency by scaling the search space size and optimizing the search strategy. The semantic-based patch generation method generates patches by integrating techniques such as fault location, constraint solving, and program synthesis.

The patch evaluation phase evaluates the generated candidate patches until a patch is found that enables all test cases to be executed by the program.

5. Grammar-Based Patching Technology

The grammatical error-based program repair technique modifies the error code by learning the code grammar features to achieve certain language specifications.

5.1. Token-Level Learning and Repair Technology. Standard LR parsers often have difficulty in resolving grammatical errors and their precise location and may even draw a misleading conclusion about where the actual error is. The source code of a computer program is a plain text representation that is similar to natural language of grammatical structure, where the development of natural language technology and deep learning technology makes it possible to learning-based repair.

Neural networks that learn similar grammatical structure information can enhance the effect of program repair, and the literature [35] proposes a technology that provides feedback on grammatical errors, which uses recurrent neural networks (RNN) to simulate grammatically valid token sequences. For a given program, a set of grammatically correct code submissions are first modelled based on the RNN network, which is queried using the prefix token sequence to predict the effective token sequence pair for code submissions with grammatical errors.

The experiment with both the RNN and LSTM networks with 1 or 2 hidden layers and each with either 128 or 256 hidden units and several other indicators (such as learning rate, batch size, and gradient threshold) are fine tuned during training. The article focuses on analysing that adding additional hidden layers with more hidden units will actually reduce the performance of the network on our dataset. The current method only uses the prefix token sequence to repair and does not consider the relationship between the context of the token sequence, which cannot guarantee the correctness of the repair sequence semantically and repair multiple errors at the same time.

Literature [36] trains two opposite language models based on the LSTM network, which comprehensively considers the token context content to solve a single token syntax error. The neural network is trained from varying the number of neurons in the hidden layer, while keeping the number neurons per layer constant while adjusting the other feature parameters. On the premise of a token vocabulary formed by training with a large number of examples, the model can guarantee the accuracy of both positioning errors and correcting error bits, however, which in turn limits the time efficiency of the model. The target audience of this tool is experienced programmers, and it is difficult for novice developers to use this tool. Therefore, the data corpus, time dimension, and scalability issues limit the performance of GrammarGuru.

The GrammarGuru scheme is limited to the error code and the training data being in the same domain, of which literature [37] overcomes problem and uses n-gram model and LSTM model modelling to correct grammatical errors. For hyperparameters' impact, the article tests 985 different configurations, varying the hidden layer size (50, 100, 200, 300, 400, and 1000), the context size (5, 10, 15, and 20), and optimizer (RMSprop and Adam), and Adam optimizer outperforms the impact of all other parameters. Evaluating the Blackbox corpus, the experimental results show that the language model can successfully locate and repair grammatical errors in manually written codes without parsing.

5.2. Text-Based Learning and Repair Technology. Due to the dependencies between the various parts of the program, even a single error may require analysis of the entire program. Repair accuracy is an important indicator in program repair techniques, and the previous literature [35–37] locates and repairs program errors based on token-level accuracy, whose generated error patches cannot completely repair programming errors.

In order to improve the effect of error repair, the literature [38] proposes an end-to-end solution called DeepFix comprehensively considering the token context text, which uses a multilayer sequence neural network of attention to capture program text dependencies. The article uses the attention-based sequence-to-sequence architecture implemented in Tensorflow and does a 5-fold crossvalidation to give an accurate evaluation of our technique and select best feature parameters. The effectiveness of iterative repair shows that if the network fails to produce a fix for a program in an iteration, and the subsequent iterations are not executed for that program as applying the same network again will not change the outcome.

DeepFix calls the model output in an iterative manner to fix multiple errors in the program one by one, but most of the repairs only involve reducing compilation errors and cannot fundamentally solve the problem. Literature [39] proposes the DeepRepair learning framework, which uses redundant or repeated codes in the program and based on the principle of code similarity to prioritize and convert the statements in the code library to generate program repair components. The article selects word2vec to train word embedding, which is used to initialize the embeddings for the recursive autoencoders. Deep Repair's search strategy using the embedding-based ingredient transformation algorithm, which yields patches in fewer time and finds higher quality patches than jGenProg [40].

Literature [41] proposes a system TRACER for fixing errors, which improved the repair accuracy by error localization, abstract code repair, and concrete three-module code repair. The writer conducts an extensive grid search to set hyperparameters, whose best configurations are chosen upon crossvalidation based on the performance measure of Precision at the Top and Smoothed Precision at the Top. At the same time, the modular approach adopted by TRACER focuses on a small set of sentences, which greatly improves the ability of the RNN framework to propose relevant local corrections.

In the existing online programming courses, the program repair generated by TRACER has more teaching value than the built-in error correction function of the compiler. However, TRACER uses a compiler-based heuristic method, which has the problem of inaccurate error location. At the same time, the input data of TRACER fails to consider the types of errors that require global context information, which makes the model difficult to deal with type errors such as opening/closing missing parentheses. In addition, TRACER and DeepFix use supervised learning techniques to train error correction models.

Literature [42] proposes a reinforcement learning framework RLAssist, which overcomes the compiler-based TRACER heuristic. The model does not require any supervision during the training process, whose agent learns from the original program to generate candidate repairs. The model is implemented following an open source implementation 2 of A3C, a suitable configuration of feature parameters for our task through experimentation. The experiment result shows that the greater the number of episodes is, the more the error messages resolved. At the same time, the other parameters have different degrees of influence on the proportion of error repair degree.

The grammar-based program repair technology still faces some problems that some methods still borrow the compiler to locate the error location and verify the error repair. Compilers usually cannot provide accurate error location information, which makes it difficult for novice programmers to perform accurate repairs based on the information provided by the compiler. In addition, it is difficult to ensure the semantic validity of program statements through the compiler to repair errors, which in turn changes the behaviour and results of the program.

The RNN network has greater advantages in processing sequence data relationships and generating text data, and its variant network structure models are often selected to achieve program repair due to the program repair process which involves the process of data generation. In terms of model performance improvement, it is often achieved by optimizing the following feature parameters: learning_rate, batch_size, hidden_layer, token_length, embedding_size gradient_optimizer, hidden_unit, and iteration. In future repair techniques, researchers should consider the above information in the program text and develop better models to locate and fix grammatical errors.

6. Semantic-Based Patching Technology

A semantic error-based program repair technique generally refers to modifying the program code so that the actual program behaviour is substantially consistent with the behaviour expected by the programmer. In order to repair the errors in the program from the semantic level, the literature [31] proposes a novel neural network architecture, which separates the nonstatistical process of generating and applying repair suggestions from the statistical process of scoring and sorting repair. The rule-based processor is first used to generate the repair candidate, and then the statistical model is scored using the novel neural network architecture so that different repair candidates can compete in the probability space to fix the error with higher precision. In the experiment stage, a small amount of hyperparameter, such as training epochs, hidden dimensions, embedding size, and dropout, are tuned to train the model, which drastically outperforms the attentional sequence-to-sequence model.

Literature [31] predicts the repair of all program locations by performing an enumeration search operation and then calculates the score of the fix to select the best fix, which causes a training/test data mismatch problem during the repair process. On this basis, the literature [32] uses the multihead pointer to jointly perform classification, positioning, and repair, which can achieve fine-grained positioning of specific variables and complete the repair process. The article defines two parameters to filter the predictions, where the influence of feature parameters on neural network performance is not discussed. This result show that the network is able to perform the localization and repair tasks jointly, efficiently, and effectively, without the need of an explicit enumeration.

In order to overcome the semantic incomprehensibility of patch collection generated by semantic patch technology, literature [33] mines semantic-related repair patterns based on iterative triple clustering strategy, using abstract syntax tree, editing operation tree, and code context tree for pattern representation, making full use of the similarities between change information and token changes in the AST diff tree. The evaluation of thousands of software patches collected in open source projects shows that the pattern generated by FixMiner is associated with the semantics of the errors addressed by the relevant patches, sufficient to generate patches with high probability of correctness, and can be correctly repaired using the mining model.

The grammatical- and semantic-based program representation differences are expressed as follows.

First, during the running of the program, the statements in the program are generally not learned in the order in which the corresponding markup sequence is presented to the deep learning model. Second, control dependencies and data dependencies in programs play an important role in program semantics, but these dependencies are not well represented in tokens and ASTs. Third, from the perspective of program performance, even a similar grammatical structure may lead to large differences in program semantics.

The literature studies [31–33] analyse the source code of the program and can only capture the dependencies between the sequential data and cannot handle the relationship such as jump or recursion. Program states of contiguous tuples of real-time variable values can accurately capture such program semantics while also providing natural results for the neural network model. Literature [34] proposes a novel semantic program embedding system, which combines variable tracking embedding, state tracking embedding, and hybrid embedding to learn semantic information from the program execution trajectory. All encoders in each of the trace model have two stacked GRU layers with 200 hidden units in each layer except that the state encoder in the state trace model has one single layer of 100 hidden units. Comparing dynamic program embeddings with syntax-based program embedding in predicting common error patterns made by students, the embeddings trained on execution traces significantly outperform those trained on program syntax.

At present, the semantic-based program repair technology mainly learns the semantic features of the program by constructing the program AST and can only analyse the dependencies between the sequential data. Like the grammar-based program repair technology, the existing technologies are also implemented based on RNN and its variant networks due to the program repair process involving the process of data generation. However, the current technology rarely discusses the influence of characteristic parameters on neural network model training and mainly focuses on algorithm optimization, which limits the performance of current technology to a certain extent.

In the future program repair technology, we can learn from the vulnerability detection technology that combines with AST and PDG technology to enhance the performance of the existing program semantic repair technology and analysis of the impact of feature parameters on the model performance to further promote the applications of deep learning technology in the program repair technology.

7. Automatic Software Defect Predicting

Software defects refer to any deficiencies in the product description, design, and coding stages. Software defect prediction can help software developers find errors quickly, allocate limited resources reasonably, and prioritize their testing work. In recent years, software developers have used various deep learning algorithms to analyse software defects and increase software testing to improve software quality, reduce software costs, and enhance software maintainability.

This section describes the latest research results of deep learning technology in software defect prediction in recent years according to the difference in data sources and prediction granularity, whose content mainly covers within-project defect prediction, crossproject defect prediction, and just-in-time defect prediction.

8. Within-Project Defect Prediction

Within-project defect prediction can be further divided into intraversion defect prediction and crossversion defect prediction, and the former uses a specific version data training model of the software system for defect prediction, and the latter uses different version data training models of the software system for defect prediction.

Table 6 shows the technical implementation and technical characteristics of deep learning in the project defect prediction. Wang et al. use CLNI to complete the data labelling task, and the Tree-LSTM model is trained in an unsupervised manner; CAP-CNN comprehensively considers code review information to implement code defect prediction, and other methods start with network models to improve prediction performance.

In order to improve the quality of the software, developers put a lot of effort into the testing and debugging process. However, in most cases, developers have limited resources and time constraints. In this case, automated software defect prediction techniques can better help them find errors and prioritize testing.

Literature [43] adopts the edit distance similarity computation algorithm [49] and CLNI [50] to eliminate data with potential incorrect labels to eliminate the influence of data noisy and uses the AST analysis program source code to obtain syntactic information, which is converted into feature vector that is input to DBN network for building effect prediction models.

In the training phase, three parameters are fine tuned to train an effective DBN for learning semantic features. The deeper the network structure, the more nodes, which will increase the model convergence time, and the performance improvement will be limited. Furthermore, the proposed semantic feature generation approach is only evaluated on

Table 6: Technical characteristics of within-defect prediction methods.

System/writer	Datasets	Metrics	Feature generation	Data labeled
Wang et al. [43]	PROMISE	P $F1$ Recall	Parse source code, handle noise, and map tokens, generate feature via DBN	CLNI
Dam et al. [44]	Samsung	P $F1$ Recall	Parse source code, map AST nodes, generate feature via Tree-LSTM	Model generation
DP-CNN [45]	PROMISE	P $F1$ Recall	Parse source code, extract and encode token, generate feature via CNN	Repository provided
SDNN [46]	NASA	$F1$ AUC	Delete repeated entities, replace missing value, data normalization	Repository provided
CAP-CNN [47]	PROMISE	$F1$	Split source modules, encoded as vector via pretrained word2vec, generate feature via CNN	Repository provided
DefectLearner [48]	12 open source projects	P $F1$ Recall	Remove comment, use word embedding method, generate feature via LSTM	Projects provided

open source java projects, whose performance on closed source software and projects written in other languages is unknown. Furthermore, the proposed semantic feature generation approach is only evaluated on open source java projects, whose performance on closed source software and projects written in other languages is unknown.

The state-of-the-art method [43] leverages Deep Belief Network in learning semantic features of token vectors extracted from programs' ASTs, which outperforms traditional feature-based approaches in defect prediction. However, it overlooks the structural information about programs which can lead to more accurate defect prediction. Literature [44] uses a novel attention mechanism to implement a tree-based LSTM network, which uses the strong predictive ability of the tree network structure to avoid the influence of noisy data and is trained in an unsupervised manner. In model training, dropout is considered to prevent overfitting in neural networks, which is further combined with perplexity evaluation metric for choosing the best model. The effectiveness of this method has been demonstrated in Samsung open source projects.

As reported by deep learning researchers in speech recognition [51] and image classification [52], Convolutional Neural Network (CNN) is more advanced than DBN since the former can capture local patterns more effectively. Literature [45] borrows the method of [43], selects three same types of nodes on ASTs as tokens, and wraps word embedding as a part of CNN architecture. The feature parameter fine tuning is a key to train a successful CNN, which directly affects the convergence of the model. The article mainly varies the values of these three parameters: the number of filters, the filter length, and the number of nodes in hidden layers, whose best values are obtained from validation experiment. During the evaluation of seven open source projects, experimental results show that, on average, DP-CNN improves existing technology methods by 12%.

Literature [46] explores the advantages of Siamese networks to propose a novel SDP model, which integrates similarity feature learning and distance metric learning into a unified approach. Compared to previous methods [43, 49, 50], SDNN uses the two identical fully connected networks to learn the highest-level similarity features and the metering function as the distance measure for between the highest-level features. In experiment process, the author explores the impact on the number of hidden layers on the performance of the model and obtains the best results from 3 hidden layers, which is the best model through experiments. In addition, cell units, mini-batch size, learning rate, and dropout are considered to further improve the accuracy of model.

Previous techniques have been often based on the program code itself, without considering information such as program code comments. Literature [47] proposes a novel defect prediction model named CAP-CNN, which is a deep learning model that automatically embeds code comments into generating semantic features of the source code for software defect prediction. CAP-CNN combines source code and comments for software defect prediction, which generates a more semantic feature representation indicating the structure and functionality of source modules. In this experiment, the author refers to the conventional CNN network structure model and parameter settings, which do not further explore the influence of various feature parameters on the model performance. Experimental results of widely used software defect prediction datasets show that code comments help improve defect prediction performance.

From the perspective of code naturalness, the literature [48] introduces crossentropy as a new software metric into the typical defect prediction of the file level, which is based on the RNN structure mining code token set of the LSTM unit to capture common mode in the program. At the same time, the paper adopts several optimization strategies of dropout, gradient clipping, and adaptive learning rate to deal with overfitting, gradient disappearance, and gradient explosion problems. In a series of comparative experiments, the fine tuning of various performance evaluation indexes further confirmed the important role of various characteristic parameters in model performance. The experimental results show that crossentropy metric is more discriminative than the traditional metrics.

Within-project defect prediction technology focuses on file-level defects, which constructs AST parsing program source code, generates word vectors using word embedding technology, and uses common network models for training. For each feature parameter of deep learning technology on within-project defect, dropout is a key parameter that can be used to prevent model overfitting, and several other parameters (such as learning rate and the number of hidden layers) are commonly used to accelerate the convergence of the model.

In addition, within-project defect prediction method mainly focuses on whether the new file contains defect information, which cannot effectively feedback the reason for the introduction of the defect or locate the position of the defect. In actual software engineering practice, it is difficult to ensure the repeatability and effectiveness of the model trained in open source projects.

9. Crossproject Defect Prediction

Affected by factors such as development processes and programming languages between different projects, previous research focuses on defect prediction within the project. In practice, a new project usually does not have enough defect data to build a predictive model, so crossproject defect prediction is necessary, which uses data from other projects to train the model to apply to the new project.

With the rapid development of artificial intelligence technology, especially the powerful learning and presentation capabilities of deep learning technology, which provides a good application basis for the research of defect prediction between different projects. Traditional research focuses on manually designing coding program features and simply exploring the performance of different machine learning algorithms, unable to capture the semantic differences of the program, and the built predictive model has low accuracy.

In order to effectively compensate for the gap between program semantics and defect prediction features, the literature [43] uses DBN to automatically learn the characteristics of token vectors extracted from the program's AST and then use these features to train the defect prediction model. In the experimental stage, the article explores how three parameters of the number of hidden layers, the number of nodes in each hidden layer, and the number of training iterations affect the model precision, $F1$, and recall. The experimental results show that the semantic features significantly improve the CPDP mode compared to traditional features, and appropriate feature parameters can better improve the performance of the model.

In order to bridge the gap between program semantics and defect characteristics, literature [44] develops a novel predictive model that captures the contextual information that is far apart in the program by constructing a tree-shaped LSTM neural network to match the abstract syntax tree representation using the program source code. In the training phase, the article focuses on the impact of dropout on the performance of the model. Literature [53, 54] proposes to use deep neural networks in assembly instruction sequences rather than AST, which use C++ compilers to compile C++ programs into assembly code and then apply convolutional neural networks to learn the datasets of assembly instruction sequences, whose prediction effect is significantly better than the AST-based method.

In the previous forecasting model, only the program code data was focused on, and the program code comment information was rarely paid attention to. Literature [47] proposes a new defect prediction model CAP-CNN, which can automatically embed code annotations to generate semantic features of the source code of software defect prediction. At the same time, the model uses network coding and absorption of comment information to automatically generate semantic features during the training process, effectively overcoming the problem of missing comments in the program. Experimental results from several widely used software datasets show that comment features can improve defect prediction performance.

Existing software prediction models are mostly limited to source code, but the files obtained in the prediction work are usually binary executables. Due to software copyright and source code protection restrictions, it is difficult for the defect prediction research community and third-party security companies to obtain source code. Literature [55] proposed the smali2vec method to capture the features of smali in apks and use deep neural networks for training. The model focus on fine tuning three parameters including the number of hidden layers, the number of neurons in each hidden layer, and the number of iterations, which train an effective DNN for predicting defects in apks. The three parameters are fine tuned by conducting experiments with different values of these parameters on our training data and find the best configuration of the parameters from the AUC value and the error rate. Compared with the within-project defect prediction technology research, the research on crossproject defect prediction based on deep learning technology is still too little, and the research process of crossproject should be further promoted in future research.

10. Just-In-Time Defect Prediction

In actual scenarios, it is difficult for us to apply the error-proneness recommendations given by the software system to overcome these problems, of which developers introduce just-in-time defect prediction technology to solve. In just-in-time defect prediction, modules that are prone to failure can be identified at an early stage and fed back to developers for changes and repairs. At present, just-in-time defect prediction technology is carried out at the code change level, whose fine-grained analysis enables developers to more efficiently solve the problems encountered in the software development process.

Literature [56] first combines deep learning methods to improve the performance of instant defect prediction, which uses the Deep Belief Network that is consisting of three restricted Boltzmann machines and a logistic regression classifier to build a deeper model for detecting more expressive features. In the article, the neural network structure of model is a general configuration, whose numbers of hidden units are based on a range of numbers using a strategy similar to greedy search. At the meanwhile, the performance of the model is evaluated through ten-fold crossvalidation, whose cost effectiveness and $F1$-score are far better than the previous method.

Different from the literature [56], the literature [57] uses three fully connected backpropagation (BP) neural networks to construct a regression model rather than a classifier model. In addition, the Model Neural Network Regression (NNR) method utilizes ten numerical metrics of code changes and then feeds them to a neural network whose output indicates how likely the code change under test contains bugs. The article does not give too much explanation on the selection of model feature parameter values, which mainly introduces how to select the best model based on the crossvalidation training method. Given the inspection resources, the number of defects can be identified more efficiently based on the effort-aware instant defect prediction.

Literature [58] proposes a novel method, TLEL, which uses decision trees and integrated learning to improve the performance of immediate defect prediction. In the inner layer, the decision tree and the bag are combined to construct a random forest model. In the outer layer, random undersampling is used to train many different random forest models and they are assembled again using the stack. The article presents the effect of varying the values of the two parameters, NTree and NLearner, on the performance of TLEL on six datasets and uses ten-fold crossvalidation to evaluate the performance of TLEL based on two evaluation metrics of cost effectiveness and $F1$-score.

Just-in-time defect prediction technology can identify modules that are prone to failure at an early stage and feed them back to developers for changes and repairs, whose fine-grained analysis enables developers to more effectively solve problems encountered in the software development process. The problem of defect prediction is to determine whether the current software program contains defective code, which can be regarded as a two-class problem to a certain extent. In the field of nlp and images, various CNN network models perform well in classification tasks, which should be promoted in the application of the just-in-time defect prediction in future research.

11. Future Directions and Challenges

It is essential to ensure the reliability of the software, during the entire life cycle of software development to deployment, where application of deep learning technology accelerates the software development cycle and saves manpower and time costs. However, it is difficult for the existing technology to analyse and process the increasing number of security issues in a unified manner with the continuous development of software technology, which further makes the application of deep learning technology in the field of software security face some challenges. As shown in Table 7, there are still some challenges in applying deep learning techniques to the field of software security.

11.1. Feature Generation. The selected features play an important role in the neural network training process, whose quality is higher, the better the model training effect. In vulnerability detection, a method based on tokens, trees, and graphs is usually used to abstract the program source code, and then feature mapping and vector transformation techniques are used to generate word vectors, which are used as the input of the deep learning model. The current mainstream methods analyse the source code, and the program behavior cannot be effectively tracked in feature selection, which limits the performance of the model to a certain extent. Analyzing the binary code can better understand the behavior of the program, which makes the constructed model better locate the location of the vulnerability. Therefore, it is necessary to explore binary-oriented deep learning vulnerability detection technology.

Table 7: Opportunities and challenges for deep learning applied to software security research.

Challenges	Opportunity
Tool review	Deep learning model automatic feedback
Feature extraction	Tree and graph model combined
Semantic feature learning	Fine-grained program feature representation
High false negatives and false positives	DL combined with static, dynamic program analysis technology
Dataset	Establish an open source unified dataset standard library
Crossproject vulnerability detection	Transfer learning
Code metric	New code attribute

11.2. Model Selection. Various deep learning models were initially mainly used in the field of computer vision and image processing, which is still in its infancy in the application of software security research, and how to use the powerful learning capabilities of deep learning models to deal with software security issues needs to be resolved urgently. Various models of deep learning have different learning capabilities for the same data, and how to select a suitable model to learn feature data requires further exploration.

11.3. Datasets. Model training requires a lot of data, which is obtained from open source projects, in the current research. There are differences in data between open source projects and closed source projects, which makes model trained in open source projects may not be applicable in closed source projects. In addition, the data used for model training often suffers from the problem of unbalanced data types, which further limits the performance of many models.

11.4. Performance Evaluation. Traditional vulnerability detection requires human experts to manually define features, which is tedious and time consuming, and often faces the risk of high false positive rates and high false negative rates. Deep learning technology can handle natural language tasks well, and program code can be regarded as text data to a certain extent, and we can learn from the processing methods and evaluation indicators of text data to improve the performance of existing methods in natural language specific tasks.

11.5. Feature Parameters. The selection of feature parameter values plays an important role in various deep learning technologies, whose unreasonable parameter values' setting will greatly influence model performance. Many feature parameter values of the current techniques are selected based on experience, whose rationality has not been verified. For different tasks, setting the same value for the feature parameters corresponding to the same network model may have different effects.

12. Conclusions

In the process of software system development, it is the common goal of software developers to improve software quality and safety. Traditional detection methods require domain experts to spend a lot of time and energy to create feature engineering, and it is vital to combine deep learning technology with program analysis technology to assist software security research to further promote the development of automated detection technology. This article introduces in detail the latest research progress of deep learning technology in software vulnerability detection, software program repair, and software defect prediction, of which expounds and discusses the existing shortcomings. Looking at the development trend of automation technology in the software security field in recent years, deep learning technology will play an increasingly important role in the research of software security automation technology in the future. For the majority of researchers, deep learning technology is a historical opportunity that will promote an innovation in software engineering technology research.

Conflicts of Interest

The authors declare that they have no conflicts of interest.

Acknowledgments

This work was supported by the National Key R&D Program of China (Grant no.2018YFC1604000) and Natural Science Foundation of Hubei Province (Grant no.2017CFB663).

References

[1] CVE, http://cve.mitre.org/.

[2] S. Bellon, R. Koschke, G. Antoniol et al., "Comparison and evaluation of clone detection tools," *IEEE Transactions on Software Engineering*, vol. 33, no. 9, pp. 577–591, 2007.

[3] L. Li, H. Feng, W. Zhuang, N. Meng, and B. Ryder, "CClearner: a deep learning-based clone detection approach," in *Proceedings of the 2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pp. 249–260, IEEE, Shanghai, China, September 2017.

[4] H. Wei and M. Li, "Supervised deep features for software functional clone detection by exploiting lexical and syntactical information in source code," in *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence*, pp. 3034–3040, Melbourne, Australia, August 2017.

[5] M. White, M. Tufano, C. Vendome et al., "Deep learning code fragments for code clone detection," in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, ACM, pp. 87–98, Singapore, September 2016.

[6] N. Marastoni, R. Giacobazzi, and M. Dalla Preda, "A deep learning approach to program similarity," in *Proceedings of the 1st International Workshop on Machine Learning and Software Engineering in Symbiosis*, ACM, Montpellier, France, pp. 26–35, September 2018.

[7] ANTLR, http://www.antlr.org/.

[8] Use JDT ASTParser to Parse Single, Java Fles, http://www.programcreek.com/2011/11/use-jdt-astparser-to-parsejava-file/.

[9] T. Mikolov, S. Kombrink, A. Deoras, L. Burget, and J. Cernocky´, "RNNLM—recurrent neural network language modeling toolkit," in *Proceedings of the ASRU'11*, Waikoloa, HI, USA, December 2011.

[10] D. Jurafsky and J. Martin, *Speech and Language Processing*, Pearson, London, UK, 2nd edition, 2009.

[11] ChristianCollberg, TheTigressCdiversifier/Obfuscator, 2015.

[12] M. Tufano, C. Watson, G. Bavota et al., "Deep learning similarities from different representations of source code," in *Proceedings of the 2018 IEEE/ACM 15th International Conference on Mining Software Repositories (MSR)*, IEEE, Gothenburg, Sweden, pp. 542–553, May 2018.

[13] A. Sheneamer, "CCDLC detection framework-combining clustering with deep learning classification for semantic clones," in *Proceedings of the 2018 17th IEEE International Conference on Machine Learning and Applications (ICMLA)*, IEEE, pp. 701–706, Orlando, FL, USA, December 2018.

[14] A. Sheneamer, H. Hazazi, S. Roy, and J. Kalita, "Schemes for labeling semantic code clones using machine learning," in *Proceedings of the 2017 16th IEEE International Conference on Machine Learning and Applications (ICMLA)*, IEEE, pp. 981–985, Cancun, Mexico, December 2017.

[15] N. Shalev and N. Partush, "Binary similarity detection using machine learning," in *Proceedings of the 13th Workshop on Programming Languages and Analysis for Security*, ACM, Toronto, Canada, pp. 42–47, October 2018.

[16] G. Zhao and J. Huang, "Deepsim: deep learning code functional similarity," in *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ACM, Lake Buena Vista, FL, USA, pp. 141–151, November 2018.

[17] A. Sheneamer, S. Roy, and J. Kalita, "A detection framework for semantic code clones and obfuscated code," *Expert Systems with Applications*, vol. 97, pp. 405–420, 2018.

[18] N. Kalchbrenner, E. Grefenstette, and P. Blunsom, "A convolutional neural network for modelling sentences," 2014, https://arxiv.org/abs/1404.2188.

[19] T. Mikolov, M. Karafiát, L. Burget et al., "Recurrent neural network based language model," in *Proceedings of the Eleventh Annual Conference of the International Speech Communication Association*, Makuhari, Japan, September 2010.

[20] H. Sak, A. W. Senior, and F. Beaufays, "Long short-term memory recurrent neural network architectures for large scale acoustic modeling," 2014.

[21] Z. Li, D. Zou, S. Xu et al., "VulDeePecker: a deep learning-based system for vulnerability detection," 2018, https://arxiv.org/abs/1801.01681.

[22] Z. Li, D. Zou, S. Xu et al., "SySeVR: a framework for using deep learning to detect software vulnerabilities," 2018, https://arxiv.org/abs/1807.06756.

[23] W. Xiaomeng, Z. Tao, W. Runpu, X. Wei, and H. Changyu, "CPGVA: code property graph based vulnerability analysis by deep learning," in *Proceedings of the 2018 10th International*

Conference on Advanced Infocomm Technology (ICAIT), IEEE, pp. 184–188, Stockholm, Sweden, August 2018.

[24] Y. J. Lee, S. H. Choi, C. Kim et al., "Learning binary code with deep learning to detect software weakness," in *Proceedings of the KSII the 9th International Conference on Internet (ICONI) 2017 Symposium*, Pittsburgh, PA, USA, July 2017.

[25] https://samate.nist.gov/SRD/.

[26] X. Ban, S. Liu, C. Chen, and C. Chua, "A performance evaluation of deep-learnt features for software vulnerability detection," *Concurrency and Computation: Practice and Experience*, vol. 31, no. 19, p. e5103, 2019.

[27] F. Wu, J. Wang, J. Liu, and W. Wang, "Vulnerability detection with deep learning," in *Proceedings of the 2017 3rd IEEE International Conference on Computer and Communications (ICCC)*, IEEE, pp. 1298–1302, Chengdu, China, December 2017.

[28] G. Grieco, G. L. Grinblat, L. Uzal et al., "Toward large-scale vulnerability discovery using machine learning," in *Proceedings of the Sixth ACM Conference on Data and Application Security and Privacy*, pp. 85–96, New Orleans, LA, USA, March 2016.

[29] Y. Wang, Z. Wu, Q. Wei et al., "NeuFuzz: efficient fuzzing with deep neural network," *IEEE Access*, vol. 7, pp. 36340–36352, 2019.

[30] R. Zhang, W. Li, and M. Tong, "Review of deep learning," *Information and Control*, vol. 47, no. 4, pp. 385–397, 2018.

[31] J. Devlin, J. Uesato, R. Singh et al., "Semantic code repair using neuro-symbolic transformation networks," 2017, https://arxiv.org/abs/1710.11054.

[32] M. Vasic, A. Kanade, P. Maniatis et al., "Neural program repair by jointly learning to localize and repair," 2019, https://arxiv.org/abs/1904.01720.

[33] A. Koyuncu, K. Liu, T. F. Bissyandé et al., "Fixminer: mining relevant fix patterns for automated program repair," 2018, https://arxiv.org/abs/1810.01791.

[34] K. Wang, R. Singh, and Z. Su, "Dynamic neural program embedding for program repair," 2017, https://arxiv.org/abs/1711.07163.

[35] S. Bhatia and R. Singh, "Automated correction for syntax errors in programming assignments using recurrent neural networks," 2016, https://arxiv.org/abs/1603.06129.

[36] E. A. Santos, J. C. Campbell, A. Hindle, and J. N. Amaral, "Finding and correcting syntax errors using recurrent neural networks," *PeerJ*, vol. 5, Article ID e3123v1, 2017.

[37] E. A. Santos, J. C. Campbell, D. Patel, A. Hindle, and J. N. Amaral, "Syntax and sensibility: using language models to detect and correct syntax errors," in *Proceedings of the 2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, IEEE, pp. 311–322, Campobasso, Italy, March 2018.

[38] R. Gupta, S. Pal, A. Kanade et al., "Deepfix: fixing common c language errors by deep learning," in *Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence*, San Francisco, CA, USA, February 2017.

[39] M. White, M. Tufano, M. Martínez, M. Monperrus, and D. Poshyvanyk, "Sorting and transforming program repair ingredients via deep learning code similarities," in *Proceedings of the 2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, IEEE, pp. 479–490, Hangzhou, China, February 2019.

[40] M. Martinez, T. Durieux, R. Sommerard, J. Xuan, and M. Monperrus, "Automatic repair of real bugs in java: a large-scale experiment on the defects4j dataset," *Empirical Software Engineering*, vol. 22, no. 4, pp. 1936–1964, 2017.

[41] U. Z. Ahmed, P. Kumar, A. Karkare et al., "Compilation error repair: for the student programs, from the student programs," in *Proceedings of the 2018 IEEE/ACM 40th International Conference on Software Engineering: Software Engineering Education and Training (ICSE-SEET)*, IEEE, Gothenburg, Sweden, pp. 78–87, May 2018.

[42] R. Gupta, A. Kanade, and S. Shevade, "Deep reinforcement learning for programming language correction," 2018, https://arxiv.org/abs/1801.10467.

[43] S. Wang, T. Liu, and L. Tan, "Automatically learning semantic features for defect prediction," in *Proceedings of the 2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*, IEEE, Austin, TX, USA, pp. 297–308, May 2016.

[44] H. K. Dam, T. Pham, S. W. Ng et al., "A deep tree-based model for software defect prediction," 2018, https://arxiv.org/abs/1802.00921.

[45] J. Li, P. He, J. Zhu et al., "Software defect prediction via convolutional neural network," in *Proceedings of the 2017 IEEE International Conference on Software Quality, Reliability and Security (QRS)*, IEEE, pp. 318–328, Prague, Czech Republic, July 2017.

[46] L. Zhao, Z. Shang, L. Zhao et al., "Siamese dense neural network for software defect prediction with small data," *IEEE Access*, vol. 7, pp. 7663–7677, 2018.

[47] X. Huo, Y. Yang, M. Li, and D.-C. Zhan, "Learning semantic features for software defect prediction by code comments embedding," in *Proceedings of the 2018 IEEE International Conference on Data Mining (ICDM)*, IEEE, pp. 1049–1054, Singapore, November 2018.

[48] X. Zhang, K. Ben, and J. Zeng, "Cross-entropy: a new metric for software defect prediction," in *Proceedings of the 2018 IEEE International Conference on Software Quality, Reliability and Security (QRS)*, IEEE, pp. 111–122, Lisbon, Portugal, July 2018.

[49] G. Navarro, "A guided tour to approximate string matching," *ACM Computing Surveys*, vol. 33, no. 1, pp. 31–88, 2001.

[50] S. Kim, H. Zhang, R. Wu, and L. Gong, "Dealing with noise in defect prediction," in *Proceedings of the 33rd International Conference on Software Engineering—ICSE'11*, pp. 481–490, New York, NY, USA, 2011.

[51] O. Abdel-Hamid, A.-r. Mohamed, H. Jiang, and G. Penn, "Applying convolutional neural networks concepts to hybrid NN-HMM model for speech recognition," in *Proceedings of the 2012 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, Kyoto, Japan, March 2012.

[52] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Proceedings of the Advances in Neural Information Processing Systems*, Lake Tahoe, NV, USA, December 2012.

[53] A. V. Phan and M. Le Nguyen, "Convolutional neural networks on assembly code for predicting software defects," in *Proceedings of the 2017 21st Asia Pacific Symposium on Intelligent and Evolutionary Systems (IES)*, IEEE, pp. 37–42, Hanoi, Vietnam, November 2017.

[54] A. V. Phan, M. Le Nguyen, and L. T. Bui, "Convolutional neural networks over control flow graphs for software defect prediction," in *Proceedings of the 2017 IEEE 29th International Conference on Tools with Artificial Intelligence (ICTAI)*, IEEE, pp. 45–52, Boston, MA, USA, November 2017.

[55] F. Dong, J. Wang, Q. Li, G. Xu, and S. Zhang, "Defect prediction in android binary executables using deep neural network," *Wireless Personal Communications*, vol. 102, no. 3, pp. 2261–2285, 2018.

[56] X. Yang, D. Lo, X. Xia, Y. Zhang, and J. Sun, “Deep learning for just-in-time defect prediction,” in *Proceedings of the 2015 IEEE International Conference on Software Quality, Reliability and Security*, IEEE, pp. 17–26, Vancouver, Canada, August 2015.

[57] L. Qiao and Y. Wang, “Effort-aware and just-in-time defect prediction with neural network,” *PLoS One*, vol. 14, no. 2, Article ID e0211359, 2019.

[58] X. Yang, D. Lo, X. Xia, and J. Sun, “TLEL: a two-layer ensemble learning approach for just-in-time defect prediction,” *Information and Software Technology*, vol. 87, pp. 206–220, 2017.