

Received 21 May 2023, accepted 19 June 2023, date of publication 23 June 2023, date of current version 28 June 2023.

Digital Object Identifier 10.1109/ACCESS.2023.3289001

RESEARCH ARTICLE

Binary Code Vulnerability Detection Based on Multi-Level Feature Fusion

GUANGLI WU^{ID}, (Member, IEEE), AND HUILI TANG^{ID}

School of Cyberspace Security, Gansu University of Political Science and Law, Lanzhou 730070, China

Corresponding author: Guangli Wu (272956638@qq.com)

This work was supported in part by the Natural Science Foundation of Gansu Province under Grant 21JR7RA570; in part by the Gansu University of Political Science and Law Major Scientific Research and Innovation Project under Grant GZF2020XZDA03; in part by the Young Doctoral Fund Project of Higher Education Institutions in Gansu Province under Grant 2022QB-123 in 2022; in part by the Gansu Provincial University Innovation Fund Project 2022A-097; in part by the Gansu Province Excellent Graduate Student Innovation Star Project, in 2022, under Grant 2022CXZX-790; and in part by the University-Level Innovative Research Team of Gansu University of Political Science and Law.

ABSTRACT The existence of software vulnerabilities will cause serious network attacks and information leakage problems. Timely and accurate detection of vulnerabilities in software has become a research focus on the security field. Most existing work only considers instruction-level features, which to some extent overlooks certain syntax and semantic information in the assembly code segments, affecting the accuracy of the detection model. In this paper, we propose a binary code vulnerability detection model based on multi-level feature fusion. The model considers both word-level features and instruction-level features. In order to solve the problem that traditional text embedding methods cannot handle polysemy, this paper uses the Embeddings from Language Models (ELMo) model to obtain dynamic word vectors containing word semantics and other information. Considering the grammatical structure in the assembly code segment, the model randomly embeds the normalized assembly code segment to represent it. Then the model uses bidirectional Gated Recurrent Unit (GRU) to extract word-level sequence features and instruction-level sequence features respectively. Then, the weighted feature fusion method is used to study the impact of different sequence features on the model performance. During model training, adding standard deviation regularization to constrain model parameters can prevent the occurrence of overfitting problems. To evaluate our proposed method, we conduct experiments on two datasets. Our method achieves an F1-score of 98.9 percent on the Juliet Test Suite dataset and a F1-score of 87.7 percent on the NDSS18 (Whole) dataset. The experimental results show that the model can improve the accuracy of binary code vulnerability detection.

INDEX TERMS Binary code vulnerability detection, embeddings from language models, feature fusion, instruction level sequence features, word level sequence features.

I. INTRODUCTION

In today's highly developed technology, the widespread use of software has greatly facilitated people's lives, but at the same time, it has brought many security problems. Software vulnerability has always been a topic that can not be ignored because once criminals maliciously invade the system through a software vulnerability, it may cause

immeasurable losses. Therefore, in the process of software development and maintenance, software security vulnerabilities are highly valued by developers. It is significant for individuals, enterprises, and even the whole society to detect software vulnerabilities timely and accurately.

Software vulnerability detection [1], [2], [3], [4], [5] is to analyze the input code in the software to detect whether there are exploitable defects. Generally, it is very difficult for researchers to obtain the software's source code, but they can get the corresponding binary files. Therefore, the study of

The associate editor coordinating the review of this manuscript and approving it for publication was Sedat Akleylek^{ID}.

binary code vulnerability detection [6] has crucial practical significance. At present, static analysis [7], [8], [9], dynamic analysis [10], [11], [12], and a combination of dynamic and static analysis can be used for binary code vulnerability detection.

The static analysis method refers to the method of analyzing the program to detect whether there are vulnerabilities in the program without running the program. This method often converts the program to be tested into an intermediate language, analyzes the typical features contained in the intermediate language, and then uses specific techniques to detect vulnerabilities.

The dynamic analysis method [13] needs to run the target program and observe and analyze the execution status of the program so as to achieve the purpose of detecting vulnerabilities. For example, environment injection error is to inject error information into its execution environment without modifying the program to be tested and then observing the running status of the program. If the program's running status is abnormal, it means that the artificial injection error triggered the potential defects in the program so as to achieve the purpose of detecting vulnerabilities.

The dynamic and static analysis method [14] combines the accuracy of dynamic vulnerability analysis technology and the path completeness of static vulnerability analysis technology to detect vulnerabilities.

Since the static analysis method studies all the running tracks of the target program, it has a great path coverage. However, the dynamic analysis method and the combination of dynamic and static analysis methods still have the problem of low path coverage. Therefore, the research on static methods in binary code vulnerability detection is also extensive.

Static binary code vulnerability detection can be divided into traditional detection methods and detection methods based on deep learning. Traditional static binary code vulnerability detection methods usually convert binary code into intermediate language, and then use some static methods to analyze the program to detect potential vulnerabilities. For example, pattern matching method detect vulnerabilities according to vulnerability patterns defined by human experts in advance. Vulnerability patterns are the analysis of a large amount of code, abstracting and summarizing typical features that may be regular expressions, string matching, code structure, etc. of specific vulnerability types. By analyzing intermediate languages, these typical features can better distinguish whether the code contains defects. However, the method of pattern matching relies too much on the features abstractly selected, and can only detect such binary code with typical features. Traditional static binary vulnerability detection methods have the advantages of detecting defects in the early stages of program development, reducing software development costs and time. However, when using them for vulnerability detection, there may be false positives, false negatives, and other situations, and it may also consume a lot of manpower and computing resources.

Due to its excellent ability to automatically extract features, deep learning has been validated as effective in code vulnerability detection [15]. Li et al. [16] were the first to apply deep learning in software vulnerability detection tasks. The designed VulDeePecker (Vulnerability Deep Pecker) system provides a new research perspective for vulnerability detection and this paper also contributes the first vulnerability dataset suitable for deep learning. Laura et al. [17] proposed the Vulnerability detection using natural code bases (VUDENC) method based on natural code repositories. VUDENC utilizes Word2Vec to represent word vectors, and LSTM networks classify sequences of vulnerable code tokens. The work of [16] and [17] are both excellent approaches for applying deep learning to source code vulnerability detection.

Applying deep learning technology to the field of binary code vulnerability detection can be divided into code scanning and similarity detection. Binary code similarity detection is to calculate the similarity between the code to be tested and the code with identified vulnerabilities, and finally determine whether the code to be tested has vulnerabilities. Wang et al. [18] proposed the jump-aware Transformer for binary code similarity detection (jTrans), which is the first solution to embed control-flow information into Transformer. jTrans combines the natural language processing (NLP) model that captures instruction semantics with the cfg that captures control information to infer the similarity representation of binary code, and finally realizes the fusion of control flow information into Transformer. At the same time, many scholars have applied the binary code similarity method for vulnerability detection. However, this method has certain disadvantages, it cannot detect unknown types of vulnerabilities, so it is extremely important to directly detect binary codes using code scanning methods.

Code scanning refers to the process of traversing and slicing the assembly code segments obtained from binary code conversion to determine whether the slices contain vulnerabilities. Existing code scanning techniques often utilize methods from natural language processing for vulnerability detection. The specific steps include data preprocessing, code embedding, and feature extraction. The code embedding network typically aims to vectorize the assembly code segments, but there are still some challenges in transforming the code into recognizable vectors by neural networks while preserving as much syntax and semantic information as possible. The Instruction2vec method proposed by Lee et al. [19] represents each word in the assembly code segments using Word2Vec word vectors. It takes into consideration the composition structure of instructions and represents each line of instructions using nine values: one for the opcode and four for each of the two operands. If there are fewer than two operands, padding is applied to fill the remaining values. Based on this, Yan et al. [20] proposed the Hierarchical Attention Network for Binary Code Vulnerability Detection (HAN-BSVD), which expands on the structural composition

of instructions mentioned in the [19]. They introduced an additional field for operand type in the instruction structure. Unlike the aforementioned instruction structure, in the method proposed by Le et al. [21], each instruction is divided into opcode and instruction information, which are embedded separately and then concatenated to form the embedded representation of the instruction. In addition, in the BVDetector method proposed by Tian et al. [22], Word2Vec is directly used to generate word vectors. However, this method can only detect vulnerabilities caused by library/API functions, and it has certain limitations. In most studies using code scanning techniques for binary code vulnerability detection, code embedding networks often utilize the traditional text embedding method Word2Vec to represent word vectors. However, Word2Vec can only learn based on a relatively large window, and the obtained word vector is static, with identical vectors for the same word regardless of its position, limiting the capture of contextual relationships. To better extract syntactic information from assembly code segments and obtain different vector representations of the same word in various contexts, we use the ELMo model to generate dynamic vector representations of words.

In the feature extraction module, the work of [19] uses TextCNN to automatically extract features from the concatenated vector representations of each line of instructions. On the other hand, the work of [20] utilizes bidirectional GRU and word attention modules to obtain the embedding representation of instructions. These embeddings are subsequently processed by TextCNN in conjunction with a spatial attention mechanism to automatically extract features. The VUDENC [17] method selects Long Short-Term Memory (LSTM) networks to classify vulnerable code sequences at a fine-grained level. Narayana et al. [23] used artificial neural network, autoencoder, etc. to automatically extract features of vulnerable codes. Ouyang et al. [24] applied Word2Vec and Long Short-Term Memory (LSTM) network to the research of binary code vulnerability detection.

The above methods have been verified to have good performance in the field of vulnerability detection. However, these methods only deal with instruction-level sequence features, and to a certain extent ignore part of the syntax and semantic information in the assembly code segment. To better learn the contextual relations between words in assembly code segments, in this paper we consider word-level sequence features. Finally, the instruction-level features and word-level sequence features in the assembly code segment are fused, and a binary code vulnerability detection model based on multi-level feature fusion is proposed.

The contribution of the model proposed in this paper is:

- In order to better identify vulnerable flaw features of the assembly code segment, a bidirectional Gated Recurrent Unit (GRU) is used to extract word-level sequence features and instruction-level sequence features. We compare the effects of feature concatenate, feature addition, and weighted feature fusion on the model performance and finally prove that weighted

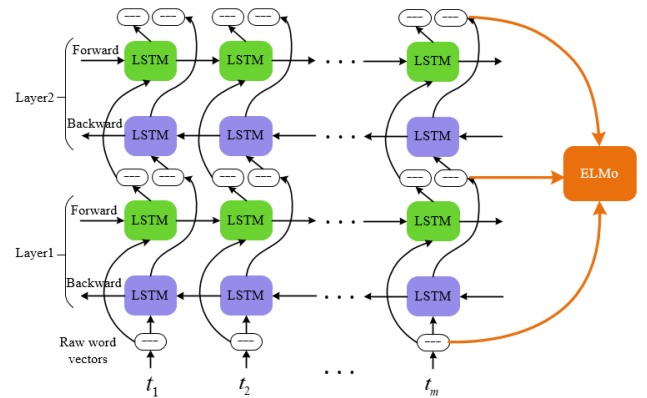


FIGURE 1. Internal structure diagram of ELMo.

feature fusion exists as the most suitable method for vulnerability detection.

- In this paper, the ELMo pre-training model is used to obtain dynamic vectors according to the context semantic information of a word, which can solve the polysemy problem of a word well, and the obtained features are more abundant, which can better represent the assembly code segment.

II. RELATED WORK

A. ELMo

ELMo [25] is a pre-trained language model. It is proposed to solve the problem of polysemy that cannot be handled by traditional language models. Through it, we can capture the deep context word information in the sentence and get the embedded representation that conforms to the current context. Even in the same sentence, the same word also has a different embedding vector representation. The ELMo model structure is shown in Fig. 1.

ELMo model includes an input layer, bidirectional language model, and output layer. The bidirectional language model is the focus of the ELMo model, which is composed of multi-layer bidirectional LSTM. Through it, the deep context semantic features of the original input can be obtained.

For the current input (t_1, t_2, \dots, t_m) of the model, ELMo's forward language model models it, and uses the above information $(t_1, t_2, \dots, t_{k-1})$ of the current word t_k in the input sequence to predict the current word. The calculation is shown in (1).

$$p(t_1, t_2, \dots, t_m) = \prod_{k=1}^m p(t_k | t_1, t_2, \dots, t_{k-1}) \quad (1)$$

ELMo's backward language model models the input and uses the following information $(t_{k+1}, t_{k+2}, \dots, t_m)$ of the current word t_k in the input sequence to predict the current word. The calculation is shown in (2).

$$p(t_1, t_2, \dots, t_m) = \prod_{k=1}^m p(t_k | t_{k+1}, t_{k+2}, \dots, t_m) \quad (2)$$

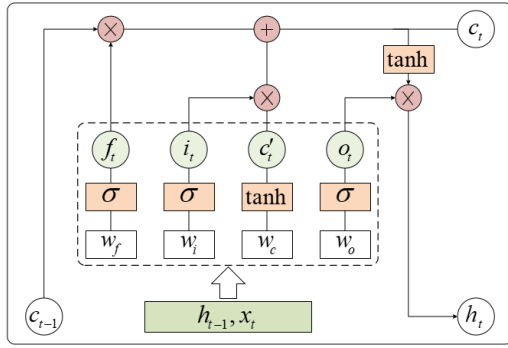


FIGURE 2. The structure diagram of LSTM.

For each word t_k , $2 * L + 1$ vectors will be obtained through the bidirectional language model of the L layer, including the forward and backward vectors of each layer and the original input vector. The final output vector is shown in (3).

$$R_k = \left\{ x_k^{LM}, \overset{\rightarrow LM}{h_{k,j}}, \overset{\leftarrow LM}{h_{k,j}} \mid j = 1, \dots, L \right\} = \left\{ h_{k,j}^{LM} \mid j = 0, \dots, L \right\} \quad (3)$$

In (3), $\overset{\rightarrow LM}{h_{k,j}}$ and $\overset{\leftarrow LM}{h_{k,j}}$ are forward and backward language model outputs, respectively. LSTM networks are often used to process data with time series characteristics. In the bidirectional language model, LSTM extracts the context semantic information of the current word according to the sequence. As shown in the Fig.2, LSTM network [26], [27] is composed of an input gate, forgetting gate, and output gate.

In Fig.2, x_t is the input at the current time t , h_t is the output at the current time, c_t is the cell state, c'_t is the new data at the current time, and σ is the activation function. f_t , i_t and o_t are respectively the forgetting gate, input gate and output gate at the current moment. w_f , w_i and w_o are respectively the weight matrix corresponding to the forgetting gate, input gate, and output gate.

B. GRU

GRU [28] is a variant of LSTM, which can process sequence data and solve the long-dependence problem in a traditional recurrent neural network. Compared with the three “door” structures contained in the LSTM shown in Fig.2, the GRU structure is simpler, including only update and reset gates.

As shown in Fig.3, the GRU network consists of update gate z_t and reset gate γ_t . The current input of GRU at t time is the hidden layer state h_{t-1} at $t - 1$ time and the input x_t at the current time.

The update door determines how much of the status at $t - 1$ time is brought into the current state. The calculation equation is:

$$z_t = \sigma(W_z \cdot [h_{t-1}, x_t]) \quad (4)$$

$$h_t = (1 - z_t) * h_{t-1} + z_t * \tilde{h}_t \quad (5)$$

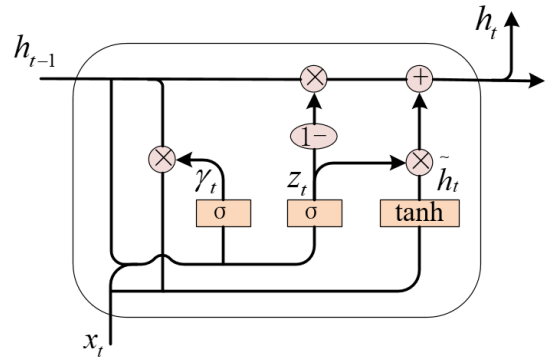


FIGURE 3. The structure diagram of GRU.

The reset gate determines how many states at $t - 1$ time are written into the current candidate set \tilde{h}_t . The calculation equation is:

$$\gamma_t = \sigma(W_\gamma \cdot [h_{t-1}, x_t]) \quad (6)$$

$$\tilde{h}_t = \tanh(W_h \cdot [\gamma_t * h_{t-1}, x_t]) \quad (7)$$

The output calculation equation of GRU is:

$$y_t = \sigma(W_O \cdot h_t) \quad (8)$$

where $[\cdot]$ represents the connection of two vectors, $*$ represents the product of matrix, $\sigma(\cdot)$ represents the sigmoid activation function.

III. THE MODEL

For binary code vulnerability detection tasks, this paper proposes a vulnerability detection model based on multi-level feature fusion. The model mainly includes three parts: feature extraction network, feature fusion module, and classifier. The feature extraction module extracts word-level and instruction-level sequence features, respectively. The model uses ELMo pre-training model to obtain the dynamic vector representation of words, and bidirectional GRU obtains the sequence characteristics between words. In the instruction-level feature extraction module, this paper first considers the syntax and semantic structure of the instruction, obtains the embedding matrix of the assembly code segment, uses bidirectional GRU to extract context information of the instruction, and obtains instruction-level sequence features. In the feature fusion module, the weighted feature fusion mechanism is used to complete the fusion between word-level and instruction-level features. Finally, the model uses a classifier to get the classification results, that is, to determine whether the assembly code segment contains vulnerabilities. The overall framework of the model is shown in Fig.4.

A. WORD-LEVEL FEATURE EXTRACTION MODULE

In this section, the ELMo pre-training language model is used to obtain the embedded representation of each word in the assembly code segment, and the bidirectional GRU obtains the long-term context dependency of the word.

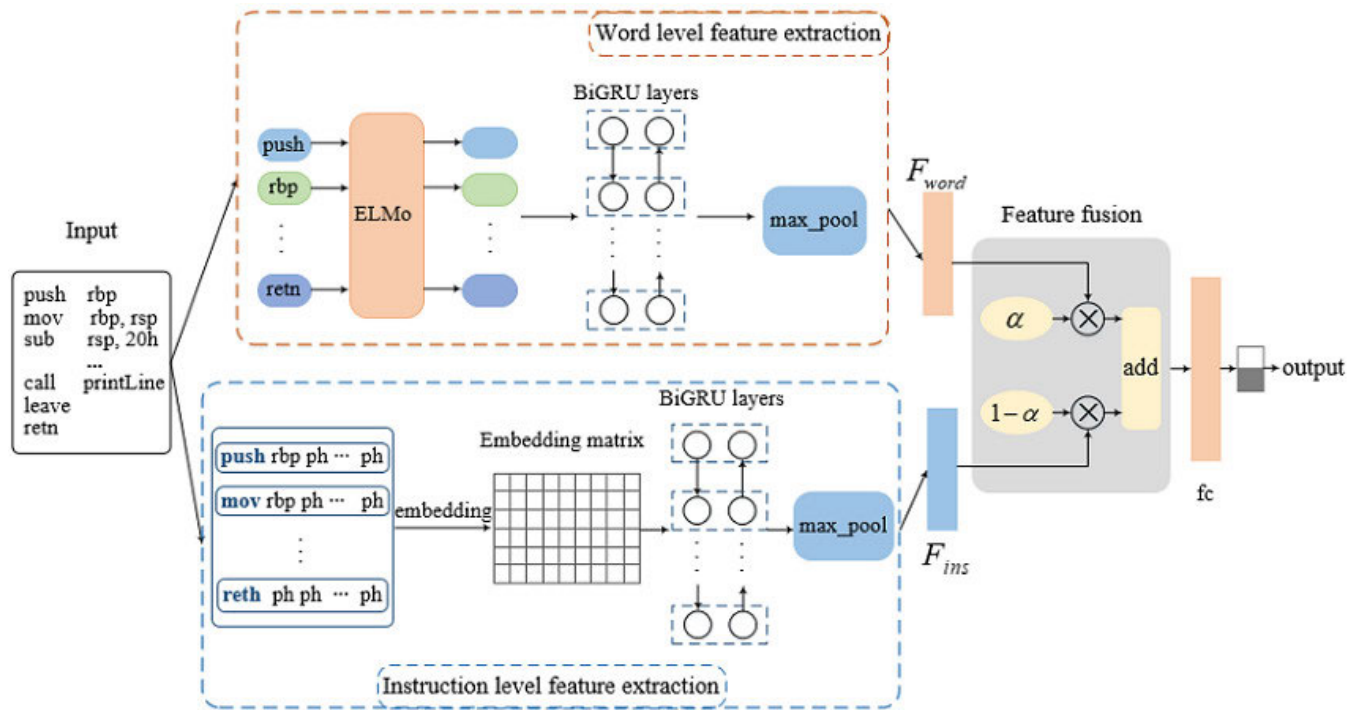


FIGURE 4. Overall architecture of the model.

When using deep learning for vulnerability detection, identifying potentially vulnerable code needs to focus on the learning of context semantic information in the code. It can improve the model detection performance by correctly and reasonably expressing the code meaning. Because Word2Vec [29] and other embedding methods are based on a large corpus and relatively long context to obtain the word vector, for the same word, the vector obtained through Word2Vec is fixed without considering the context and other factors of the word. Based on this, this paper uses ELMo to obtain the dynamic word vector. The specific steps are as follows: Firstly, a pre-trained language model is generated to learn the embedding of the word. When used, the word already has specific context information. Secondly, the word vector is adjusted according to the context information of the word in the current task so that the word in different positions has different vector representations.

For the word-level feature extraction module, the input of the model is $T = (t_1, t_2, \dots, t_m)^T$, where $t_i, i = 1, \dots, m$ represents the words in the assembly code segment, and m represents the number of words in the input assembly code segment. Each word is represented by a vector through the input layer of the ELMo module. Here, the input layer of the ELMo model uses the random embedding representation method to obtain the embedding matrix $E = (E_1, E_2, \dots, E_m)^T$. Then this paper uses a two-level bidirectional language model to model the grammar, semantics and other characteristics of words. Take the vector E_i of a word in the embedded matrix E as the input, get the hidden state $h_{i,1}$

through the forward LSTM of the first layer, and get the hidden state $h_{i,2}$ by inputting it into the forward LSTM of the second layer, so $h_{i,j}, j = 1, 2$ is output as E_i through the forward language model. Similarly, $h_{i,j}, j = 1, 2$ is output as E_i through backward language model. So the output of word E_i is $R_i = \{E_i, h_{i,j}, h_{i,j} \mid j = 1, 2\} = \{h_{i,j}^{LM} \mid j = 0, 1, 2\}$. The output layer of the ELMo model takes into account the output of the last layer of LSTM, the original input vector, and the intermediate word vector. The calculation equation is as follows:

$$ELMo_i = \gamma \cdot (\lambda_0 \cdot h_{i,0}^{LM} + \lambda_1 \cdot h_{i,1}^{LM} + \lambda_2 \cdot h_{i,2}^{LM}) \quad (9)$$

where γ represents the scaling coefficient of all word vectors obtained by ELMo when they are finally used, in this experiment, the initial value is 1. $\lambda_0, \lambda_1, \lambda_2$ is the weight coefficient after softmax processing, and the initial value is 0. They are learnable parameters, and the specific values are obtained by model training.

The input dimension in the word-level module is $(Batch_size, m)$. The model input is embedded into the representation through the input layer of the ELMo module, and the dimension is $(Batch_size, m, embed_dim)$. Here m represents the number of words in the assembly code segment, $embed_dim$ indicates the dimension of the word embedding, that is, the dimension of E_i . Then through the bidirectional language model and the output

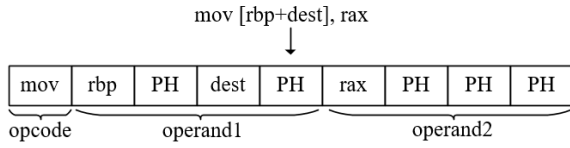


FIGURE 5. Normalization result of one line instruction.

layer, the dimension of the embedded matrix is $(Batch_size, m, hidden_size * 2)$.

After learning each word vector in the assembly code, the model uses GRU to extract the word-level sequence features. In order to better consider the sequence impact, this paper uses bidirectional GRU because the output of the current t-time is not only related to the previous state but also related to the future state. The input of bidirectional GRU is $EL = (ELM_{o1}, ELM_{o2}, \dots, ELM_{om})^T$, at the current time t , the calculation equation of forward GRU is as follows:

$$\vec{h}_t = \vec{f}(ELM_{ot}) \quad (10)$$

The calculation equation of backward GRU is as follows:

$$\overleftarrow{h}_t = \overleftarrow{f}(ELM_{ot}) \quad (11)$$

Then the two results are merged to get the output $h_t = \vec{h}_t + \overleftarrow{h}_t + b$ of bidirectional GRU at the current time t . Max pooling is used in the final part of the word-level feature extraction module, which aims to make the model pay more attention to important features so as to improve model performance.

B. INSTRUCTION-LEVEL FEATURE EXTRACTION MODULE

Considering that the context instructions in the assembly code are also related, this module uses bidirectional GRU to extract the context information of the input vector.

The input of the instruction-level feature extraction module is the assembly code segment, but each line instruction has a different length, so it needs to standardize each line instruction. When standardizing instructions, first, considering the syntax structure of assembly instructions and referring to the method of processing instructions in Instruction2Vec, the instructions are expressed in the form of one opcode and two operands, where four values represent each operand. Second, analyze the type of each operand in the instruction, and place the operand in a fixed position according to the type. If the value of the operand is not enough for a fixed length, it is filled with the invalid operand 'PH'. The normalization result of an instruction line is shown in Fig.5.

The standardized result of the original assembly code segment is shown in Fig.6.

In this paper, the random embedding method is used to obtain the embedding matrix $X = [x_1, x_2, \dots, x_n]^T$ of the assembly code segment with the dimension of $(Batch_size, n, 117)$. x_i represents the i -th instruction in the assembly code segment, $i = 1, \dots, n$. Then the model inputs each

vector in the embedded matrix into the bidirectional GRU in the form of a time step and finally obtains the instruction level sequence feature. The result of bidirectional GRU is also processed by maximum pooling, and the final output dimension is $(Batch, hidden_gru_size * 2)$.

After extracting the features of the input assembly code segment, this paper's focus is to consider the feature of word-level sequence and instruction-level sequence. When using deep learning to detect vulnerabilities, we often take a line of instructions as a whole and use a neural network to obtain instruction-level features, or Wei et al. [30] only consider word-level features. Therefore, this paper combines word-level and instruction-level sequence features and fuses the two features according to a certain method, fully considering their impact on vulnerability detection. Deep learning has many feature fusion methods, including feature concatenate, feature addition, and weighted feature fusion. The weighted feature fusion method is used in this paper, and the calculation equation is as follows:

$$F = \alpha \cdot F_{word} + (1 - \alpha) \cdot F_{ins} \quad (12)$$

$$F_{word} = \text{Maxpool}_{word}(\text{BiGRU}_{word}(f_{LM}(T))) \quad (13)$$

$$F_{ins} = \text{Maxpool}_{ins}(\text{BiGRU}_{ins}(X)) \quad (14)$$

where F_{word} represents the word level sequence feature obtained in section I, and F_{ins} represents the instruction level sequence feature obtained in section II. The value of weighted feature fusion parameter α is given in section IV.

C. ALGORITHM STEPS

The vulnerability detection model proposed in this paper begins by processing the input assembly code segments. It uses an embedding network to obtain a matrix representation of the input, and through a feature extraction network, it obtains word-level features and instruction-level features. Then, a feature fusion module is employed to obtain blended features. Finally, the classification results are obtained based on these features.

The specific algorithm steps are as follows:

Input: Raw input assembly code segments and labels $D = \{x^{(n)}, y^{(n)}\}_{n=1}^N$

step1. Use the ELMo model to get the dynamic vector representation of words in different contexts;

step2. The word-level features are obtained through the bidirectional GRU network and max pool, as shown in (13);

step3. Use random embedding to obtain the vector representation of the words in the instruction, and obtain the instruction-level features through the bidirectional GRU network and max pool, as shown in (14);

step4. Fusion of word-level sequence features and instruction-level sequence features, as shown in (12);

step5. Input the final feature into the fully connected layer with Softmax, and update the weight parameters according to (17);

Output: The results of the vulnerability detection model in this paper.

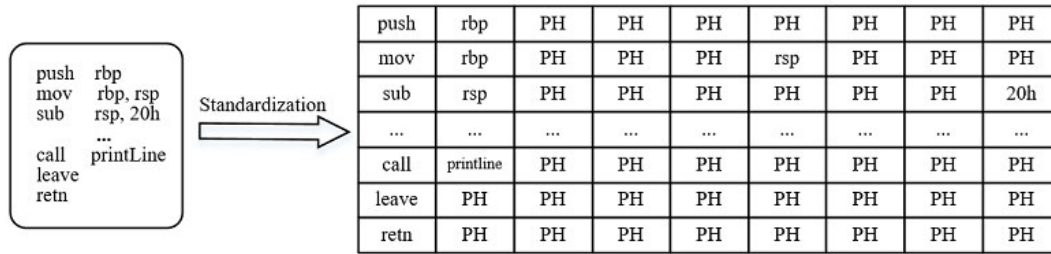


FIGURE 6. Assembly Code Segment Normalization Result.

D. LOSS CONSOLIDATION

In this paper, a regularization technique [31] based on the standard deviation is used. The motivation for adding this new regularization is to control each value of the weight to solve the over-fitting problem in the model, which can improve the detection performance of the model. The standard deviation regularization equation is:

$$\lambda \sum_{i=1}^k \sigma(w_i) \quad (15)$$

where, λ is the regularization coefficient, $\sigma(\cdot)$ is the standard deviation, and the calculation equation is:

$$\sigma(w) = \sqrt{\frac{1}{nk} \left\{ \sum_{i=1}^{nk} w_i^2 - \frac{1}{nk} \left(\sum_{i=1}^{nk} w_i \right)^2 \right\}} \quad (16)$$

In (16), n depends on the number of features in the data set, which represents the number of columns of a specific weight matrix. Similarly, k is the number of rows in the weight matrix.

In this task, the input is $D = \{x^{(n)}, y^{(n)}\}_{n=1}^N$, and the loss function of the model is finally expressed as:

$$J(\theta) = \frac{1}{N} \sum_{n=1}^N L(y^{(n)}, f(x^{(n)}; \theta)) + \lambda \sum_{i=1}^k \sigma(w_i) \quad (17)$$

The ultimate goal of the task is to minimize the loss between the real value and the predicted value of the model and reach the optimal solution. Where $L(\cdot)$ is the cross entropy loss function and $f(x^{(n)}; \theta)$ is the output result of the model. The model in this paper is tested under the framework of Pytorch. The Adam optimizer is used to learn and update the parameters of the neural network, and the learning rate value is set to 0.0005.

IV. EXPERIMENT

The previous sections have introduced this paper's relevant work and model. This section will focus on the experimental process, including datasets, evaluation indicators, and experimental results and analysis.

A. DATASET

This experiment was mainly conducted on the Juliet Test Suite dataset mentioned in [20] and NDSS18 dataset mentioned in [21].

1) JULIET TEST SUITE

The dataset is compiled and disassembled from source code, which is from Juliet Test Suite v1.3 for C/C++ test cases. The work of [20] selected CWE121 type for research, which is a stack-based buffer overflow, resulting in 6506 files, of which 3244 contain defects and 3262 have no defects.

Data preprocessing specifically includes the following steps:

Compilation: The compilation process converts source code into binary code. In this paper, makefile included in the Juliet Test Suite v1.3 for C/C++ is used for compilation. Makefile typically includes rules, each of which defines a set of targets, dependencies, and a set of commands that describe how to generate target files, including the specific steps for generating the target files. Makefile included in the dataset of this paper typically defines the compiler and linker instructions for this type of vulnerability, as well as other necessary variables and macro definitions, to ensure that test cases are compiled and built correctly. For a single test case of CWE121 type, its source code contains two types of functions: one with vulnerabilities and one without vulnerabilities. If it is directly compiled into a binary file using the makefile, only one file containing both types will be generated. Therefore, we implement binary classification by modifying the CFLAGS parameter in the makefile. Specifically, in order to obtain vulnerable samples, a macro OMIT-GOOD is appended after CFLAGS. This instructs the compiler to ignore some good test cases and only compile and run test cases that contain potential vulnerabilities. In order to obtain normal, non-vulnerable samples, a macro OMIT-BAD is appended after CFLAGS, which will ignore some potential vulnerabilities in the test cases.

Finally, using the modified makefile, we run the make command, which can compile all source code into corresponding ELF files, divided into two categories: one containing vulnerabilities and the other without vulnerabilities.

Disassembly: Disassembly is the process of converting binary code to assembly code segments. In this paper, we used the disassembly tool IDA Pro 7.0 to read the compiled ELF (Executable and Linkable Format) file obtained from the compilation process and convert it into assembly language form. By analyzing the output of the disassembly, we can obtain information such as program instructions, function names, and comments. Instructions are the basic operations

TABLE 1. Number of NDSS18 datasets under different platforms.

	Non-Vulnerable	Vulnerable	Binaries
Windows	8999	8978	17,977
Linux	6955	7349	14,304
Whole	15,954	16,327	32,281

that the program performs, encoded in binary form and stored in the program. Function names provide a way of naming functions, which helps us better understand the structure of the program. Comments can add human-readable explanations to a program, helping us better understand the function and implementation of the program.

In this paper, we consider a program as a collection of functions and replace calling instructions of each function with the corresponding function body. When extracting the dataset, we chose the functions called by the main function as the entry points for fragment extraction. This is because the main function is usually the entry point of the program and it calls other functions to perform various tasks. By choosing the functions called by the main function as the entry points for fragment extraction, we can extract code fragments related to the main functionality of the program, and ultimately obtain a dataset containing positive and negative samples.

2) NDSS18(Linux)

This dataset is a public dataset in the field of binary code vulnerability detection, which contains code weaknesses CWE119 and CWE399 and CVE (Common Vulnerability and Exposures) samples [32]. It was compiled by Le et al. [21] by extracting functions from the source code dataset NDSS18 according to different platforms, and finally obtained binary code datasets under Windows and Linux systems. Table 1 shows the number of binary codes that contain vulnerabilities and do not contain vulnerabilities under the two platforms. Whole in the Table 1 includes all binary code datasets under Windows and Linux systems.

B. EVALUATING INDICATOR

In order to compare with other methods, this paper refers to the evaluation indicators of [21], including accuracy, precision, recall, and F1-score. The calculation is shown in the equation. In the (18), TP, TN, FP, and FN are calculated by the confusion matrix. TP represents the actual positive data, and the forecast is also positive data. TN represents the data that is verified by the truth, but the forecast is negative. FP represents data that is negative in reality but positive in prediction. FN indicates that the real data is negative, and the forecast is also negative.

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN} \quad (18)$$

$$Precision = \frac{TP}{TP + FP} \quad (19)$$

$$Recall = \frac{TP}{TP + FN} \quad (20)$$

TABLE 2. Experimental results on Juliet Test Suite dataset(unit:%).

Model	Accuracy	Precision	Recall	F1-score	AUC
Instruction2vec	97.4	95.5	99.4	97.4	97.4
O-TextCNN	97.2	95.2	99.4	97.3	97.3
VulDeePecker	95.7	93.5	98.1	95.8	95.7
Our approach	98.9	98.2	99.7	98.9	98.9

$$F1 = \frac{2 \times Precision \times Recall}{Precision + Recall} \quad (21)$$

AUC is the area under the ROC curve, and is an objective evaluation index to measure the advantages and disadvantages of the binary classification model. Its value is between 0 and 1. The higher the value, the better the classification performance of the model.

C. EXPERIMENTAL RESULTS AND ANALYSIS

This paper conducts experiments on the Juliet Test Suite dataset. Moreover, experiments were carried out on the Windows subset, Linux subset, and datasets containing all binary codes of NDSS18, verifying the effectiveness of the model in this paper. At the same time, some classic methods that have been proposed are compared under several datasets.

1) COMPARATIVE EXPERIMENT

The experimental results of this paper on the Juliet Test Suite dataset are shown in Table 2, where bold indicates that the best results are obtained compared with other methods. (1) The Instruction2Vec method in Table 2 efficiently models the instructions in the assembly code segment, and uses TextCNN to extract local features. (2) O-TextCNN uses the embedding method in the literature to express instructions and then uses TextCNN to extract the local features of the embedding matrix. (3) VulDeePecker is a method proposed by Li et al. [16] for source code vulnerability detection, which can also be used for binary code vulnerability detection. The model uses bidirectional LSTM, a Dense layer, and a Softmax classifier.

It can be seen from Table 2 that the model proposed in this paper has a better effect on the Juliet Test Suite dataset, and the results on each evaluation index are higher than the baseline method. Among them, accuracy and F1-score are both increased by 1.5 percent compared with the best results in the Table 2. The baseline methods in Table 2 only extract instruction-level sequence features. Through comparison, it is found that the experiments of the model in this paper are higher than other methods based on instruction-level feature extraction in the Table 2, which further verifies the effectiveness of the method in this paper.

The model proposed in this paper was tested on the Windows subset, the Linux subset, and the Whole dataset of NDSS18. The experimental results are shown in Table 3, Table 4, and Table 5. Among them, MDSAE is a method of maximum divergence sequence autoencoder based on VAE improvement. The model encourages maximum divergence

TABLE 3. Experimental results on NDSS18(Windows) dataset (unit:%).

Model	Accuracy	Precision	Recall	F1-score	AUC
O-TextCNN	83.7	77.3	95.3	85.4	84.3
VulDeePecker	82.5	94.4	76.5	84.5	82.4
MDSAE	84.5	97.2	77.7	86.4	84.4
Our approach	85.4	78.4	97.7	87.0	85.5

TABLE 4. Experimental results on NDSS18(Linux) dataset (unit:%).

Model	Accuracy	Precision	Recall	F1-score	AUC
O-TextCNN	84.8	81.2	92.3	86.4	84.4
VulDeePecker	85.5	80.5	94.2	86.8	85.4
MDSAE	86.9	80.6	97.8	88.3	86.8
Our approach	87.6	81.9	98.3	89.4	86.9

TABLE 5. Experimental results on NDSS18(Whole) dataset (unit:%).

Model	Accuracy	Precision	Recall	F1-score	AUC
O-TextCNN	84.8	78.1	97.3	86.7	82.4
VulDeePecker	83.5	81.0	79.5	84.8	83.4
MDSAE	85.3	98.1	78.4	87.1	85.2
Our approach	86.3	79.9	97.2	87.7	86.2

between defective and non-defective files and can detect binary code vulnerabilities.

According to the results shown in Table 3, Table 4, and Table 5, the proposed model in this paper exhibits improvements in Accuracy, F1-score, and AUC on the Windows subset, the Linux subset, and the Whole dataset of NDSS18. On the Windows dataset of NDSS18, compared with other baseline models in Table 3, the model in this paper has a certain improvement in the evaluation indicators except Precision. On the Linux dataset of NDSS18, our model improves F1-score by 1.1 percent compared to the baseline model.

In the field of computer security, recall is a crucial evaluation metric that represents the proportion of samples predicted as containing defects by the network model out of all actual samples containing defects in the dataset. A higher recall indicates that the network model will have fewer defect samples incorrectly classified as non-defective during prediction. The experimental results demonstrate that on the Window subset and the Linux subset of NDSS18, the proposed model in this paper achieves better recall values compared to other models. F1-score balances precision and recall and holds more reference value in practical applications. It can be seen that the proposed model shows improvements in all evaluation metrics, including F1-score.

2) ABLATION EXPERIMENT

In order to verify the effectiveness of the innovation points proposed in this paper, in this section, we conducted ablation experiments for different modules and different feature fusion methods.

a: DIFFERENT MODULES

In order to verify the effectiveness of the word-level feature extraction module and the instruction-level feature extraction

TABLE 6. Results of ablation experiments for different modules on the Juliet Test Suite dataset (unit:%).

Word level feature extraction	Word level feature extraction	Accuracy	Precision	Recall	F1-score	AUC
✓	×	97.7	96.1	99.0	97.5	97.7
×	✓	98.3	98.2	98.5	98.3	98.3
✓	✓	98.9	98.2	99.7	98.9	98.9

TABLE 7. Results of ablation experiments for different modules on the NDSS18(Windows) dataset (unit:%).

Word level feature extraction	Word level feature extraction	Accuracy	Precision	Recall	F1-score	AUC
✓	×	82.8	74.9	96.8	84.4	83.3
×	✓	83.6	77.2	94.9	85.1	83.8
✓	✓	85.4	78.4	97.7	87.0	85.5

TABLE 8. Results of ablation experiments for different modules on the NDSS18(Linux) dataset (unit:%).

Word level feature extraction	Word level feature extraction	Accuracy	Precision	Recall	F1-score	AUC
✓	×	84.7	78.2	97.0	86.6	84.5
×	✓	85.0	80.3	94.2	86.7	84.7
✓	✓	87.6	81.9	98.3	89.4	86.9

TABLE 9. Results of ablation experiments for different modules on the NDSS18(Whole) dataset (unit:%).

Word level feature extraction	Word level feature extraction	Accuracy	Precision	Recall	F1-score	AUC
✓	×	83.5	77.7	94.8	85.4	83.4
×	✓	84.1	78.8	94.1	85.7	83.9
✓	✓	86.3	79.9	97.2	87.7	86.2

module, we conducted ablation experiments on the Juliet Test Suite and NDSS18, as shown in Table 6, Table 7, Table 8, Table 9. By comparing the performance of different modules, it can be found that the accuracy and F1-score are different when using different modules, and the model performance is the best after considering the word level and instruction level feature modules, which verifies the effectiveness of using multi-level features proposed in this paper.

Analyzing the experimental results reveals that after incorporating both word-level sequence features and instruction-level sequence features, the proposed model in this paper exhibits improvements across all datasets. However, the magnitude of performance improvement varies across different datasets. Through ablation experiments, it was observed that the experimental performance on various datasets of NDSS18 showed significant improvement. The accuracy and F1-score improved the most on the Linux dataset, with accuracy increasing by 2.6 percent and F1-score increasing by 2.7 percent. On the Juliet Test Suite dataset, the experimental results have also improved. From the experimental results, it can be seen that the detection performance of the model that combines word-level features and instruction-level features is better, while the model that only considers word-level features or only instruction-level features exhibits slightly worse performance. It can be concluded that in the

TABLE 10. Results of ablation experiments for different fusion methods on the Juliet Test Suite dataset (unit:%).

Feature fusion methods	Accuracy	Precision	Recall	F1-score	AUC
Feature concatenate	98.8	98.4	99.0	98.7	98.8
Feature addition	98.6	99.3	97.7	98.5	98.5
Weighted feature fusion	98.9	98.2	99.7	98.9	98.9

TABLE 11. Results of ablation experiments for different fusion methods on the NDSS18(Windows) dataset (unit:%).

Feature fusion methods	Accuracy	Precision	Recall	F1-score	AUC
Feature concatenate	84.0	79.1	93.4	85.7	83.7
Feature addition	84.0	77.6	96.3	85.9	83.8
Weighted feature fusion	85.4	78.4	97.7	87.0	85.5

TABLE 12. Results of ablation experiments for different fusion methods on the NDSS18(Linux) dataset (unit:%).

Feature fusion methods	Accuracy	Precision	Recall	F1-score	AUC
Feature concatenate	86.0	80.5	96.6	87.8	86.6
Feature addition	85.7	79.3	97.4	87.4	85.4
Weighted feature fusion	87.6	81.9	98.3	89.4	96.9

TABLE 13. Results of ablation experiments for different fusion methods on the NDSS18(Whole) dataset (unit:%).

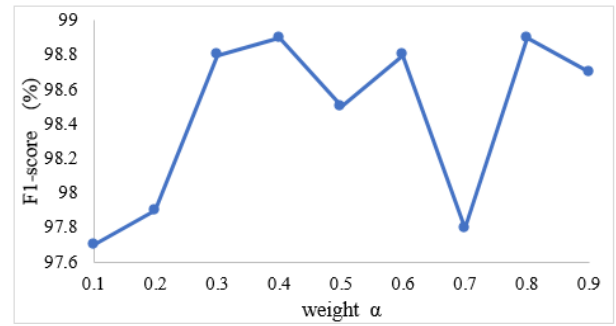
Feature fusion methods	Accuracy	Precision	Recall	F1-score	AUC
Feature concatenate	84.2	78.1	95.5	95.9	84.1
Feature addition	83.8	79.6	91.5	85.2	83.6
Weighted feature fusion	86.3	79.9	97.2	87.7	86.2

multi-level feature fusion vulnerability detection model proposed in this paper, it is indeed effective to consider both word-level sequence features and instruction-level sequence features. The features obtained through feature fusion can better express the syntax, semantics, and other information in the original assembly code segment, thereby improving the detection performance of the model.

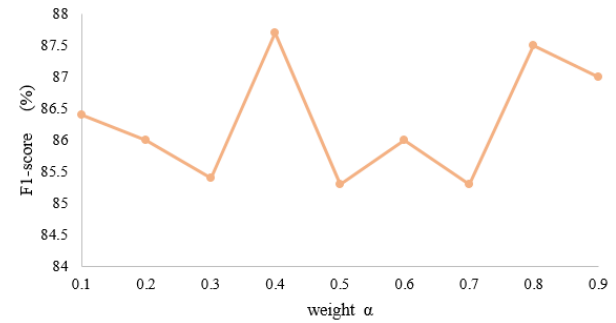
b: DIFFERENT FEATURE FUSION METHODS

For word-level and instruction-level sequence features, this paper uses feature concatenate, feature addition, and weighted feature fusion. The ablation results on two datasets are shown in Table 10, Table 11, Table 12 and Table 13.

Different fusion methods have different characteristics and different representation capabilities. By analyzing Table 10, Table 11, Table 12 and Table 13, it can be seen that the weighted feature fusion method performs best on each data set. Compared with other fusion methods, the weighted feature fusion method improves the most on the NDSS18 (Whole) dataset, with accuracy increased by 2.1 percent, and F1-score increased by 1.8 percent. The analysis shows that the performance of the two fusion methods of feature splicing and feature addition is low, and the weighted feature fusion method considers the contribution of different features to the detection task in this paper, and the experimental results are the best.



(a) F1-score change on the Juliet Test Suite dataset



(b) F1-score change on the NDSS18(Whole) dataset

FIGURE 7. F1-score variation diagram using different weight values α on different datasets.

3) PARAMETER SELECTION

The selection of parameter values in neural network has a great influence on the model effect. In this experiment, two parameters that are more important to the model are selected for discussion, namely α in the weighted feature fusion method and λ in the loss standard deviation regularization coefficient.

a: α IN WEIGHTED FEATURE FUSION

It can be seen from III that the weighted feature fusion method has good experimental results on both datasets. Different weight values indicate a different emphasis on word level and instruction level features. The selection of weight value α is the focus of this section.

Fig.7 shows the impact of different weights on the model. As shown in Fig.7(a), when the values are 0.4 and 0.8, F1-score is higher on the Juliet Test Suite dataset. However, through compared with Fig.7(b), it is found that when $\alpha = 0.4$ is used on NDSS18 (Whole) dataset, F1-score has a higher score than when $\alpha = 0.8$ is used. Therefore, this paper finally chooses $\alpha = 0.4$ as the weight of word-level vector features. Although the NDSS18 (Whole) dataset contains all the data of the windows and linux platforms, and contains more types of vulnerabilities than the Juliet Test Suite dataset, its representation is single, and word-level sequence features have a greater impact on it.

b: COEFFICIENT λ OF STANDARD DEVIATION REGULARIZATION IN LOSS

The standard deviation regularizes the constraint model parameters and obtains the regularization term by multiplying

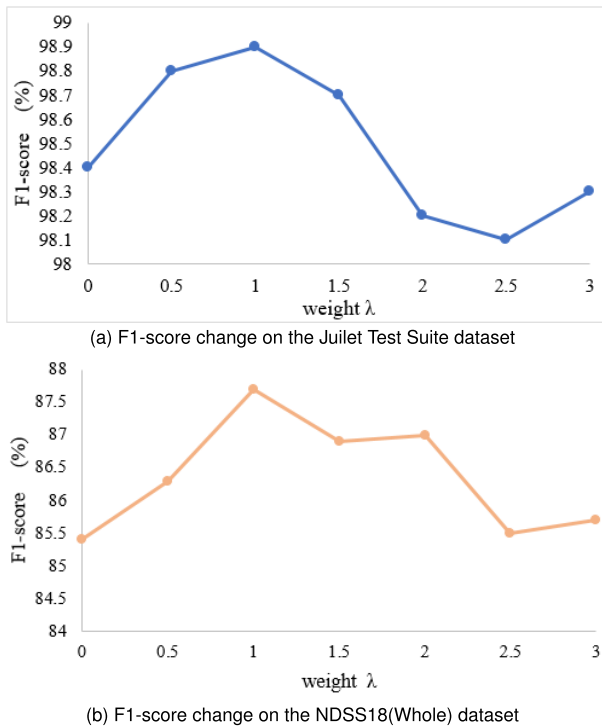


FIGURE 8. F1-score variation diagram using different weight values λ on different datasets.

the standard deviation of the weight matrix so as to reduce the error and prevent over-fitting.

Fig.8 shows the impact of different weight values λ on the model. By analyzing Fig.8, it can be found that when λ value is 1, the results of the model on both datasets are optimal. 0 means that the model does not use standard deviation regularization during training. When the value of λ is between 0 and 1, F1-score is constantly improving, which means that when the value of λ is within a certain range, the standard deviation regularization has a certain fitting ability for the model parameters, which can improve the model performance. However, when the value of λ is greater than 1, the F1-score decreases, so the value of λ should not be too small or too large.

V. DISCUSSION

In summary, the model proposed in this paper can improve the detection performance to a certain extent in the binary code vulnerability detection task. However, this paper also has certain limitations. The main limitation is the scarcity of publicly available binary code datasets. Juliet Test Suite dataset used in this paper is obtained by processing its source code dataset, involving the compilation of source code into binary code and the disassembly of binary code into assembly language. In this process, various issues such as choosing a compiler, different compilation platforms, and implementing a disassembler must be considered in actual situations. The data processing process is very complex. This article refers to the method described in the work of [20], which involves

using makefile provided with the dataset for compilation and the IDA Pro tool for disassembly, resulting in assembly language with Intel syntax.

In future work, further research can be conducted in the data processing phase to investigate the impact of different compilation platforms and disassemblers on the task of binary code vulnerability detection.

VI. CONCLUSION

In this paper, we propose a binary code vulnerability detection model based on multi-level feature fusion. The model proposed in this paper considers the word-level sequence features in the assembly code segment and learns the dynamic vector representation of the same word in different contexts through ELMo model. Then, word-level sequence features and instruction-level sequence features in assembly code segments are fused. In the process of feature fusion, this paper uses methods such as feature splicing, feature addition, and weighted feature fusion to discuss the influence of different features on the binary code vulnerability detection task. Considering the phenomenon of overfitting in model training, this paper uses standard deviation regularization to improve model performance. We conduct experimental evaluation and comparison on the Juliet Test Suite and NDSS18 datasets. On the Juliet Test Suite dataset, the F1-score reaches 98.9 percent, and on the NDSS18 (Whole) dataset, the F1-score reaches 87.7 percent. Compared to the baseline model, the model proposed in this paper exhibits higher accuracy in the task of binary code vulnerability detection.

REFERENCES

- [1] H. Hanif, M. H. N. M. Nasir, M. F. Ab Razak, A. Firdaus, and N. B. Anuar, "The rise of software vulnerability: Taxonomy of software vulnerabilities detection and machine learning approaches," *J. Netw. Comput. Appl.*, vol. 179, Apr. 2021, Art. no. 103009, doi: [10.1016/j.jnca.2021.103009](#).
- [2] F. Lomio, E. Iannone, A. De Lucia, F. Palomba, and V. Lenarduzzi, "Just-in-time software vulnerability detection: Are we there yet?" *J. Syst. Softw.*, vol. 188, Jun. 2022, Art. no. 111283, doi: [10.1016/j.jss.2022.111283](#).
- [3] A. C. Eberendu, V. I. Udegbe, E. O. Ezennorom, A. C. Ibegbulam, and T. I. Chinebu, "A systematic literature review of software vulnerability detection," *Eur. J. Comput. Sci. Inf. Technol.*, vol. 10, no. 1, pp. 23–37, Apr. 2022.
- [4] G. Lin, S. Wen, Q. Han, J. Zhang, and Y. Xiang, "Software vulnerability detection using deep neural networks: A survey," *Proc. IEEE*, vol. 108, no. 10, pp. 1825–1848, Oct. 2020, doi: [10.1109/JPROC.2020.2993293](#).
- [5] X. Yuan, G. Lin, Y. Tai, and J. Zhang, "Deep neural embedding for software vulnerability discovery: Comparison and optimization," *Secur. Commun. Netw.*, vol. 2022, pp. 1–12, Jan. 2022, doi: [10.1155/2022/5203217](#).
- [6] P. Xu, Z. Mai, Y. Lin, Z. Guo, and V. S. Sheng, "A survey on binary code vulnerability mining technology," *J. Inf. Hiding Privacy Protection*, vol. 3, no. 4, pp. 165–179, 2021, doi: [10.32604/jihpp.2021.027280](#).
- [7] S. Alrabaee, M. Debbabi, and L. Wang, "A survey of binary code fingerprinting approaches: Taxonomy, methodologies, and features," *ACM Comput. Surv.*, vol. 55, no. 1, pp. 1–41, Jan. 2022, doi: [10.1145/3486860](#).
- [8] C. B. Sahin and L. Abualigah, "A novel deep learning-based feature selection model for improving the static analysis of vulnerability detection," *Neural Comput. Appl.*, vol. 33, no. 20, pp. 14049–14067, Oct. 2021, doi: [10.1007/s00521-021-06047-x](#).
- [9] R. Scandariato, J. Walden, and W. Joosen, "Static analysis versus penetration testing: A controlled experiment," in *Proc. IEEE 24th Int. Symp. Softw. Rel. Eng. (ISSRE)*, Nov. 2013, pp. 451–460.

- [10] S. Dinesh, N. Burow, D. Xu, and M. Payer, "RetroWrite: Statically instrumenting COTS binaries for fuzzing and sanitization," in *Proc. IEEE Symp. Secur. Privacy (SP)*, May 2020, pp. 1497–1511, doi: [10.1109/SP40000.2020.00009](https://doi.org/10.1109/SP40000.2020.00009).
- [11] C. Beaman, M. Redbourne, J. D. Mummery, and S. Hakak, "Fuzzing vulnerability discovery techniques: Survey, challenges and future directions," *Comput. Secur.*, vol. 120, Sep. 2022, Art. no. 102813, doi: [10.1016/j.cose.2022.102813](https://doi.org/10.1016/j.cose.2022.102813).
- [12] O. Zaazaa and H. El Bakkali, "Dynamic vulnerability detection approaches and tools: State of the art," in *Proc. 4th Int. Conf. Intell. Comput. Data Sci. (ICDS)*, Oct. 2020, pp. 1–6, doi: [10.1109/ICDS50568.2020.9268686](https://doi.org/10.1109/ICDS50568.2020.9268686).
- [13] J. Jurn, T. Kim, and H. Kim, "An automated vulnerability detection and remediation method for software security," *Sustainability*, vol. 10, no. 5, p. 1652, May 2018, doi: [10.3390/su10051652](https://doi.org/10.3390/su10051652).
- [14] R. Zhang, S. Huang, Z. Qi, and H. Guan, "Combining static and dynamic analysis to discover software vulnerabilities," in *Proc. 5th Int. Conf. Innov. Mobile Internet Services Ubiquitous Comput.*, Jun. 2011, pp. 175–181, doi: [10.1109/IMIS.2011.59](https://doi.org/10.1109/IMIS.2011.59).
- [15] Q. Wang, Y. Li, Y. Wang, and J. Ren, "An automatic algorithm for software vulnerability classification based on CNN and GRU," *Multimedia Tools Appl.*, vol. 81, no. 5, pp. 7103–7124, Jan. 2022, doi: [10.1007/s11042-022-12049-1](https://doi.org/10.1007/s11042-022-12049-1).
- [16] Z. Li, D. Zou, S. Xu, X. Ou, H. Jin, S. Wang, Z. Deng, and Y. Zhong, "VulDeePecker: A deep learning-based system for vulnerability detection," 2018, *arXiv:1801.01681*.
- [17] L. Wartschinski, Y. Noller, T. Vogel, T. Kehrner, and L. Grunske, "VUDENC: Vulnerability detection with deep learning on a natural code-base for Python," *Inf. Softw. Technol.*, vol. 144, Apr. 2022, Art. no. 106809, doi: [10.1016/j.infsof.2021.106809](https://doi.org/10.1016/j.infsof.2021.106809).
- [18] H. Wang, W. Qu, G. Katz, W. Zhu, Z. Gao, H. Qiu, J. Zhuge, and C. Zhang, "JTrans: Jump-aware transformer for binary code similarity detection," in *Proc. 31st ACM SIGSOFT Int. Symp. Softw. Test. Anal.*, Jul. 2022, pp. 1–13.
- [19] Y. Lee, H. Kwon, S.-H. Choi, S.-H. Lim, S. H. Baek, and K.-W. Park, "instruction2vec: Efficient preprocessor of assembly code to detect software weakness with CNN," *Appl. Sci.*, vol. 9, no. 19, p. 4086, Sep. 2019, doi: [10.3390/app9194086](https://doi.org/10.3390/app9194086).
- [20] H. Yan, S. Luo, L. Pan, and Y. Zhang, "HAN-BSVD: A hierarchical attention network for binary software vulnerability detection," *Comput. Secur.*, vol. 108, Sep. 2021, Art. no. 102286, doi: [10.1016/j.cose.2021.102286](https://doi.org/10.1016/j.cose.2021.102286).
- [21] T. Le, T. Nguyen, T. Le, D. Phung, P. Montague, O. De Vel, and L. Qu, "Maximal divergence sequential autoencoder for binary software vulnerability detection," in *Proc. Int. Conf. Learn. Represent.*, 2019, pp. 1–15. [Online]. Available: <https://openreview.net/pdf?id=ByloLiCqYQ>
- [22] J. Tian, W. Xing, and Z. Li, "BVDetector: A program slice-based binary code vulnerability intelligent detection system," *Inf. Softw. Technol.*, vol. 123, Jul. 2020, Art. no. 106289, doi: [10.1016/j.infsof.2020.106289](https://doi.org/10.1016/j.infsof.2020.106289).
- [23] K. L. Narayana and K. Sathiyamurthy, "Automation and smart materials in detecting smart contracts vulnerabilities in blockchain using deep learning," *Mater. Today, Proc.*, vol. 81, pp. 653–659, Jan. 2023, doi: [10.1016/j.matpr.2021.04.125](https://doi.org/10.1016/j.matpr.2021.04.125).
- [24] W. Ouyang, M. Li, Q. Liu, and J. Wang, "Binary vulnerability mining based on long short-term memory network," in *Proc. World Autom. Congr. (WAC)*, Aug. 2021, pp. 71–76, doi: [10.23919/WAC50355.2021.9559467](https://doi.org/10.23919/WAC50355.2021.9559467).
- [25] M. Peters, M. Neumann, M. Iyyer, M. Gardner, C. Clark, K. Lee, and L. Zettlemoyer, "Deep contextualized word representations," in *Proc. Conf. North Amer. Chapter Assoc. Comput. Linguistics, Human Lang. Technol.*, 2018, pp. 2227–2237. [Online]. Available: <https://aclanthology.org/N18-1202.pdf>
- [26] M. Sundermeyer, R. Schluter, and H. Ney, "LSTM neural networks for language modeling," in *Proc. Interspeech*, Sep. 2012, pp. 1–4. [Online]. Available: https://www.isca-speech.org/archive_v0/archive_papers/interspeech_2012/i12_01_94.pdf
- [27] A. Salah, M. Bekhit, E. Eldesouky, A. Ali, and A. Fathalla, "Price prediction of seasonal items using time series analysis," *Comput. Syst. Sci. Eng.*, vol. 46, no. 1, pp. 445–460, 2023.
- [28] J. Chung, C. Gulcehre, K. Cho, and Y. Bengio, "Empirical evaluation of gated recurrent neural networks on sequence modeling," 2014, *arXiv:1412.3555*.
- [29] T. Mikolov, K. Chen, G. Corrado, and J. Dean, "Efficient estimation of word representations in vector space," 2013, *arXiv:1301.3781*.
- [30] H. Wei, G. Lin, L. Li, and H. Jia, "A context-aware neural embedding for function-level vulnerability detection," *Algorithms*, vol. 14, no. 11, p. 335, Nov. 2021, doi: [10.3390/a14110335](https://doi.org/10.3390/a14110335).
- [31] M. A. Albahar, "A modified maximal divergence sequential auto-encoder and time delay neural network models for vulnerable binary codes detection," *IEEE Access*, vol. 8, pp. 14999–15006, 2020, doi: [10.1109/ACCESS.2020.2965726](https://doi.org/10.1109/ACCESS.2020.2965726).
- [32] K. Filus, P. Boryszko, J. Domanska, M. Siavvas, and E. Gelenbe, "Efficient feature selection for static analysis vulnerability prediction," *Sensors*, vol. 21, no. 4, p. 1133, Feb. 2021, doi: [10.3390/s21041133](https://doi.org/10.3390/s21041133).



GUANGLI WU (Member, IEEE) was born in Weifang, Shandong, China, in 1981. He received the Ph.D. degree. He is currently a professor. His research interests include network security and artificial intelligence.



HUILI TANG is currently pursuing the master's degree with the Gansu University of Political Science and Law. Her current research interests include artificial intelligence and vulnerability detection.

...