Computers & Security 114 (2022) 102572

Contents lists available at ScienceDirect

Computers & Security

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

An approach for predicting multiple-type overflow vulnerabilities based on combination features and a time series neural network algorithm

Zhangqi Zheng[a,b], Bing Zhang[a,b,*], Yongshan Liu[a,b], Jiadong Ren[a,b], Xuyang Zhao[a,b], Qian Wang[a,b]

[a] School of Information Science and Engineering, Yanshan University, Qinhuangdao, Hebei, PR China
[b] The Key Laboratory of software Engineering of Hebei Province, Hebei Province, Qinhuangdao City, China, 066004

ARTICLE INFO

Article history:
Received 30 April 2021
Revised 27 September 2021
Accepted 4 December 2021
Available online 7 December 2021

Keywords:
multitype
overflow vulnerability
combination features
time series
recurrent neural network

ABSTRACT

Overflow vulnerability is a common and dangerous software vulnerability that can lead to information theft, resource control, system collapse and other hazards. However, recent studies on predicting software overflow vulnerability have failed to specifically analyze factors and features that can lead to each type of overflow vulnerability and have only focused on binary classification problems rather than multiclassification problems, which are inefficient and time-consuming. Therefore, this paper proposes a multiple-type overflow vulnerability prediction method based on a combination of features and a time series neural network algorithm. First, by analyzing software overflow vulnerability features, a method is proposed to extract the internal vulnerability features of program source code. Then, an IFS set of internal vulnerability features of software overflow vulnerability is constructed. Second, an EFS set of external vulnerability features of software overflow vulnerability is extracted using a source code static analysis tool. A software overflow vulnerability feature library is constructed based on the IFS set and the EFS set. Finally, a multiple-type overflow vulnerability prediction method is constructed based on a time series bidirectional recurrent neural network after the symbol transformation and vector transformation of software overflow vulnerability features. Experiments show that the proposed method offers a higher precision, accuracy, recall rate, and F1 value. Moreover, this method can accurately detect the overflow vulnerability in actual software vulnerability predictions.

1. Introduction

Since the first buffer overflow attack occurred in 1988, overflow vulnerability has become a common and devastating threat to software because it can cause great harm to software security. There have been many studies assessing overflow vulnerabilities, but recent research on predicting software overflow vulnerability has failed to analyze factors and features that lead to each type of spillover vulnerability and has only focused on binary classification problems rather than multiclassification problems, which are inefficient and time-consuming. Common Weakness Enumeration (CWE) is a software security strategic project funded by the National Computer Security Department of the U.S. Department of Homeland Security. As the most authoritative source code defect research project, CWE has been recognized by an increasing number of professionals. CWE classifies and describes security vulnerabilities and defines them as different types according to the conditions of vulnerability occurrence and their triggering vulnerability (http://cwe.mitre.org/data/definitions/1000.html). According to the CWE Individual Dictionary Definition (4.4), there are more than 15 types of buffer overflow vulnerability. Therefore, to better understand overflow vulnerability, it is necessary to analyze the characteristics of different types of vulnerability and then predict these different types of overflow vulnerability in the software. Therefore, it is important to design an efficient method for overflow vulnerability prediction.

At present, software vulnerability detection methods can be divided into two types: binary vulnerability analysis and source code vulnerability analysis. As binary code is the final manifestation of most software, a software release is likely to have a binary execution program, so vulnerability analysis based on binary code has broad applicability compared with source code. However, a binary code execution program lacks program structure and type information, so it is difficult to detect a typical vulnerability pattern in binary code. In particular, it is difficult to determine buffer bound-

* Corresponding author.
E-mail address: bingzhang@ysu.edu.cn (B. Zhang).

https://doi.org/10.1016/j.cose.2021.102572
0167-4048/

aries on a stack accurately, and even accessing the jump address may not be reasonable. Thus, the accuracy of binary analysis has some disadvantages compared with source code analysis. Source code is the original form of most software, and its security defects are a direct source of software vulnerability. A vulnerability analysis of source code generally includes model extractions of program code, extractions of program detection rules, static vulnerability analysis and result analysis. This can result in an excessive false-positive rate in vulnerability detection if the program code model and the program detection rules are not extracted comprehensively. For example, using software metrics such as lines of code and cyclomatic complexity as inputs for a vulnerability prediction method (Ahmad et al., 2011, Schneidewind, 1992, Choudhary et al., 2018), while providing high prediction accuracy, often leads to a high false-positive rate, as software metrics only reflect the external features of code; thus, the extracted metrics do not completely express the features related to vulnerability.

The wide application of machine learning and deep learning in various industries has led to their combination with source code-based vulnerability analysis methods to predict software vulnerability. Although their prediction results have achieved desirable accuracy and correctness, a sound analysis of relevance between algorithm input and vulnerability prediction has been missing.

Based on software source code, this paper predicts multiple types of overflow vulnerabilities in software, analyzes internal features of code related to eight types of overflow vulnerability, extracts internal features such as sensitivity functions, logic control codes and variable codes from the source code, and analyzes the extracted internal features to form an internal vulnerability feature set. Then, by extracting the external features of the software and adding them to the internal vulnerability feature set, a software overflow vulnerability feature library is constructed. The software overflow vulnerability features are transformed into an embedding layer by vector transformation and other feature processing, and a multitype overflow vulnerability prediction method is built based on a bidirectional recurrent neural network with a time series.

The main contributions of this paper are as follows.

(1) The causes of eight kinds of software overflow vulnerability were analyzed: **Stack-based Buffer Overflow**, **Heap-based Buffer Overflow**, **Buffer Underwrite**, **Buffer Overread**, **Buffer Underread**, **Integer Overflow or Wraparound**, **Integer Underflow** and **Free of Pointer – not at Start of Buffer** were analyzed. According to these eight kinds of overflow vulnerability, a method for extracting internal features of the source code was proposed.
(2) Symbolic transformation and vectorization of the extracted vulnerability features were carried out. According to the software overflow vulnerability features, a bidirectional recurrent neural network based on time series was analyzed, and a multiple-type software overflow vulnerability prediction model was built using a bidirectional neural network.
(3) Our proposed method achieved superior accuracy, precision, recall, and F1 value when predicting overflow vulnerabilities in the Juliet dataset and successfully predicted the corresponding type of vulnerability in four real applications: *Linux Kernel*, *GNU LibreDWG*, *Espruino*, and *PDFResurrect*.

The rest of this paper is organized as follows. Section 2 describes the related work. Section 3 introduces the preliminaries of this study. Section 4 introduces the method. Section 5 details the experiments and results. Section 6 discusses the method. Section 7 offers the conclusion of this paper.

2. Related work

This section divides the related work into two parts: software vulnerability detection and research on software overflow vulnerability.

2.1. Research on Software Vulnerability Detection

Common software vulnerability detection methods include static analysis methods, dynamic analysis methods, and a combination of static and dynamic analysis methods (Rahimi and Zargham, 2013, Niu et al., 2013). Static analysis methods include source code static analysis methods and binary code static analysis methods. Dynamic analysis methods and combined dynamic and static analysis methods are mostly used for the analysis of binary files. Source code static analysis techniques mainly include taint analysis, symbolic execution, model checking, and theorem proving (Clause et al., 2007, Zhang et al., 2012, Li et al., 2013). At present, binary static analysis methods mainly exist in binary code comparison-based vulnerability analysis methods and pattern-based vulnerability analysis methods. Dynamic analysis mainly includes random fuzzing testing and intelligent fuzzing testing (Liu et al., 2020). Dynamic and static analysis technologies mainly include smart gray box testing and dynamic stain analysis.

At present, it is becoming increasingly popular to detect software vulnerability based on the source code of software. Dam et al. (Dam et al., 2019) proposed a defect prediction model method based on a deep learning tree. This model was built on a tree-structured long short-term memory network, which directly matched the abstract syntax tree of the source code. Shippey et al. (Shippey et al., 2019) used an abstract syntax tree to identify features of Java code containing defects. In source code-based approaches to detect software vulnerability, software measurement metrics have become a quantitative criterion for evaluating code quality. Previous studies have shown a correlation between software measurement metrics and software vulnerability (Liu and Traore, 2006). The main research methods for software measurement include the Halstead measurement method, McCabe measurement method, cyclomatic complexity measurement method, essential complexity measurement method, design complexity measurement method, integration complexity measurement method, C&K measurement method based on inheritance trees, MOOD measurement method, and cohesion measurement methods (Alves et al., 2016).

With the widespread application of machine learning and deep learning in various fields, Russell et al. (Russell et al., 2018) combined the features of C/C++ source code with random forest concepts to design a machine learning-based vulnerability detection system. Dam et al. (Dam et al., 2018) proposed a method based on a long short-term memory model. *LSTM* was used to automatically learn the semantics and syntactics of the code, and the *Random Forest* was used to predict vulnerability. Li et al. (Li et al., 2018) proposed a VulDeePecker method based on *LSTM* to detect software vulnerability. VulDeePecker trained code snippets related to library/API function calls based on a neural network and detected vulnerability caused by improper use of these calls. The related work on software vulnerability prediction is summarized in Table 1.

2.2. Research on Overflow Types of Software Vulnerability

Because overflow vulnerability is the most common and dangerous software vulnerability, it has always been a critical research direction of software vulnerability detection. In research using a binary code analysis method to detect overflow vulnerability, Cui

Table 1
Summary of the related work on vulnerability prediction of software.

Research articles	Year	Analysis methods	Tool	Testbeds	Algorithm	Assessment
Rahimi et al. (Rahimi and Zargham, 2013)	2013	Static	Compass	NVD		The number of potential overflow
Niu et al. (Niu et al., 2013)	2013	Dynamic		gird_(cm).wmf,mqyrf.mp3		The number of potential overflow
Clause et al. (Clause et al., 2007)	2007	Dynamic	PIN	Software(FIREFOX), Benchmark suite		The number of potential overflow
Zhang et al. (Zhang et al., 2012)	2012	Dynamic static combination	DynamoRIO	Software(IrfanView)		SPEC CINT2006
Li et al. (Li et al., 2013)	2013	Static		Juliet Test Suites		Precision, Recall, F1
Liu et al. (Liu et al., 2020)	2020	Static		NVD,CVE,ASTs	BiLSTM, Random forest	Precision, Recall, F-measure
Dam et al. (Dam et al., 2019)	2019	Static		PROMISE,Open source projects	Tree-LSTM, Random Forests, Logistic Regression	Precision, Recall, F-measure,ROC
Shippey et al. (Shippey et al., 2019)	2019	Static		Java systems	Naive Bayes, J48, Random Forest	Precision, Recall, F-measure, MCC
Liu et al. (Liu and Traore, 2006)	2006	Static	JProbe Profiler	Medical Record Keeping System		Residual, Actual, Fitted
Alves et al. (Alves et al., 2016)	2016	Static	Understand	Software patch file		The number of potential overflow
Russell et al. (Russell et al., 2018)	2018	Static	Clang,Cppcheck, Flawfinder	SATE IV Juliet Test,Debian,GitHub	CNN, RNN, Random Forest	Recall, PR, AUC, ROC, AUC, MCC, F1
Dam et al. (Dam et al., 2018)	2018	Static		Android Dataset,Firefox Dataset	LSTM, Random Forest	Precision, Recall, F-measure, AUC
Li et al. (Li et al., 2018)	2018	Static	Checkmarx	NVD,SARD Project	BLSTM	FPR,FNR,TPR,Precision,F1

et al. (Gui et al., 2014) proposed a software vulnerability detection technique to identify integer overflow vulnerability in binary executable files. This method combined dynamic taint tracking and target filtering. Dynamic taint tracking was used to reduce the variation space, and a target filtering function was used to filter test cases in the test case generation process. Yang et al. (Yang et al., 2015) proposed a platform called INDIO, which integrated pattern matching, vulnerability ranking, and selective symbolic execution to detect integer overflow vulnerability in Windows binary files. Mouzarani et al. (Mouzarani et al., 2016) proposed using concolic execution to calculate the vulnerability constraints of each execution path in a program and generated test data based on the calculated constraints to detect buffer overflows in the feasible execution path of the program.

In a study based on software source code analysis of buffer overflow vulnerability, Zhang et al. (Zhang et al., 2016) proposed a dynamic method to detect integer overflow and buffer overflow vulnerability. This method first used the characteristics of integer overflow to find integer-sensitive code areas using static analysis and then used selective symbol execution to explore these code areas and checked the security conditions on each receiving point to find safe errors. For suspicious integer overflow points, a POC was automatically generated. Hao et al. (Hao et al., 2016) proposed a technique to identify benign integer overflow through equivalence checking across multiple precisions. By comparing the effect of overflow integer operations in the real world with the same operations in the ideal world, a method called IntEQ was implemented based on the GCC compiler and Z3 solver to determine whether an integer overflow was benign.

Among the research approaches that used a combination of static analysis and dynamic testing, Ding et al. (Ding and Yuan, 2011) developed a buffer overflow vulnerability testing system. PE format files and dynamic link library files were detected separately using this system. Padmanabhuni et al. (Padmanabhuni and Tan, 2014) used simple rules to generate testing data, automatically confirmed some vulnerability through dynamic analysis, and analyzed the remaining data by mining static code attributes to detect software buffer overflow vulnerability.

Overflow vulnerability is the most common and dangerous software vulnerability. The root cause is data being added outside of the memory that is allocated to the buffer. As software becomes more complex in its functionality, code logic becomes more complex. There are actually many different scenarios that can lead to overflow vulnerability. Therefore, to better detect overflow vulnerability, a more thorough analysis of multiple types of overflow vulnerability is required. Previous studies of overflow vulnerability have detected few types of overflow vulnerability at the same time. Therefore, this paper proposes a method to predict multiple types of software overflow vulnerability based on an analysis of the characteristics of multiple overflow vulnerabilities. The related work on software overflow vulnerability prediction is summarized in Table 2.

3. Preliminaries

This paper analyzed how to extract the features of vulnerability and build a multiple-type overflow vulnerability prediction model. In this section, the preliminaries of predicting multiple-type overflow vulnerability are introduced in four aspects: (1) What causes software overflow vulnerability? (2) How are sensitive functions selected? (3) How are source code features extracted? (4) How are neural networks selected?

3.1. What causes software overflow vulnerability?

Overflow refers to a situation in which data are added outside of the memory block allocated to the buffer. The memory that software occupies on computers can be divided into **stack areas, heap areas, global areas, literal memories, and program codes** (Zhang, 2014), where the parameter length, parameter type, and data transfer code in the data are highly related to overflow vulnerability. As the reasons for triggering different types of overflow vulnerability are different, we analyzed the source code of a large amount of vulnerability and conducted feature analysis on 8 summarized types of overflow vulnerability as follows.

Table 2
Summary of the related work on overflow vulnerability prediction of software.

Research articles	Year	Analysis methods	Tool	Vulnerability	Testbeds	Assessment
Gui et al. (Gui et al., 2014)	2014	Dynamic static combination		Integer Overflow	Applications(Realplay.exe,Winamp.exe, Mspaint.exe)	The number of potential overflow
Yang et al. (Yang et al., 2015)	2015	Static		Integer Overflow	Programs(comctl32, gdi32,libpng,png2swf, jpeg2swf,libwav plugin)	FPR
Mouzarani et al. (Mouzarani et al., 2016)	2016	Static	STP,Valgrind, Fuzzgrind	Stack-based buffer overflow	Juliet Test Suites	TP,FP,TN,FN,Time
Zhang et al. (Zhang et al., 2016)	2016	Static		Integer Overflow	Juliet Test Suites	FP,FN
Hao et al. (Hao et al., 2016)	2016	Static	GCC,Z3	Integer Overflow	SPECINT 2000, SPECINT 2006, 7 software	FN
Ding et al. (Ding and Yuan, 2011)	2011	Dynamic static combination	BugScam, IDA Pro, OllyDbg	Buffer Overflow	100 PE-format files in WINDOWS XP system, Npdsplay.dll	The number of potential overflow
Padmanabhuni et al. (Padmanabhuni and Tan, 2014)	2014	Dynamic static combination	Pin, BODetect	Buffer Overflow	MIT Lincoln Laboratories test suite	TP,FP,TN,FN

(1) **Stack-based Buffer Overflow (O_1)**: Since local variables of a program are allocated on the stack, the size of memory occupied by all local variables should not exceed the size of the corresponding stack when running the program. Otherwise, a stack overflow will occur, which will lead to a system crash. In this case, an out-of-bounds array, a deep recursive for a function and a dead loop would result in a stack-based buffer overflow. Therefore, array length, recursion conditions and susceptibility to dead loops are significant features for this type of overflow.

A buffer size is fixed, as shown in Code 1, but there is no guarantee that the size of the string in $argv_{[1]}$ must not exceed the buffer size, so a stack buffer overflow occurs when the size of the string in $argv_{[1]}$ exceeds the buffer size.

(2) **Heap-based Buffer Overflow (O_2)**: A heap buffer overflow occurs when the amount of memory in use exceeds the amount of memory allocated by the function that allocates on the heap. As shown in Code 2, in C/C++ programs, heap space allocation and release functions should include malloc, calloc, realloc, new, free, etc. The subsequent code snippet allocates a fixed amount of heap memory to the buffer. However, there is no guarantee that the size of the string in $argv_{[1]}$ will not exceed the buffer size, so a string in $argv_{[1]}$ exceeding the buffer size will cause a heap buffer overflow. Thus, code features such as heap space allocation and release functions are highly relevant to this type of overflow.

(3) **Buffer Underwrite Overflow (O_3):** A buffer underwrite overflow usually occurs when a pointer or its index decrements to a position before the start of the buffer, when pointer arithmetic starts before a valid memory location, or when a negative index is used. As shown in Code 3, if function find () returns a negative value to indicate *ch* is not found in *srcBuf*, this may result in the buffer being rewritten, and transferring the data to *destBuf* will also fail. Here, the features of the function return values, pointer index and data transfer are related to this type of overflow.

(4) **Buffer Overread Overflow (O_4)**: A buffer overread overflow occurs when software reads data from outside the upper boundary of the buffer, such as adjacent memory, through a buffer access mechanism, such as indices or pointers. As shown in Code 4, the method processMessageFromSocket receives a message from the socket and places it in a buffer, which parses the contents of the buffer and places it in a structure containing the length and body of the message. A *for* loop is used to copy the message body to a local string. This code uses the message length variable *msgLength* as the end condition of the *for* loop without verifying that *msgLength* actually reflects the length of the message body. If the value of *msgLength* is longer than the actual size of the message body, it will overread the *for* loop and cause the buffer to be read out of bounds. Thus, the features of data transfer and data length are also relevant to this type of overflow.

(5) **Buffer Underread Overflow (O_5)**: Buffer underread occurs when software reads data from a buffer using buffer access mechanisms (such as an index or pointer) that refer to a memory location before starting the target buffer. This happens because a pointer or its index decrements to a position before starting the buffer, and pointer arithmetic starts before a valid memory location or a negative index is used. The features of the pointer index and pointer setting are related to this type of overflow.

(6) **Integer Overflow or Wraparound Overflow (O_6)**: An integer overflow or wrapping occurs when the value of an integer grows too large or the value becomes a minimal value or a negative number. If an integer overflow occurs when it is used to control a loop, it can pose a threat to the security of the software. The causes of an integer overflow include numeric types, numeric length, and judgment statements. As shown in Code 5, the code allocates a table of size *num_imgs*, but as *num_imgs* increases, a calculation to determine the list's size will eventually overflow.

(7) **Integer Underflow Overflow (O_7)**: an integer underflow occurs when one value is subtracted from another value, and the result is less than the minimum allowable integer value. The value produced by the integer value is not equal to the correct result. Therefore, the reasons for an **integer underflow** include numerical types, numerical length, and judgment statements. As shown in Code 6, the value of *i* is already the smallest possible negative value. The value of *i* will incorrectly become 2147483647 after subtracting 1.

(8) **Free of Pointer – not at Start of Buffer Overflow (O_8)**: This occurs when an application calls a *free* function on a pointer to a memory resource that has been allocated on the heap, but this pointer is not at the beginning of the buffer. Such a vulnerability can cause an application to crash or, in some cases, modify critical program variables or execute code. As shown in

Table 3
Code features of eight overflow types.

	O_1	O_2	O_3	O_4	O_5	O_6	O_7	O_8
variable length	√	√	√	√	√	√	√	√
variable types	√	√	√	√	√	√	√	√
data transfer	√	√	√	√	√	√	√	√
array length	√							
pointer setting			√		√			√
pointer index			√		√			√
functions return value			√					
recursion	√							
logic control	√							
memory allocation		√						√
memory release		√						√

Code 7, the code attempts to mark up a string and put it into an array using the *strsep* function, which inserts '\0' bytes instead of spaces or tabs. After the loop completes, each string in the *ap* array points to a position in the input string. Since the *strsep* function does not allocate any new memory, freeing the element in the middle of the array is equivalent to freeing the pointer in the middle of the input string. Then, *free (ap[4])* will lead to the free pointer not being at the start of buffer overflow. Therefore, code features such as functions that allocate memory on the heap, free functions, and pointers are relevant to this type of overflow.

In addition to the features of variable length, variable types, and data transfer code that are highly relevant to the overflow vulnerabilities, we categorized and mapped the code feature set (IFS) to each type of overflow vulnerability mentioned above. The results are shown in Table 3.

When there is an overflow vulnerability in a function in the source code, there may be multiple variables in the function. However, not all the variables are related to overflow vulnerability, but the variables in sensitive functions such as library APIs would be more likely to cause vulnerabilities (Ganapathy et al., 2005). Therefore, this paper selects the variable features of all parameters in the sensitive function.

3.2. How are sensitive functions selected?

The source code contains a large number of functions in standard function libraries. In this section, the functions related to overflow vulnerability are selected as sensitive functions from the standard library according to the **string memory function, dynamic allocation function, and memory manipulation function**. Taking the C/C++ language library as an example, sensitive functions related to overflow vulnerability are extracted in Table 4 as follows.

3.3. How are source code features extracted?

The basic unit of the program is a function. This section analyzed the program at the granularity of the functions. The functions in the program have characteristics of the internal code content and the external structure. Since the internal features contain the information of data operations related to the overflow vulnerability, the external features contain information about the metrics of the functions associated with the vulnerability. Therefore, the features related to multitype overflow vulnerability in the functions are extracted from both the internal and external aspects of the function.

In this paper, a feature set IFS related to various types of vulnerability is set according to the internal features of the functions. Simultaneously, data flow information and logic control dependencies of the program need to be fully considered when extracting internal vulnerability features, as detailed in 4.1.1. An external feature set EFS is set for the external features of the functions, as detailed in 4.1.2.

3.4. How are neural networks selected?

Because neural networks have powerful self-learning capabilities, strong associative storage and the ability to find optimal solutions at high speed, they are often used in areas such as image processing, speech recognition, and natural language processing. This section chooses neural networks as vulnerability prediction algorithms.

3.4.1. Time series neural network

Since functions can be encapsulated, recursively called and invoked, the parameters of function calls may be affected by both the code before this statement and the code after this statement. Therefore, the root causes of program overflow vulnerability are not necessarily the current code statement that triggers vulnerability. For example, if a code fails to define the data correctly, after multiple parameter passes, an execution of a statement that is far from the data defining statement triggers an overflow. Alternatively, data overflow occurs because of improper logical control. In this section, to preserve the context information, a recurrent neural network based on a time series is selected.

The primary purpose of an *RNN* recurrent neural network is to process and predict sequence data. In either fully connected neural networks or convolutional neural networks, data are derived from the input layer to the hidden layer and then passed to the output layer. The layers are fully or partially connected to each other. A recurrent neural network portrays a sequence to describe the current output related to the previous information. An *RNN* stores previous information and uses the previous information to influence the subsequent output. The *RNN* hidden layer input includes not only the output of the input layer but also the output of the hidden layer at the previous moment. Therefore, the *RNN* algorithm fits well with the need to preserve contextual information when predicting software vulnerability. Due to the great programming flexibility and the complexity of functions in current programs, there are cases where arguments are passed multiple times, leading to an overflow vulnerability in a particular statement that is far away from the parameter definition code. As the statements associated with overflow vulnerability become more distant in the context, the *RNN* loses the ability to learn connected information that is far away and suffers from long-term dependency.

An *LSTM* neural network solves the long-term dependency problem of the *RNN*. *LSTM* is a unique network structure with a forget gate, input gate, and output gate, which allows information to selectively influence the state of *RNN* at each moment. The role of the "gate" can shed invalid information as the statements associated with overflow vulnerability become more distant in the context. Therefore, *LSTM* is more suitable than an *RNN* as a neural network to predict software overflow vulnerability. The parameters of *LSTM* have greatly influenced its accuracy and performance. For example, Peng et al. (Peng et al., 2020) proposed an *LSTM* prediction optimization method based on fruit flies, which uses an intelligent algorithm to improve the prediction performance of *LSTM*. Gers et al. (Gers and Schmidhuber, 2001) proved the superior performance of *LSTM* in learning simple context-free and context-sensitive languages. However, the number of parameters of an *LSTM* neural network is four times that of an *RNN*, so there will be a risk of overfitting when the number of parameters is too large.

Based on *LSTM*, the *GRU* neural network merges the forget gate and the input gate into a single update gate and merges the data unit state and the hidden state simultaneously, making the model structure simpler than that of *LSTM*. An update gate and a reset

Table 4
Sensitive Function.

	Function
String memory function	*fscanf, vscanf, vsscanf, vfscanf, vsprintf, gets, getopt, getpass, strcpy, strecpy, strcat, strncat, strtrns, strlen, sprint, scanf, sscanf, strcmp, strncmp, strncpy, strlwr, strtok, strset, strstr, strchr, strrchr, strerror, strupr, strrev, streadd, realpath, strsep, etc.*
Dynamic allocation function	*malloc, calloc, free, alloc, realloc, etc.*
Memory manipulation function	*memset, memcpy, memmove, memset, memchr, etc.*

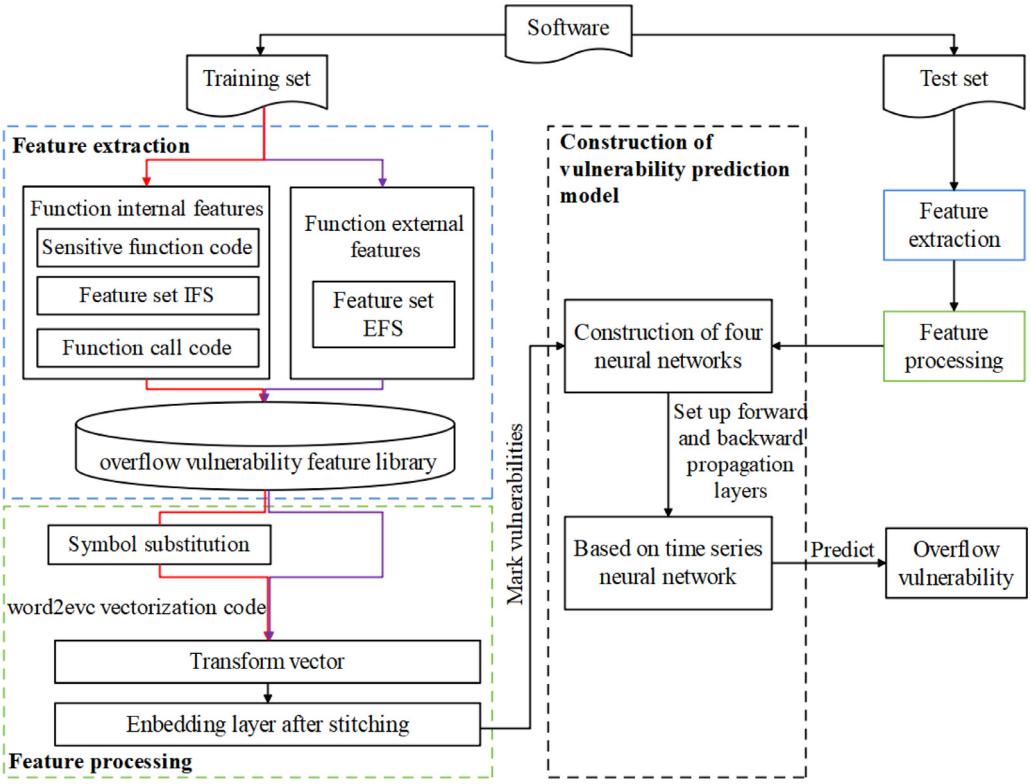


Fig. 1. The framework of the multiple-type overflow vulnerability prediction.

gate are set in the *GRU* model. The update gate controls the extent to which the state information of the previous moment is brought into the current state. In contrast, the reset gate controls how much information from the previous state is written to the current candidate set. Therefore, the *GRU* can solve the long-term dependency problem that exists in the *RNN*. A *GRU* only uses two gate switches, so the *GRU* can avoid the risk of overfitting based on maintaining better long-term learning capability. A *GRU* model can efficiently solve practical problems. For example, Chung et al. (Chung et al., 2014) proposed that *GRU* is better than *LSTM* in convergence time and required epoch. Han et al. (Han et al., 2021) proposed a K-means clustering algorithm combined with a *GRU* prediction method, which can learn more obvious feature information and improve prediction accuracy.

3.4.2. Bidirectional recurrent neural network

RNN, *LSTM*, and *GRU* can only predict the output of the next time based on the time sequence information of the previous time. However, in predicting software vulnerabilities, the output of the current time is related to the previous state and may be related to the future state.

The basic concept of a bidirectional recurrent neural network is that each training sequence is two recurrent neural networks forward and backward, and they are connected with an output layer. This structure provides complete past and future context information for each point in the input sequence of the output layer.

Therefore, to predict vulnerability based on context, this paper finally selects an overflow vulnerability solution using bidirectional recurrent neural network prediction software.

4. Method

This paper proposes a method that can detect eight types of overflow vulnerability. This section introduces the overall design concept of this method. First, an extraction method for internal and external features of the source function is introduced. Second, the feature processing of overflow vulnerability is presented. Finally, how this method can predict overflow vulnerability using a neural network based on time series is illustrated.

As shown in Fig. 1, this method extracts vulnerability features of programs (.C or .CPP file) from both internal features of functions and external features of functions in the source code. A method is proposed to extract the internal vulnerability features of programs and construct an internal vulnerability feature set for software overflow vulnerability. Then, the external software features extracted by the source code static analysis tool are added to the internal vulnerability feature set to build a software overflow vulnerability feature library. The internal vulnerability features are processed by replacing some of the function names and parameter

names with symbols, and Word2vec is used to transform the internal features represented by symbols into vector representations. The external vulnerability features in target functions are extracted according to the software metric, and all features are converted into vector form. After vulnerability tagging, this method forms a new embedding layer by splicing internal vulnerability features with external vulnerability features and feeds the spliced embedding layer into the neural network for model training.

4.1. Feature extraction

4.1.1. Internal function features

First, non-ASCII characters and comments are removed from the source code, and then the internal features of functions are extracted. The software internal vulnerability feature set is defined as IFS. IFS consists of the critical code of the software. The **critical code** is defined as follows.

Definition 1: Critical code. The critical code consists of several program statements that contain sensitive functions related to the overflow vulnerability in the language library, which is the code for all parameters of the sensitive functions, including the feature code in the feature set IFS. If this function calls another function in the program, the critical code includes the related code in the called function.

The **critical code** needs to include code containing sensitive functions, such as heap space allocation, heap space release, and numeric length handling. Sensitive functions are chosen because most of the causes of software overflows are related to incorrect use of functions that store, allocate, and manipulate memory. Some sensitive functions significantly impact overflow vulnerability. For example, **Heap-based Buffer Overflow** occurs when the heap memory is out of bounds, mainly because the memory being manipulated exceeds the size allocated on the heap by calloc, malloc, and new memory function, resulting in the subsequent failure of calloc, malloc, and new operation. **Free of Pointer – not at Start of Buffer** is caused by the application calling *free* on a pointer to a memory resource that has been allocated on the heap when the pointer is not at the beginning of the buffer. Therefore, the extraction of sensitive functions can quickly and accurately extract the characteristics relevant to vulnerability.

The **critical code** contains the parameter characteristics of sensitive functions and the characteristics of the feature set IFS code. For example, if the array index exceeds the length of the array definition, other variables will be overwritten, resulting in a stack overflow. An **Integer Overflow or Wraparound** occurs when an integer value grows too large. The control logic and dependencies can also lead to overflow vulnerability, so logical control statements can significantly impact overflows. For example, a program with too many recursive calls or dead loops due to logic control can easily lead to a **Stack-based Buffer Overflow**. The lines of code extracted when extracting **critical code** contain the following logical control keywords: for, do, while, break, and continue for a loop statement; if, else, and goto for a conditional statement; switch, case, and default for a switch sentence; return for a return statement.

The **critical code** defines that if the current function calls other functions in the program, the **critical code** will include the relevant code in the called function. Software has a very large amount of code and complex control logic. For example, C++ is an object-oriented language with many object-oriented features, such as encapsulation, inheritance, polymorphism, calls to other functions in this file, or even functions in other files, and often occurs in functions. Because of this phenomenon, an in-depth extraction strategy is adopted in this method. For example, VARa is the parameter of sensitive function FUNA that needs to be extracted in FileA.cpp. If the parameter VARa is the return value VARb of the function FUNB in FileB.cpp, then the function name of the function FUNB and all lines of code related to the return value VARb need to be extracted. If the value of VARb in function FUNB is the return value of VARc from function FUNC in FileC.cpp, the function name of the function FUNC and all lines of code related to the return value VARc also need to be extracted. However, due to the complexity of function calls and the possibility of too many function calls in real programs, in this method, the depth of mutual calls between different functions or files is limited. Up to five different function calls in different source files or functions are studied in this section.

In this section, the alias of the source code is code, the alias of function is F ($f \in$ code), and the alias of sensitive functions is $S=\{s_1, s_2,..., s_n\}$. The alias of parameters is p. The alias of **critical code** slices is D. The algorithm Slicef for extracting **critical code** is as follows.

Algorithm: Slicef (f)
Input: A program code $=\{f_1, f_2,...,f_i,..., f_n\}$, a set S
Output: A set D of F
(1) D $\leftarrow \Phi$
(2) **for** each function $f_i \in$ code **do**
(3) **While** logic control code$\in$current code
(4) **if** current code $\notin$ D **then**
(5) D = D + current code
(6) **end if**
(7 **While** $s_i \in$ current code
(8) **if** s_i not contains the parameters **and** current code $\notin$ D **then**
(9) D = D + current code
(10) **end if**
(11) **if** s_i contains the parameters **then**
(12) **for** each line code$\in f_i$ **do**
(13) **if** parameters $\in$ code and current code $\notin$ D **then**
(14) D = D + current code
(15) **end if**
(16) **end if**
(17) **if** other custom function$\in$current code **then**
(18) **if** $s_i \in$ other custom function **then**
(19) **Slicef** (other custom function)
(20) **end if**
(21) **end if**
(22) The code lines in D are arranged in the order of the code lines in the program

The **critical code** extraction method in the method proposed in this paper is as follows: A line of code is extracted as long as it satisfies the definition of **critical code**. Each critical line of code is extracted only once. All **critical code** extracted is stored in the order in which the code appears in the source code. For a function, the D set extracted by the slicef algorithm is the internal feature set IFS of the function.

4.1.2. External function features

The extraction of external features is based on the traditional software measurement method, and the measurement index is extracted with the function granularity. The software external vulnerability feature set is defined as EFS.

In the method proposed in this section, the selection of feature extraction focuses on three metrics: Cyclomatic, CyclomaticModified, and CyclomaticStrict. Cyclomatic, CyclomaticModified, and Cyclomaticstrict are the three methods of calculating cyclomatic complexity. The calculation formula is shown in Formula 1–3.

Cyclomatic: Cyclomatic complexity under standard calculation approach.

$$V(G) = E - N + 2P \tag{1}$$

E is the number of edges in a control flow graph, N is the number of nodes in the control flow graph, and P represents the number of connected components of the graph. The number of components of a graph is the maximum set of connected nodes. If the control flow graph is connected, then P is 1 by default.

Table 5
External feature.

Feature	Specific content of feature
AltCountLineCode	Number of lines containing source code, including inactive regions.
AltCountLineComment	Number of lines containing comment, including inactive regions.
CountInput	Number of calling subprograms plus global variables read.
CountLine	Number of all lines.
CountLineCode	Number of lines containing source code.
CountLineCodeDecl	Number of lines containing declarative source code.
CountLineCodeExe	Number of lines containing executable source code.
CountLineComment	Number of lines containing comment.
CountLinePreprocessor	Number of preprocessor lines.
CountOutput	Number of called subprograms plus global variables set.
CountPath	Number of possible paths, not counting abnormal exits or gotos.
CountPathLog	log10, truncated to an integer value, of the metric CountPath.
CountSemicolon	Number of semicolons.
CountStmt	Number of statements.
CountStmtDecl	Number of declarative statements.
CountStmtExe	Number of executable statements.
Cyclomatic	Cyclomatic complexity.
CyclomaticModified	Modified cyclomatic complexity.
CyclomaticStrict	Strict cyclomatic complexity.
Essential	Essential complexity.
Knots	Measure of overlapping jumps.
MaxEssentialKnots	Maximum Knots after structured programming constructs have been removed.
MaxNesting	Maximum nesting level of control constructs.
MinEssentialKnots	Minimum Knots after structured programming constructs have been removed.
RatioCommentToCode	Ratio of comment lines to code lines.

CyclomaticModified is the second method of the cyclomatic complexity calculation.

$$V(G) = N_{dn} + 1 \tag{2}$$

N_{dn} is the number of decision nodes.

CyclomaticStrict is the third computing method of cyclomatic complexity.

$$V(G) = R \tag{3}$$

R is the number of areas of the plane divided by the control flow diagram.

In this section, 25 features, such as ALTCountLineCode, ALTCountLineComment and CountInput, are selected for research. The external feature set is EFS={AltCountLineCode, AltCountLineComment, CountInput, CountLine, CountLineCode, CountLineCodeDecl, CountLineCodeExe, CountLineComment, CountLinePreprocessor, CountOutput, CountPath, CountPathLog, CountSemicolon, CountStmt, CountStmtDecl, CountStmtExe, Cyclomatic, CyclomaticModified, CyclomaticStrict, Essential, Knots, MaxEssentialKnots, MaxNesting, MinEssentialKnots, RatioCommentToCode}. The details of the external features are shown in Table 5.

4.2. Feature processing of overflow vulnerability

After feature extraction, IFS and EFS are combined to build the software overflow vulnerability feature library. The features in the feature library are divided into symbol substitution, vector transformation, and splicing embedding layers.

4.2.1. Symbol substitution

To improve the generalization capability of the prediction method, the extracted IFS is transformed into symbols. The objects replaced in this step are function names and parameter names. Function names include the custom functions in a program and insensitive functions in the function library. Arguments are all arguments of the functions in a code slice, including pointer type. An example of a code slice is shown in Code 8.

Function names in each code slice are replaced with symbol names that represent the function (for example, "FUN0", "FUN1"). The code slice contains memset, memmove, and printLine functions. Since memset and memmove are sensitive functions, only the name of the printLine function is replaced with FUN0. The code slice after the symbol replacements is Code 9.

The parameter names in code slices are replaced with symbol names that represent the parameters (for example, "VAR0", "*VAR0"). The code slice of the example function contains four parameters: data, dataBuffer, A, and dest. Data with VAR1, dataBuffer with VAR2, A with VAR3, 100–1 with VAR4 and dest with VAR5 are replaced in the order in which they appear in the code snippet. The code slice after the symbol replacements is as follows Code 10.

4.2.2. Vector transformation

After the symbol replacements of internal features, there are many identifiers, keywords and symbols in every code slice. To enable neural network models to better learn these features and mine the connection of code context information (Bengio et al., 2003), the method proposed in this section transforms words into vector representations.

First, word segmentation was applied to features in IFS. Because symbols are an essential part of the code (for example, <, >, +, -), the feature code symbols were preserved during word segmentation. Second, the *word2vec* tool was used to encode and vectorize the segmented words to obtain the word embedding of each word (Mikolov et al., 2013). The CBOW model of word2vec was adopted in this section. In the word vectorization process, the word vector of the target words was determined by learning the word vector corresponding to the context-related words. The specific process is shown in Fig. 2.

The code " VAR1 * VAR5 = (VAR1*) ALLOCA (100*sizeof (VAR1)); " is offered as an example. A forward propagation process of the CBOW model is as follows: The radius is set to 3, and the central word ALLOCA is represented as W(t) when vector transformations W(t-3), W(t-2), W(t-1), W(t+1), W(t+2) and W(t+3) are used to represent "VAR1", "*", ")" before ALLOCA and "(", "100" and "*" after ALLOCA. Take one-hot encoded vectors W(t-3), W(t-2), W(t-1), W(t+1), W(t+2), W(t+3) as input. After constructing the dictionary, set the input layer matrix as Win. The size is |v|*d. |v| is the size of a dictionary. d is the dimension of the word vector. The v

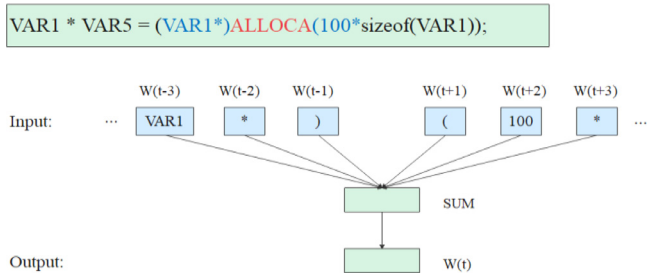


Fig. 2. Vector transformation process.

vector is a row of W_{in}. The process of SUM is expressed as follows.

$$w_i^T W_{in} = v \tag{4}$$

Multiply the one-hot encoded vector w_i with W_{in}. The row number corresponding to the word w_i in the input matrix Win is the index number in the dictionary. A dense vector is obtained after input layer operations. Let the matrix of the P_{middle} output layer be W_{out} and the size of the output matrix be $d^*|v|$; then, the output vector is P_{out}.

$$P_{out} = P_{middle}^T W_{out} \tag{5}$$

The values of each dimension of Pout are represented as the context W(t-3), W(t-2), W(t-1), W(t+1), W(t+2), W(t+3) The output word ALLOCA is the logit probability of each word in the dictionary, and the probability of each word is obtained after applying a softmax function.

4.2.3. Splicing the embedding layer

This section adopts the method of simultaneously feeding multiple classes of embedding into the neural networks when predicting overflow vulnerability. First, *Word2vec* is used to transform all the tokens in the internal vulnerability features of the source code into an embedding vector with information about overflow vulnerability features. Second, all the embedding vectors with vulnerability feature information are stitched together into an embedding layer containing internal vulnerability features (Guo and Berkhahn, 2016). The dimension of internal vulnerability embedding is set according to the maximum number of internal features. Similarly, each dimension extracted from the external vulnerability feature set EFS of the source code is transformed into a fixed dimension embedding vector with overflow vulnerability feature information. The embedding vector with vulnerability feature information is then joined into an embedding layer containing external vulnerability features. The dimensions of external vulnerability embedding are set according to the number of external feature sets multiplied by the number of fixed dimensions.

In this section, the internal and external embedding layers of the feature are spliced, and stitched embedding is used for the input of the neural network (Grbovic and Cheng, 2018). The embedding structures of the internal and external features are shown in Fig. 3.

4.3. Prediction for overflow vulnerability

4.3.1. Marking vulnerability

This paper studied eight types of overflow vulnerability: **Stack-based Buffer Overflow**, **Heap-based Buffer Overflow**, **Buffer Underwrite**, **Buffer Overread**, **Buffer Underread**, **Integer Overflow or Wraparound**, **Integer Underflow**, **Free of Pointer – not at Start of Buffer**. According to the eight types of vulnerability study in this section, the extracted **critical code** slices are marked into nine types. The specific steps are as follows.

(1) Code slices are classified based on whether there is a potential for vulnerability to occur. A **critical code** is marked as no overflow vulnerability if there is no vulnerability to an overflow. Otherwise, the **critical code** is marked as having an overflow.
(2) For code slices with hidden vulnerability, we classify them according to the types of overflow involved in code slices. For example, a code slice calls free on a pointer to a memory resource allocated on the heap, but the pointer is not at the beginning of the buffer. The overflow type involved in the code slice should be determined to be **Free of Pointer – not at Start of Buffer**.

4.3.2. Building prediction model

The embedding layer contains sensitive functions, parameters, program control, function calls, and other information related to the overflow vulnerability of the source code. This method uses the embedding layer as the neural network input and employs a time series-based bidirectional neural network to predict overflow vulnerability in the program. In bidirectional neural networks based on time series, bidirectional *RNN*, bidirectional *LSTM*, and bidirectional *GRU* are selected in this section to predict the overflow vulnerability of software.

This section takes a bidirectional *GRU* as an example to describe how to predict overflow vulnerability. Since bidirectional *GRU* modeling based on time series can retain the contextual information of the input data, an embedding layer with overflow vulnerability information as the input to the bidirectional *GRU* neural network can predict software overflow vulnerability more accurately. Sensitive functions, parameters, program control, function calls, and other information related to overflow vulnerability are retained in the bidirectional *GRU* as the context distance increases.

There are four layers in the bidirectional recurrent neural network: layer one is the input layer of the neural network, layer two is the layer of forward propagation, layer three is the layer of back propagation, and layer four is the output layer. There are six kinds of data that propagate in the neural network. Fig. 4 shows a bidirectional recurrent neural network based on time expansion.

First, the spliced embedding is fed to the first layer, The input information is propagated to the second and third layers of neural network, where i1 is propagated to the second layer of neural network and i3 is propagated to the third layer of neural network; Second, during the forward propagation process of the *GRU*, it not only receives the i1 input from the first layer but also receives the semantic information i2 related to the vulnerability, then uses the reserved semantic information of the second layer as i4 to the output layer; Next, during the back-propagation process of the third layer, it not only receives the i3 propagated by the first layer but also accepts the semantic information i5 related to the vulnerability below as input, and the reserved semantic information of the third layer propagated to the output layer is i6; Finally, the input layer combines the forward propagation i4 and the backward i6 to predict software overflow vulnerability.

The update gate is used in each *GRU* neural network to control the extent to which state information of the previous moment is carried into the current state. If the data above contain information about overflow vulnerability, the update gate will set a larger value. That is, the current moment can retain more information related to the previous moment. The reset gate is used in the *GRU* neural network to control how much information from the previous state is written to the current candidate set. If the data above do not contain much information about the overflow vulnerability, the smaller the reset gate will be set, so less information from the previous state is written.

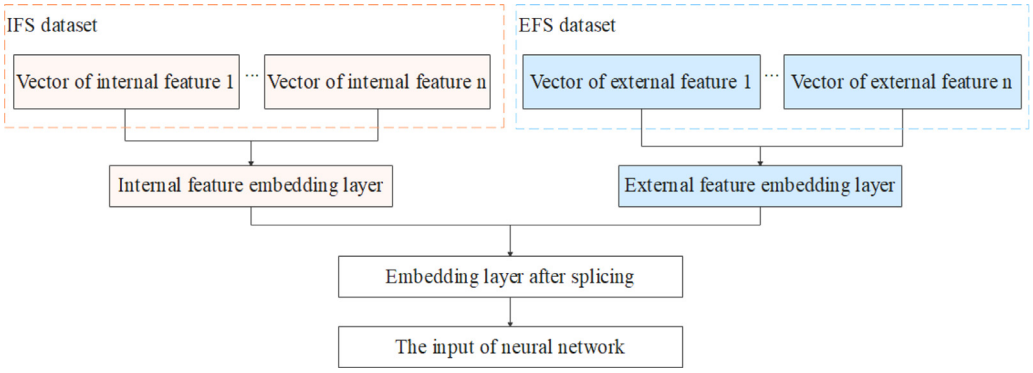


Fig. 3. Embedding processing method.

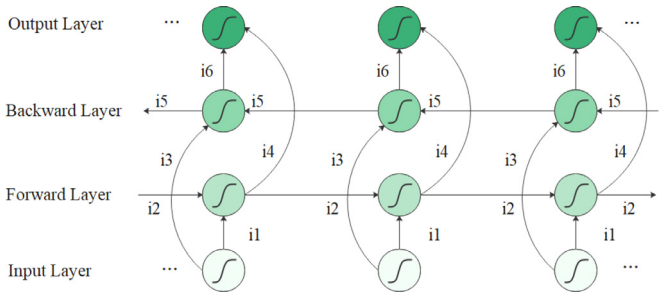


Fig. 4. Bidirectional recurrent neural network based on time expansion.

Table 6
The number of different types of functions in the dataset.

Vulnerability type	number
Stack-based Buffer Overflow	2645
Heap-based Buffer Overflow	1863
Buffer Underwrite	989
Buffer Overread	829
Buffer Underread	1043
Integer Overflow or Wraparound	1086
Integer Underflow	866
Free of Pointer – not at Start of Buffer	306
Not prone to overflow type vulnerability	8454

5. Experiments and results

In this experiment, executable programs containing eight types of overflow vulnerability were extracted from the Juliet dataset of the NIST library as experimental data. All the programs selected in this experiment can be compiled and run on Microsoft Windows using *Microsoft Visual Studio*. In this section, experiments are introduced from three aspects: dataset, experimental prediction results, and real software verification.

5.1. The Dataset

In this chapter, in the Juliet dataset of the NIST library (https://samate.nist.gov/SRD/view.php), executable programs containing eight kinds of overflow vulnerability were extracted as experimental data. In this dataset, there are 8 types of overflow vulnerabilities, and the number of functions without overflow vulnerabilities is shown in Table 6.

The dataset contains a total of 18,081 functions, among which the proportion of samples prone to vulnerability and samples not prone to vulnerability is 53.24%: 46.76%. In this experiment, to prevent overfitting in the experiment, 80% of the dataset data were randomly extracted as the experimental training set, and the remaining 20% of the data were extracted as the experimental prediction set.

5.2. Experimental prediction results

In this experiment, *Keras* was used to implement a time series-based bidirectional neural network. The experimental environment for algorithm implementation and dataset validation was mainly a Windows operating system. The experimental procedures were as follows.

- According to the feature set IFS, the vulnerability features of functions in the source code of executable programs were extracted. Based on the definition of the **critical code**, the **critical code** of functions was extracted according to the Slicef algorithm.
- Based on the external vulnerability feature set EFS, the external features of the functions were extracted using the static analysis tool of the Understand software.
- Non-ASCII characters and comments were removed from the code slices, and the function names and parameter names of each **critical code** segment were replaced with symbol names in a one-to-one correspondence. The function names include the names of the custom functions in the program and the function names of the nonresearch sensitive functions. Parameters are the parameters in all the functions in the code slice, including pointer type.
- Word segmentation was performed based on retaining the symbols in the feature code, and word2vec was used to obtain the word embedding of each word in the **critical code** and convert internal and external features into vector representations.
- The internal and external features of vector representation were transformed into an embedding layer, and the two embedding layers were stitched together.
- The extracted **critical code** segments were marked into nine types.
- A time-based bidirectional neural network was established to predict overflow type vulnerability. The input layer uses mosaic embedding as input. The middle layer was the forward propagation neural network and the backward propagation neural network based on time series, and the output layer was the type of software overflow vulnerability.

The process of setting the experimental initial parameters is as follows. **First,** determine the necessary parameters, such as the size of the embedded vector and output the dimension. The embedded vector size is set to 128 according to the dimension of the feature vector converted by word2vec. The dimensions of the input and middle layers of the neural network are set to 128. Because the data in this paper are divided into 9 categories, the output dimension of the last layer of the neural network is 9.

Table 7
Experimental results of *RNN*, *LSTM*, and *GRU* bidirectional recurrent neural network.

Neural Network	Accuracy (%)	Precision (%)	Recall rate (%)	F1 (%)
Bidirectional RNN	82.22	64.57	63.43	62.43
Bidirectional LSTM	98.14	97.08	97.07	97.05
Bidirectional GRU	96.95	95.22	94.24	94.69

Second, the activation functions of the middle layer and the output layer are considered. Because the sigmoid function can map variables between 0 and 1, it has smoothness and accessible derivation properties. Therefore, the activation function of the middle layer is set to sigmoid. Since this paper is multiclassification, the activation function of the output layer is a softmax function.

Finally, consider three parameters: optimizer, kernel initializer, and recurrent initializer. The optimization algorithm optimizer is set to SGD. The kernel initialization method defines the method of setting the initialization weight for the Keras layer. In this experiment, glorot uniform distribution initialization (Gal and Ghahramani, 2015) is adopted. The initialization method of the Recurrent_Initializer cyclic core uses a random orthogonal matrix which is set to orthogonal (Glorot and Bengio, 2010).

The accuracy rate, precision rate, recall rate, and F1 value were used to evaluate the vulnerability detection system. This paper first tests *RNN*, *LSTM*, and *GRU* bidirectional recurrent neural networks according to the initial parameters. The results are shown in Table 7.

From Table 7, the prediction results of the *RNN* bidirectional recurrent neural network were inferior to those of the *LSTM* and the *GRU* bidirectional recurrent neural network in terms of accuracy, precision, recall rate, and F1 value. Therefore, it can be proven that as the distance between the relevant statements triggering overflow vulnerability in the context increases, the bidirectional *RNN* will lose its ability to learn information related to the features of overflow vulnerability. The bidirectional *LSTM* and the bidirectional *GRU* outperform the bidirectional *RNN* in predicting software overflow vulnerability.

This experiment focuses on the analysis of vulnerability prediction results based on *LSTM* and *GRU* bidirectional recurrent neural networks. To verify the effectiveness of the feature combination method more clearly, three groups of comparative experiments were carried out with *LSTM*, and *GRU* bidirectional recurrent neural networks.

The first group of feature comparison experiments: This comparison experiment aims to verify the rationality of extracting **critical code**s and to verify the impact of logic control statements on buffer overflow vulnerabilities.

IF: When predicting software multitype buffer overflow vulnerabilities based on a time series neural network algorithm, the internal features were extracted according to the **critical code** extraction method, but the internal features of the logic control code in IFS were not mentioned during feature extraction.

IF_L: When predicting software multitype buffer overflow vulnerabilities based on a time series neural network algorithm, the internal features are extracted according to **critical code** extraction methods, and all internal features in IFS are extracted during feature extraction.

The second group of comparative experiments: This group of comparative experiments verified the feasibility of code features after symbol replacements.

IF_L_S: Based on **IF_L** in the first set of experiments, the extracted internal features were converted into symbolic form, and then the buffer overflow was predicted based on this feature.

The results of the first set of experiences are compared, and the effectiveness of the code features are analyzed after symbol replacement.

The third group of comparative experiments: This group of comparative experiments verified that the inclusion of external features provides a more reasonable and comprehensive representation of vulnerability features.

Based on the second group of experiments, a prediction model with internal and external features of logic control code after symbol conversion was constructed.

IF_L_S_E: Based on **IF_L_S** in the first set of experiments, external features were added during feature extraction, and then the buffer overflow was predicted based on the combination of features.

The results of the second group of experiments were compared to analyze whether the addition of external features can represent vulnerability features more reasonably and comprehensively.

To further improve the effect of predicting vulnerabilities. Based on the initial parameters, the learning rate of *LSTM* and *GRU* bidirectional recurrent neural network was adjusted, the number of network layers was increased, and the batch size and optimal grid search parameters were changed. Based on many iterative trials, to get the best effect, the optimal parameters of *LSTM* and *GRU* bidirectional recurrent neural network are set as below in this paper. Embedding vector size: input_dim = 128. Parameters of *LSTM* and *GRU*: units = 128, kernel_initializer = glorot_uniform, recurrent_initializer = orthogonal, recurrent_dropout = 0.1. Dense layer: units = 128, activation = sigmoid. Output: units = 9, activation = softmax. Noptimizer = adam. The experimental results are shown in Table 8.

An analysis of the results of the three groups of comparative experiments based on the *LSTM* and *GRU* bidirectional recurrent neural networks is shown below.

5.2.1. Analysis of the first group of feature comparison experiment results

In the method of bidirectional *LSTM* used to predict multiple types of buffer overflow vulnerability based on **IF** features, the accuracy rate was 96.32%, the precision rate was 96.41%, the recall rate was 96.50%, and the F1 value was 96.36%. In the method of bidirectional *LSTM* used to predict multiple types of buffer overflow vulnerability based on **IF_L** features, the accuracy rate was 98.50%, the precision rate was 98.21%, the recall rate was 97.47%, and the F1 value was 97.75%. In the method of bidirectional *GRU* used to predict multiple types of buffer overflow vulnerability based on **IF** features, the accuracy rate was 96.82%, the precision rate was 97.59%, the recall rate was 95.76%, and the F1 value was 96.54%. In the method of bidirectional *GRU* used to predict multiple types of buffer overflow vulnerability based on **IF_L** features, the accuracy rate was 98.45%, the precision rate was 98.18%, the recall rate was 97.50%, and the F1 value was 97.75%.

Compared with the results of predicting multiple types of buffer overflow vulnerability based on **IF** features, the four evaluation indicators in the results of predicting multiple types of buffer overflow vulnerability based on **IF_L** features had corresponding improvements, so this comparative experiment verifies the rationality and effectiveness of extracting **critical code** in this paper. The effectiveness of the logic code feature in predicting buffer overflow vulnerabilities is verified.

5.2.2. Analysis of the second group of comparative experimental results

In the method of bidirectional *LSTM* used to predict multiple types of buffer overflow vulnerability based on **IF_L_S** features, the accuracy rate was 94.99%, the precision rate was 93.30%, the recall rate was 95.14%, and the F1 value was 94.06%. In the method

Table 8
Experimental results of *LSTM*, and *GRU* bidirectional recurrent neural network.

Neural Network	Feature combination	Accuracy (%)	Precision (%)	Recall rate (%)	F1 (%)
Bidirectional	IF	96.32	96.41	96.50	96.36
LSTM	IF_L	98.50	98.21	97.47	97.75
	IF_L_S	94.99	93.30	95.14	94.06
	IF_L_S_E	98.72	97.81	97.88	97.75
Bidirectional	IF	96.82	97.59	95.76	96.54
GRU	IF_L	98.45	98.18	97.50	97.75
	IF_L_S	97.95	97.70	96.97	97.25
	IF_L_S_E	99.19	98.43	98.62	98.51

of bidirectional *GRU* used to predict multiple types of buffer overflow vulnerability based on **IF_L_S** features, the accuracy rate was 97.95%, the precision rate was 97.70%, the recall rate was 96.97%, and the F1 value was 97.25%.

Experiments show that compared with the first group of experiments, the second group still maintained good prediction results after symbol conversions, which further verified the effectiveness of the code features after symbol replacements.

5.2.3. Analysis of the third group of comparative experimental results

In the method of bidirectional *LSTM* used to predict multiple types of buffer overflow vulnerability based on **IF_L_S_E** features, the accuracy rate was 98.72%, the precision rate was 97.81%, the recall rate was 97.88%, and the F1 value was 97.75%. In the method of bidirectional *GRU* used to predict multiple types of buffer overflow vulnerability based on **IF_L_S_E** features, the accuracy rate was 99.19%, the precision rate was 98.43%, the recall rate was 98.62%, and the F1 value was 98.51%.

Compared with the second group of experimental results, the four evaluation indicators maintained good prediction results after adding external features, and most of the indicators had improved. The experiment proves that the addition of external features based on internal features can represent vulnerability features more comprehensively.

Three groups of experiments proved that the prediction results based on the **IF_L_S_E** feature combination prediction method were desirable. The *GRU* transfers the hidden state directly to the next unit, while *LSTM* packages the hidden state with a memory cell. In theory, the *GRU* can avoid the risk of overfitting better than *LSTM*. The effect of a bidirectional neural network based on a *GRU* was better than that based on *LSTM*. To further analyze the effect of *LSTM* and *GRU* bidirectional neural networks in the multitype buffer overflow vulnerability prediction method proposed in this paper, in this experiment, each type of buffer overflow vulnerability data in the dataset was extracted according to the proportions of 20%, 40%, 60%, and 80% for the experiments. The experimental results are shown in Fig. 5.

As shown in Fig. 5, with the increase in datasets, the accuracy, precision, recall rate, and F1 score of the bidirectional *LSTM* and bidirectional *GRU* prediction results tend to increase. Experiments show that with the increase in datasets, the prediction of multiple types of buffer overflow software vulnerability based on bidirectional *GRU* was better. For different proportions of data, the accuracy rate, precision rate, recall rate, and F1 value of the bidirectional *GRU* were always higher than those of the bidirectional *LSTM* and maintained a stable growth trend.

The precision, recall rate, and F1 values of various overflow vulnerabilities predicted by the bidirectional *GRU*-based **IF_L_S_E** feature combination prediction method are shown in Table 9.

As shown in Table 9, based on *GRU* bidirectional recurrent neural networks, this method performed well in predicting **Integer Overflow or Wraparound**, **Integer Underflow**, and **Free of Pointer – not at Start of Buffer** vulnerability. The reason why this method performed well on these three types of vulnerabilities was fully proven. When analyzing the causes of these three types of vulnerabilities in C language programs, this paper proposes some features, such as the type and length of extracted parameters, for these two types of vulnerabilities.

However, The **Buffer Underwrite** and **Buffer Underread** were relatively poorly predicted. This was because only variable length, variable types, data transfer, pointer setting, pointer index, and function return value were analyzed as the internal features in the **Buffer Underwrite** vulnerability prediction. Only variable length, variable types, data transfer, pointer setting, and pointer index were analyzed as the internal features in the **Buffer Underread** vulnerability prediction. It is not enough to analyze the causes of these two types of buffer overflow vulnerabilities in C language programs. The internal features of these two types of overflow vulnerabilities have not been accurately extracted. Regardless, prediction performance became poor when features associated with vulnerability were extracted insufficiently.

5.3. Verification in Real Software

To further verify the rationality and effectiveness of the multitype overflow vulnerability prediction method based on combined features and time-series neural network algorithms proposed in this paper, this method was applied to predict overflow vulnerability in four pieces of software: *Linux kernel*, *GNU LibreDWG*, *Espruino*, and *PDFResurrect*. This method successfully predicted the vulnerability in the four software programs. The verification results are shown in Table 10.

- *Linux kernel* is an open-source operating system. In the multiple-type overflow vulnerability prediction method based on combination features and a time series neural network algorithm, the rose\u recvmsg function in net/rose/af\u rose.c was predicted as stack buffer overflow vulnerability based on bidirectional *LSTM* and bidirectional *GRU*. The China National Vulnerability Database verified that the rose_recvmsg function was indeed vulnerable to a **Stack-based Buffer Overflow** vulnerability (CNVD-2013–04137). The rose_recvmsg function of the *Linux kernel* rose subsystem does not initialize all the sockaddr_rose/full_sockaddr_rose members when filling the sockaddr information, nor does it initialize the bytes of the structure for alignment. This can lead to net/socket.c leaking uninitialized kernel stack information to user space. Currently, the *Linux kernel* supplier has issued a security bulletin and patch information to fix this vulnerability.
- LibreDWG is a C language library of GNU for processing DWG files. In *GNU LibreDWG* 0.9.3 and earlier versions, using the multiple-type overflow vulnerability prediction method based on combination features and a time series neural network algorithm, the bit_write_UMC function in src/bits.c was predicted to be prone to **Stack-based Buffer Overflow** vulnerability based on bidirectional *LSTM* and bidirectional *GRU*. The China National Vulnerability Database verified that the bits.c file in *GNU LibreDWG* 0.9.3 and earlier versions had **Stack-based Buffer**

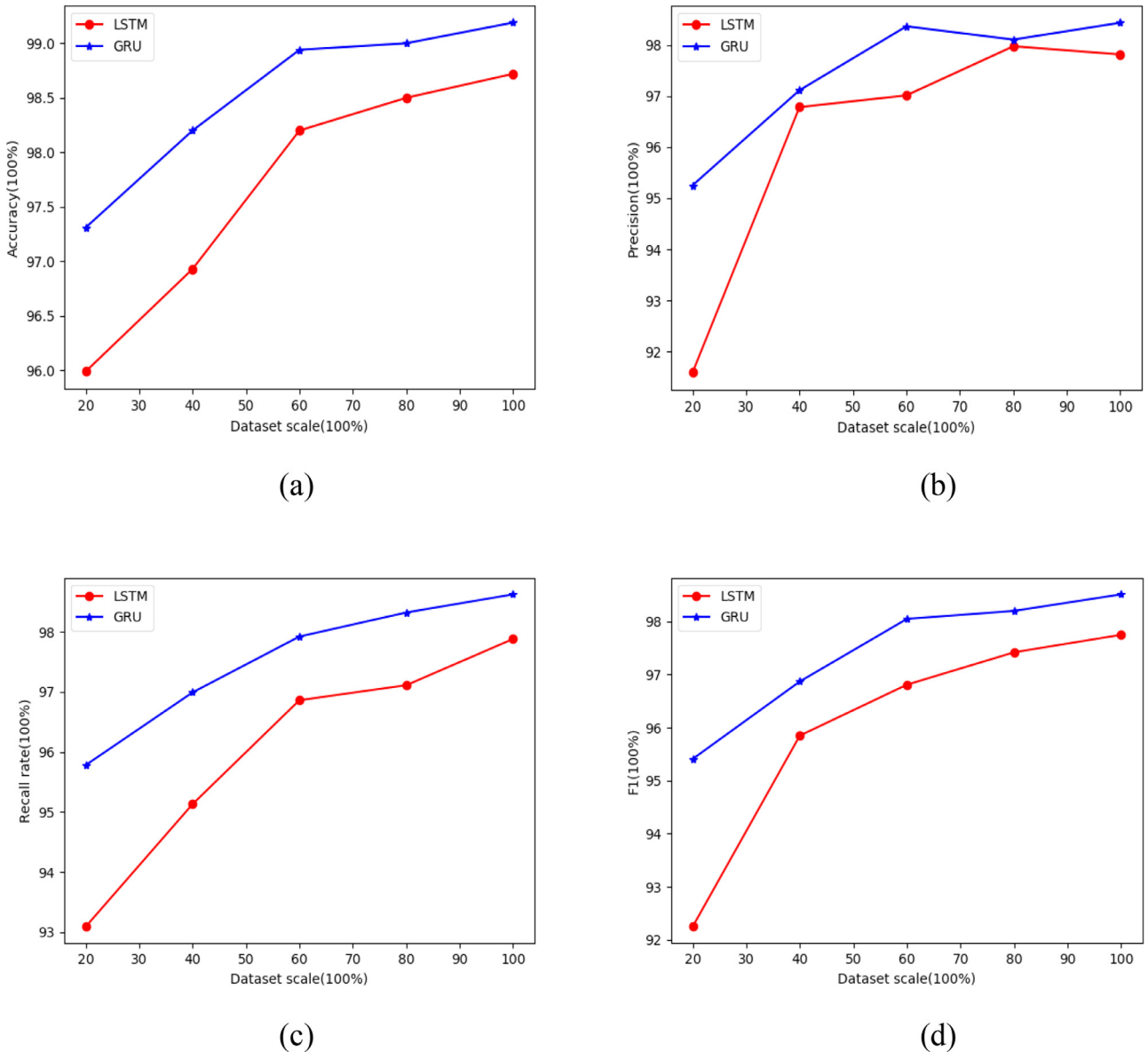


Fig. 5. Vulnerability prediction results of data sets with different proportions.

Table 9
Prediction results of various types of buffer overflow vulnerability.

Vulnerability type	Precision (%)	Recall rate (%)	F1 (%)
Stack-based Buffer Overflow	99.80	99.80	99.80
Heap-based Buffer Overflow	100	99.47	99.73
Buffer Underwrite	97.10	91.80	94.38
Buffer Overread	96.98	99.38	98.17
Buffer Underread	92.15	97.40	94.71
Integer Overflow or Wraparound	100	100	100
Integer Underflow	100	100	100
Free of Pointer – not at Start of Buffer	100	100	100
Not prone to overflow type vulnerability	99.88	99.70	99.79

Table 10
Verification result of software buffer overflow vulnerability.

Software	Version	File	Function	Vulnerability type
Linux kernel	Linux Kernel 3.9-RC7	af_rose.c	rose_recvmsg	Stack-based Buffer Overflow
GNU LibreDWG	0.9.3	bits.c	bit_write_UMC	Stack-based Buffer Overflow
Espruino	Espruino <1.99	jslex.c	jslTokenAsString	Stack-based Buffer Overflow
PDFResurrect	PDFResurrect <0.18	pdf.c	pdf_load_pages_kids	Buffer Underwrite

Code 1
Stack-based Buffer Overflow (Out of bounds array).

```
1  char buf [BUFSIZE];
2  Strcpy (buf, argv [1]);
```

Code 2
Heap-based Buffer Overflow sample code.

```
1  char * buf;
2  buf=(char *) malloc (sizeof (char)* BUFSIZE);
3  Strcpy (buf, argv [1]);
```

Code 3
Buffer Underwrite overflow (*Negative index*).

```
1  Strncpy (destBuf, &srcBuf [find (srcBuf, ch)],1024);
```

Code 4
Buffer Overread sample code.

```
1  int processMessageFromSocket(int
   socket) {
2  int success; char buffer[BUFFER_SIZE];
   char message[MESSAGE_SIZE]; int
   index;
3  if (getMessage(socket, buffer,
   BUFFER_SIZE) > 0) {
4  ExMessage
   *msg=recastBuffer(buffer);
5  for (index=0; index <
   msg->msgLength; index++) {
6  message[index]=msg-
   >msgBody[index];}
7  ...
```

Code 5
Integer Overflow or Wraparound sample code.

```
1  img_t table_ptr; /*struct containing img data, 10kB each*/
2  int num_imgs;
3  ...
4  num_imgs=get_num_imgs();
5  table_ptr=(img_t*)malloc(sizeof(img_t)*num_imgs);
```

Code 6
Integer Underflow sample code.

```
1  int i;
2  i=-2147483648;
3  i=i - 1;
```

Code 7
Free of Pointer - not at Start of Buffer sample code.

```
1  char **ap, *argv[10], *inputstring;
2  for (ap=argv; (*ap=strsep(&inputstring, " \t")) != NULL;)
3    if (**ap != '\0')
4      if (++ap >= &argv[10])
5        break;
6  ...
7  free(ap[4]);
```

Code 8
Function code slice.

```
1  Example function()
2  char * data;
3  data=dataBuffer;
4  memset(data, 'A', 100-1);
5  data[100-1]='\0';
6  char dest[50]=" ";
7  memmove(dest, data, strlen(data)*sizeof(char));
8  dest[50-1]='\0';
9  printLine(data);
```

Code 9
Example function symbol replaces function name.

```
1  Example function()
2  char * data;
3  data=dataBuffer;
4  memset(data, 'A', 100-1);
5  data[100-1]='\0';
6  char dest[50]=" ";
7  memmove(dest, data, strlen(data)*sizeof(char));
8  dest[50-1]='\0';
9  FUN0(data);
```

Code 10
Example function symbol replaces parameter name.

```
1  Example function()
2  char * VAR1;
3  VAR1=VAR2;
4  memset(VAR1, ' VAR3', VAR4);
5  VAR1 [VAR4]='\0';
6  char VAR5 [50]=" ";
7  memmove(VAR5, VAR1, strlen(VAR1)*sizeof(char));
8  VAR5 [50-1]='\0';
9  FUN0(VAR1);
```

Overflow vulnerability (CNVD-2020–41852). The vulnerability stems from a failure to properly validate data boundaries when performing operations on memory, resulting in incorrect read and write operations to other memory locations. Currently, the vendor has released a security bulletin and patch for the *GNU LibreDWG* stack overflow vulnerability to fix this vulnerability.

- In *Espruino* versions before 1.99, using the multiple-type overflow vulnerability prediction method based on combination features and a time series neural network algorithm, the jslTokenAsString function in the src/jslex.c file was predicted to be prone to **Stack-based Buffer Overflow** vulnerability based on bidirectional *LSTM* and bidirectional *GRU*. The China National Vulnerability Database verified that the jslex.c file in *Espruino* versions before 1.99 has **Stack-based Buffer Overflow** vulnerability (CNVD-2018-10890). The vulnerability stems from the incorrect use of the strcpy function in the jslex.c file. Attackers can use a specially crafted file to exploit the vulnerability to cause a denial of service (application crash) and possibly steal information. Currently, the vendor has issued a security bulletin and patch information to fix this vulnerability.
- In *PDFResurrect* versions before 0.18, using the multiple-type overflow vulnerability prediction method based on combination features and a time series neural network algorithm, the pdf_load_pages_kids function in the pdf.c file was predicted to be prone to **Buffer Underwrite** vulnerability based on bidirectional *LSTM* and bidirectional *GRU*. The China National Vulnerability Database verified that the pdf_load_pages_kids function in the pdf.c file in *PDFResurrect* versions before 0.18 has **Buffer Underwrite** vulnerability (CNVD-2020–16832). Attackers can use this vulnerability to cause malloc failure and out-of-bounds writes. Currently, the vendor has issued a security bulletin and patch information to fix this vulnerability.

Experimental results show that the multiple-type overflow vulnerability prediction method based on combination features and a time series neural network algorithm can accurately predict overflow vulnerability in the *Linux kernel* operating system, *GNU LibreDWG*C language library, *Espruino* software, and *PDFResurrect* software, proving the rationality and effectiveness of the method proposed in this paper.

6. Discussion

This paper proposes a multitype overflow vulnerability prediction method based on a combination of features and time series neural network algorithms. This method has three advantages. 1. This paper integrates the internal vulnerability characteristics and external vulnerability characteristics of the function, which can extract the vulnerability characteristics more comprehensively. In this way, not only the code structure characteristics of a single function can be retained, but also the vulnerability characteristics between multiple functions can be retained. 2. This paper can predict multiple types of overflow vulnerabilities simultaneously, which can improve the efficiency of repairing vulnerabilities. 3. The method proposed in this paper has a high precision rate, accuracy rate, recall rate, and F1 value. Moreover, it can effectively detect overflow vulnerabilities in real software. There are some limitations in the design, implementation and evaluation of this method: 1. This method is limited to vulnerability detection through the source code of the program. It cannot detect executable files of programs whose source code cannot be obtained. Designing and detecting vulnerability in executable files is a more challenging problem. 2. This method is currently applicable for C/C++ programs. Programs developed in other programming languages have not be detected. It is still necessary to study whether this method design ideas are applicable to programs developed in other programming languages for vulnerability detection. 3. The current evaluation of this method is limited. We verified the software for which vulnerability information has been made public. Some software with undiscovered 0-day vulnerability has not been detected.

7. Conclusion

In this paper, aiming at common and dangerous buffer overflow vulnerabilities, a method was proposed to detect overflow vulnerability in software. Because the causes of overflow vulnerabilities are diverse, various types of overflow vulnerabilities can be more accurately predicted. This paper investigates eight types of overflow vulnerabilities to analyze the causes of each type of overflow vulnerability in software. This paper analyzed how to select relevant sensitive functions when extracting vulnerability features from various overflow vulnerabilities. A method for extracting internal and external features of functions related to vulnerability was proposed. This method can comprehensively and accurately represent the characteristics related to the overflow vulnerability in the source code. The experimental results showed that this method had high accuracy and effectiveness. The method was also verified by four programs: *Linux kernel*, *GNU LibreDWG*, *Espruino*, and *PDFResurrect*. It is further proven that the method proposed in this paper has achieved good results in practical applications and can accurately predict the corresponding types of overflow vulnerabilities in actual software. In our future work, we will focus on the software developed in different languages and the dynamic execution or binary program of the software so as to explore the different type of faults or 0-day vulnerability.

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.

CRediT authorship contribution statement

Zhangqi Zheng: Conceptualization, Methodology, Software, Formal analysis, Investigation, Writing – original draft. **Bing Zhang:** Validation, Writing – review & editing, Project administration, Funding acquisition. **Yongshan Liu:** Data curation. **Jiadong Ren:** Visualization. **Xuyang Zhao:** Supervision. **Qian Wang:** Resources.

Acknowledgements

This work was supported by the National Natural Science Foundation of China (61802332, 61807028, 61772449, 61972334), and the Natural Science Foundation of Hebei Province P. R. China under Grant No. F2019203120.

References

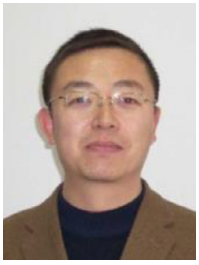
Ahmad, T., Ari, K., Lauri, M., 2011. Recognizing Algorithms Using Language Constructs, Software Metrics and Roles of Variables: An Experiment with Sorting Algorithms. Computer Journal 54 (7), 1049–1066.
Schneidewind, N.F., 1992. Methodology for Validating Software Metrics. IEEE Transactions on Software Engineering 18 (5), 410–422.
Choudhary, G.R., Kumar, S., Kumar, K., Mishra, A., Catal, C., 2018. Empirical analysis of change metrics for software fault prediction. Computers and Electrical Engineering 67, 15–24.
Rahimi, S., Zargham, M., 2013. Vulnerability Scrying Method for Software Vulnerability Discovery Prediction Without a Vulnerability Database. IEEE Transactions on Reliability 62 (2), 395–407.
Niu, W.N., Ding, X.F., Liu, Z., Zhang, X.S., 2013. Vulnerability Finding Using Symbolic Execution on Binary Programs. Computer Science 40 (10), 119–121 138.
Clause, J., Li, W., Orso, A., 2007. Dytan: a generic dynamic taint analysis framework. In: Proceedings of 2007 international symposium on Software testing and analysis, London, UK, pp. 196–206.
Zhang, R., Huang, S., Qi, Z., Guan, H., 2012. Static program analysis assisted dynamic taint tracking for software vulnerability discovery. Computers And Mathematics with Applications 63 (2), 1–36.
Li, H., Kim, T., Erdene, M.B., Lee, H., 2013. Software Vulnerability Detection Using Backward Trace Analysis and Symbolic Execution. In: Proceedings of 2013 International Conference on Availability, Reliability and Security. Regensburg, Germany.
Liu, S., Lin, G., Han, Q.L., Wen. J. Zhang, S., Xiang, Y., 2020. DeepBalance: Deep-Learning and Fuzzy Oversampling for Vulnerability Detection. IEEE Transactions on Fuzzy Systems 28 (7), 1329–1343.
Dam, H.K., Pham, T., Ng, S.W., Tran, T., Grundy, J., Ghose, A., Kim, T., Kim, C.J., 2019. Lessons Learned from Using a Deep Tree-Based Model for Software Defect Prediction in Practice. In: Proceedings of the 16th International Conference on Mining Software Repositories, Montreal, Quebec, Canada, pp. 46–57.
Shippey, T., Bowes, D., Hall, T., 2019. Automatically identifying code features for software defect prediction: Using AST N-grams. Information and software technology 106 (2), 142–160.
Liu, M.Y., Traore, I., 2006. Empirical relation between coupling and attackability in software systems: a case study on dos. In: Proceedings of 2006 Programming Languages and Analysis for Security Workshop, Ottawa, Ontario, Canada, pp. 57–64.
Alves, H., Fonseca, B., Antunes, N., 2016. Software metrics and security vulnerability: Dataset and exploratory study. In: Proceedings of the 12th European Dependable Computing Conference, Gothenburg, Sweden, pp. 37–44.
Russell, R.L., Kim, L., Hamilton, L.H., Lazovich, T., Harer, J.A., Ozdemir, O., Ellingwood, P.M., Mcconly, M.W., 2018. Automated Vulnerability Detection in Source Code Using Deep Representation Learning. In: Proceedings of the 17th IEEE International Conference on Machine Learning and Applications. Orlando, Florida, USA.
Dam, H.K., Tran, T., Pham, T.T.M., Ng, S.W., Ghose, A., 2018. Automatic feature learning for predicting vulnerable software components. IEEE Transactions on Software Engineering 47 (1), 67–85.
Li, Z., Zou, D., Xu, S., Ou, X., Jin, H., Wang, S., Deng, Z., Zhong, Y., 2018. VulDeePecker: A deep learning-based system for vulnerability detection. In: Proceedings of 2018 Network and Distributed System Security Symposium. San Diego, CA, USA.
Gui, B., Liang, X., Zhao, B., Feng, Z., Wang, J., 2014. Detecting Integer Overflow Vulnerabilities in Binary Executables Based on Target Filtering and Dynamic Taint Tracing. Chinese Journal of Electronics 23 (2), 348–353.
Yang, Z., Sun, X., Yi, D., Liang, C., Zeng, S., Yu, F., Feng, D., 2015. Improving Accuracy of Static Integer Overflow Detection in Binary. In: Proceedings of the 18h International Symposium on Research in Attacks, Kyoto, Japan, pp. 246–269.
Mouzarani, M., B, Sadeghiyan, Zolfaghari, M., 2016. Smart fuzzing method for detecting stack-based buffer overflow in binary codes. IET Software 10 (4), 96–107.
Zhang, B., Feng, C., Wu, B., Tang, C., 2016. Detecting integer overflow in Windows binary executables based on symbolic execution. In: Proceedings of the 17th IEEE/ACIS International Conference on Software Engineering, Shanghai, China, pp. 385–390.
Hao, S., Zhang, X.Y., Zheng, Y.H., Zeng, Q.K., 2016. IntEQ: Recognizing Benign Integer Overflows via Equivalence Checking across Multiple Precisions. In: Proceedings of the 38th International Conference, Austin, TX, USA, pp. 1051–1062.
Ding, S., Yuan, J, 2011. Identifying buffer overflow vulnerabilities based on binary code. In: Proceedings of 2011 IEEE International Conference on Computer Science and Automation Engineering, Shanghai, China, pp. 738–742.

Padmanabhuni, B.M., Tan, H.B.K., 2014. Auditing Buffer Overflow Vulnerabilities Using Hybrid Static-Dynamic Analysis. In: Proceedings of the 38th IEEE Annual International Computers Software and Applications Conference, Vasteras, Sweden, pp. 394–399.
Zhang, H., 2014. Research on memory allocation of C++ language. Computer Era 000 (5), 44–46.
Ganapathy, V., Seshia, S.A., Jha, S., Reps, T.W., Bryant, R.E., 2005. Automatic Discovery of API-Level Exploits. In: Proceedings of the 27th International Conference on Software Engineering. St. Louis, MO, USA.
Peng, L., Zhu, Q., Lv, A.X., Wang, L., 2020. Effective long short-term memory with fruit fly optimization algorithm for time series forecasting. SOFT COMPUTING 24 (19), 15059–15079.
Gers, F.A., Schmidhuber, E., 2001. LSTM recurrent networks learn simple context-free and context-sensitive languages. IEEE Transactions on Neural Networks 12 (6), 1333–1340.
Chung, J., Gulcehre, C., Cho, K., Bengio, Y., 2014. Empirical Evaluation of Gated Recurrent Neural Networks on Sequence Modeling. Eprint Arxiv.
Han, P., Wang, W.Q., Shi, Q.Y., Yue, J.C., 2021. A combined online-learning model with K-means clustering and GRU neural networks for trajectory prediction. Ad Hoc Networks 117 (44), 102476.
Bengio, Y., Ducharme, R., Vincent, P., Jancin, C., 2003. A Neural Probabilistic Language Model. Journal of Machine Learning Research 3, 1137–1155.
Mikolov, T., Chen, K., Corrado, C., Dean, J., 2013. Efficient Estimation of Word Representations in Vector Space. In: Proceedings of 2013 ICLR Workshop. Scottsdale, AZ, USA.
C. Guo, and F. Berkhahn, Entity Embeddings of Categorical Variables, 2016, arXiv: 1604.06737.
Grbovic, M., Cheng, H., 2018. Real-time Personalization using Embeddings for Search Ranking at Airbnb. In: Proceedings of the 24th ACM SIGKDD International Conference, London, UK, pp. 311–320.
Gal, Y., Ghahramani, Z., 2015. A Theoretically Grounded Application of Dropout in Recurrent Neural Networks. In: Proceedings of the 30th Conference on Neural Information Processing Systems. Barcelona, Spain.
Glorot, X., Bengio, Y., 2010. Understanding the difficulty of training deep feedforward neural networks. In: Proceedings of the 13th International Conference on Artificial Intelligence and Statistics. Chia Laguna Resort, Sardinia, Italy.

Zhangqi Zheng received the M.S. degree in computer technology from Yanshan University in 2019, and he is currently a Ph.D. Student with the computer science and technology, Yanshan University. Her research interests include data mining, machine learning, and software security.

Bing Zhang received the B.S. degree from the College of Computer and Information Technology, Three Gorges University, China, in 2012, and the Ph.D. degree from the School of Information Science and Engineering, Yanshan University, China, in 2018. He is currently an associate professor at the School of Information Science and Engineering, Yanshan University. His research interests include data mining, machine learning, and software security.

Yongshan Liu received the M.S. degree in computer science and technology from Yanshan University in 1989 and the Ph.D. degree in computer and applications from Harbin University of Science and Technology in 2006. From 1994 to now, he is a Professor at the School of information science and engineering, Yanshan University. His research interests include data mining and machine learning.



Jiadong Ren received the B.S. and M.S. degrees from the Northeast Heavy Machinery Institute in 1989 and 1994, respectively, and the Ph.D. degree from the Harbin Institute of Technology in 1999. He is a professor in the school of Information Science and Engineering, Yanshan University, China. His research interests include data mining, temporal data modeling and software security.

Xuyang Zhao received B.S degree from the school of information science and engineering, Yanshan University, China, in 2020. He is currently pursuing his M.S degree in the school of information science and engineering, Yanshan University, China. His research interests include complex network, data mining, and software security.

Qian Wang received the B.S., M.S., and Ph.D. degree from the School of Information Science and Engineering, Yanshan University, China, in 2009, 2012, and 2016, respectively. She has been a Visiting Scholar with the University of Hull from 2015 to 2016. She is currently an associate professor with the School of Information Science and Engineering, Yanshan University, China. Her research interests include data mining, complex network, and software security.