Hindawi
Security and Communication Networks
Volume 2021, Article ID 9997641, 11 pages
https://doi.org/10.1155/2021/9997641

Research Article

A Vulnerability Detection System Based on Fusion of Assembly Code and Source Code

Xingzheng Li, Bingwen Feng, Guofeng Li, Tong Li, and Mingjin He

College of Information Science and Technology, Jinan University, Guangzhou 510632, China

Correspondence should be addressed to Bingwen Feng; bingwfeng@gmail.com

Received 15 March 2021; Accepted 17 July 2021; Published 30 July 2021

Academic Editor: Liguo Zhang

Software vulnerabilities are one of the important reasons for network intrusion. It is vital to detect and fix vulnerabilities in a timely manner. Existing vulnerability detection methods usually rely on single code models, which may miss some vulnerabilities. This paper implements a vulnerability detection system by combining source code and assembly code models. First, code slices are extracted from the source code and assembly code. Second, these slices are aligned by the proposed code alignment algorithm. Third, aligned code slices are converted into vector and input into a hyper fusion-based deep learning model. Experiments are carried out to verify the system. The results show that the system presents a stable and convergent detection performance.

1. Introduction

Software vulnerability detection is crucial to cybersecurity defenses. In recent years, the number of software vulnerabilities reported has grown rapidly with the development of the software industry. According to Common Vulnerabilities and Exposures (CVE) [1], 6447 security vulnerabilities were published in 2016, and this number has increased to 9000 in the first half of 2020. These vulnerabilities have led to many cybersecurity incidents. An effective tool to handle these software vulnerabilities is vulnerability detection. For this reason, many scholars and businesses have devoted much of their energies to the research of vulnerability detection.

There are a wide variety of vulnerability detection methods for discovering vulnerabilities in software. They can be broadly divided into dynamic analysis and static analysis methods. The former identify vulnerable behaviors in the process of analyzing and executing software [2, 3]. Despite a low false alarm rate, this type of methods may miss some vulnerabilities due to the difficulty of analyzing the entire program's behavior [4]. The latter detect software vulnerabilities by analyzing a code without executing it. They have high convergence and, thus, may be more popular than the former. Some static analysis-based detectors explore code similarity to detect the vulnerabilities caused by code cloning [5–7]. However, these approaches are hard to generalize to other types of vulnerabilities. In contrast, pattern-based, especially machine learning-based, detectors can use a large number of software codes to learn the patterns of universal vulnerabilities [8, 9]. Recently, deep learning models have been introduced to extract vulnerability features from program fragments [10–12] and achieved considerable accuracy.

The method based on code metrics [13] selects appropriate metrics to predict vulnerabilities. This method can complete the detection quickly, but with a low accuracy. The graph-based deep learning vulnerability detection method in [14] extracts the graph of the code to predict vulnerabilities. This method has a high detection effectiveness; however, the detection speed is too slow for large-scale code detection. In order to balance the detection speed and accuracy, we use a similar token sequence-based deep learning model as those in [10–12, 15]. These methods extract the token sequences related to the vulnerability information. Some of them extract token sequences from source codes [10, 11], while some use assembly codes. Experimentally we find that, due to the limitations of employing a single model, these schemes do not perform very stably when facing some vulnerabilities, thus reducing their performances. In view of this, we

combine the source code model and assembly code model to enhance the detection ability.

Multimodal machine learning has the ability of processing and understanding multisource modal information. Multimodal fusion is one of the earliest concerns for multimodal machine learning. It processes data from different modalities and obtains more characteristics by extracting multimodal data. Multimodal fusion usually achieves better results than only using a single mode. In audio-visual speech recognition (AVSR), the scheme in [16] fuses visual and audio cues to improve the performance relative to audio-only recognition. The scheme in [17] uses video features fusion with spectral and prosodic speech information for multimodal laughter detection. Compared to using only video features, performance can be improved by up to 3.7%. In multimodal emotion recognition, the scheme in [18] considers video only, audio only, and audio visual for the purpose of emotion recognition. The scheme in [19] uses the early fusion of information from facial expressions, body movements, gestures, and speech to recognize emotions. These methods successfully improved the recognition rate of emotions on the original basis. As a result, we believe that the multimodal fusion of code information could also be usable for vulnerability detection.

In this paper, a vulnerability detection system is proposed by fusing the assembly code and source code models (code and data are available: https://github.com/onstar99/VulnerabilitySystem). Given the source code of a software program, it is compiled to obtain the corresponded assembly code. Then, code slices are extracted from both the source code and the assembly code. After that, a code alignment algorithm is suggested to connect each source code statement with its corresponded assembly code statements. Then, the aligned codes are combined to form a fused slice. These new slices, together with the original source code slices and assembly code slices, are fed into the hyper fusion-based deep learning model for vulnerability detection. The main contributions of this paper include the following:

(1) We improve the performance of the vulnerability detection by fusing the models of assembly codes and source codes

(2) We suggest a simple but effective alignment method between source codes and assembly codes to quickly align the data slices

(3) We collect a vulnerability dataset composed of source and assembly codes, which can be used to train and verify the proposed multimodal-based vulnerability detection method

2. Related Work

2.1. Code Representation for Vulnerability Detection. In order to better express vulnerability characteristics in a deep learning-based vulnerability detector, code representation is usually introduced as an intermediate representation. Code representation commonly used includes code metrics, token sequences, abstract syntax trees, and graphs. Token sequences are directly obtained by statistics at the code level, so they have a strong correlation with vulnerabilities. The scheme in [23] scans the character stream of a code and marks its important information according to the lexical rules of a programming language. Then, the character stream of the code is converted into a token sequence representation. Finally, through vectorization processing such as one-hot and word2vec [24], vectors can be obtained and used as the input of a machine learning model.

2.2. Vulnerability Detection by Source Code. Source codes are computer program files that have not yet been compiled. Many vulnerability detectors are implemented by parsing the source code. Li et al. [10] extracted Syntax-based Vulnerability Candidates (SyVCs) from the source code and converted it to Semantics-based Vulnerability Candidates (SeVCs) through a suggested algorithm. Then, word2vec [24] was used to convert SeVC into vectors that facilitate deep learning. Li et al. [11] used LLVM IR technique and suggested a fine-grained deep learning-based vulnerability detector to locate the vulnerability accurately. Cao et al. [20] extracted abstract syntax tree (AST), control flow graph (CFG), and data flow graph (DFG) from the source code. Then, they are used to generate the code composite graph (CCG), which can be combined with a bidirectional graph neural network for vulnerability detection.

Source codes may have more intuitive information than assembly codes or executable codes. This information can better help detect vulnerabilities. For example, source codes can provide more syntactic and semantic information, which makes it easier to know how the data and inputs drive the paths of execution [25].

2.3. Vulnerability Detection by Assembly Code. Assembly codes are program files obtained by compiling the source codes. Practically, assembly codes might be easier to obtain in comparison with source code files. Grieco et al. [12] developed and implemented VDiscover. This system extracts different feature sets including static features of an assembly code and dynamic features during execution to solve large-scale vulnerability discovery. Xu et al. [21] presented a neural network-based cross-platform method to detect the similarity between assembly codes, which is an aid to detect vulnerabilities. Liu et al. [22] employed the attention mechanism on top of a bidirectional long short-term memory to detect assembly code vulnerabilities. Tian et al. [15] extracted code slices from assembly codes in their vulnerability detection system. A summary of the above studies is presented in Table 1, where we have highlighted the key differences of different works.

It has been found that vulnerability detection on the assembly code level can better solve cross-architecture problems [26]. Moreover, assembly codes are sometimes more sensitive to the semantic errors of a program [27]. This means that the assembly code can be used as a powerful supplement to improve the detector's performance.

Table 1: Reviewed studies for vulnerability detection.

	Data type	Network	Feature representation
SySeVR [10]	Source code	BGRU	SeVCs depicting semantic information induced by data dependency and control dependency
VulDeePecker [11]	Source code	BLSTM	Code gadget depicting data flow and control flow
BGNN4VD [20]	Source code	BGNN	CCG based on AST, CFG, and DFG
VDiscover [12]	Assembly code	Logistic regression, MLP of \single hidden layer, and random forest	Dynamic and static call sequences of library functions
Gemini [21]	Assembly code	Structure2vec	Attributed control flow graph (ACFG)
System [22]	Assembly code	Att-BiLSTM	Static call sequences of binary functions
BVDetector [15]	Assembly code	BGRU	Code slices depicting library/API function calls from binary programs

3. Designed System

3.1. System Overview. The proposed method analyzes the source code and assembly code of software to obtain more comprehensive information. First, we give the definitions of the source code slice and assembly code slice.

Definition 1 (source code slice). A source code slice. S_i is a snippet of a semantically related multiline source code, denoted as $S_i = (s_{i1}, s_{i2}, s_{i3}, \ldots, s_{in})$, where s_{ij}, $1 \le j \le n$, is the jth line in S_i.

Definition 2 (assembly code slice). An assembly code slice. D_i is a snippet of a semantically related multiline assembly code, denoted as $D_i = (d_{i1}, d_{i2}, d_{i3}, \ldots, d_{in})$, where d_{ij}, $1 \le j \le n$, is the jth line in D_i.

Figure 1 shows an example of the source code and corresponded assembly code, where assembly code D is compiled from source code S by using the GCC compiler. s_4 in S corresponds to (d_7, d_8) in D.

The workflow of our system is depicted in Figure 2. It has two phases: the training phase and the testing phase. In the training phase, the system first extracts source code slices from training codes and adds labels to them. Second, the system compiles training codes to assembly codes, from which assembly code slices are extracted and labeled. Third, source code slices and assembly code slices are aligned for further multimodal fusion. Fourth, the system converts source code slices and assembly code slices to vectors. Finally, the system trains a deep learning model by using these vectors. In the testing phase, the system tests the trained model using targeted codes and evaluates its performance.

3.2. Multimodal Network Structure. The designed system fuses assembly codes and source codes to predict the outcome. We use the multimodal hybrid fusion strategy [28] to conduct its network, as shown in Figure 3. It first combines assembly code slices and source code slices via early fusion. Then, these combined slices, together with the source code slices and assembly code slices, are fed into three individual networks. At last, the results of the three networks are used to give the final decision through late fusion.

3.2.1. Code Representation. We use code slices to represent different codes. Given a snippet of the source code, Joern [29] is first used to analyze it and generate a control flow graph (CFG), which represents the real-time execution of a process in the form of a graph. After that, we get the program dependency graph (PDG) from the source code, which is a graphical representation of the control dependency and data dependency among program statements. Finally, CFG and PDG are traversed forward and backward to extract all affected statements. These statements are combined to form a code slice, providing data for vulnerability detection. Figure 4 shows a running example of how we generate a code slice from CFG and PDG of a source code. All the affected statements of each function are captured through CFG and combined into a source code slice based on PDG.

Besides the assembly and source code slices, we further construct fused code slices by applying early fusion to them. We combine each source code slice S_i and the corresponded assembly code slice D_i to derive a new slice $G_i = (S_i, D_i)$. Moreover, redundant information in G_i caused by data fusion is removed, yielding a new set of code slices as part of the network input.

3.2.2. Network Model. The three types of slices are input into three individual networks having the same structure. Each of them consists of five layers: input layer, bidirectional LSTM (BLSTM) layer, dense layer, softmax layer, and output layer, as shown in Figure 5. The input layer is responsible for inputting vector data. The BLSTM layer extracts vulnerability characteristics from vulnerability samples. It contains 300 LSTM units in a bidirectional form (altogether 600 LSTM units). The dense layer reduces the dimensionality of the vector. We set up two dense layers to obtain better detection results. To prevent overfitting, we apply dropout with the value of 0.5 after the dense layers. The softmax layer represents and formats the classification results. At last, the output layer gives a decision. In our experiment, the loss

```
int main (int argc,char *argv[])
{
  float buf[10];

  buf[4105] = 55.55;

  return 0;
}
```

```
_main:
    push    %ebp
    movl    %esp, %ebp
    andl    $-16, %esp
    subl    $48, %esp
    call    __main
    movl    LC0, %eax
    movl    %eax, 16428(%esp)
    movl    $0, %eax
    leave
    ret
```

Figure 1: Demonstration of source code S and corresponded assembly code D.

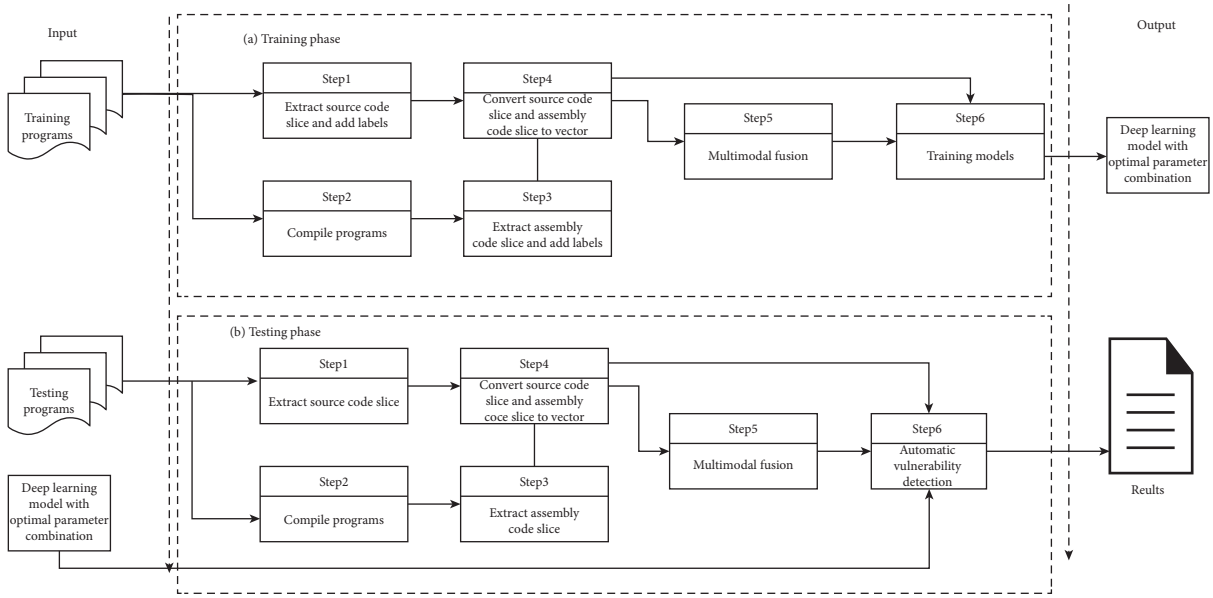

Figure 2: Workflow of the vulnerability detection system.

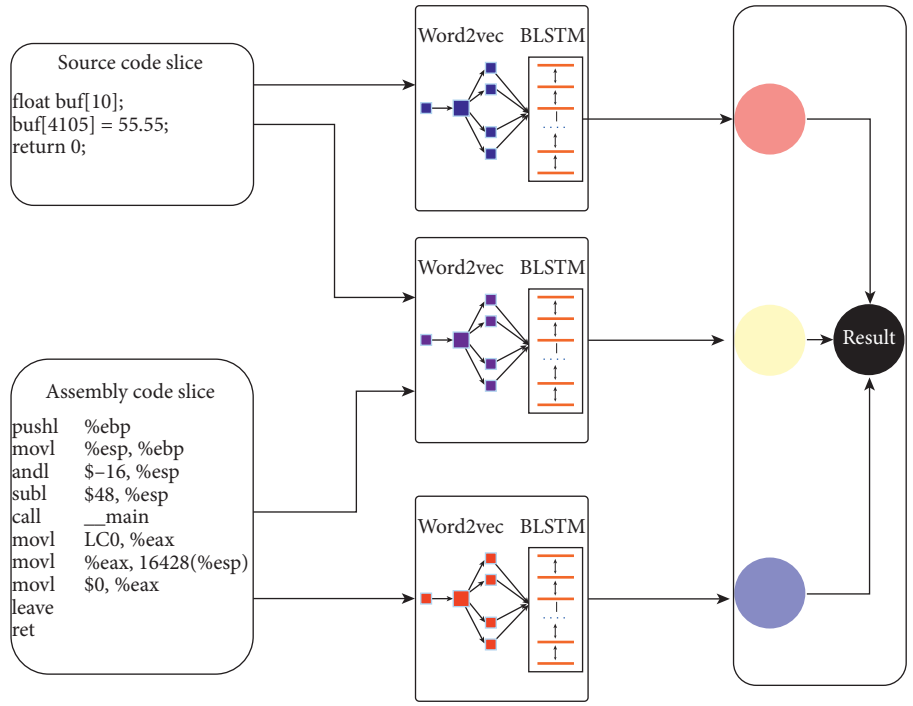

Figure 3: Network structure based on hybrid fusion.

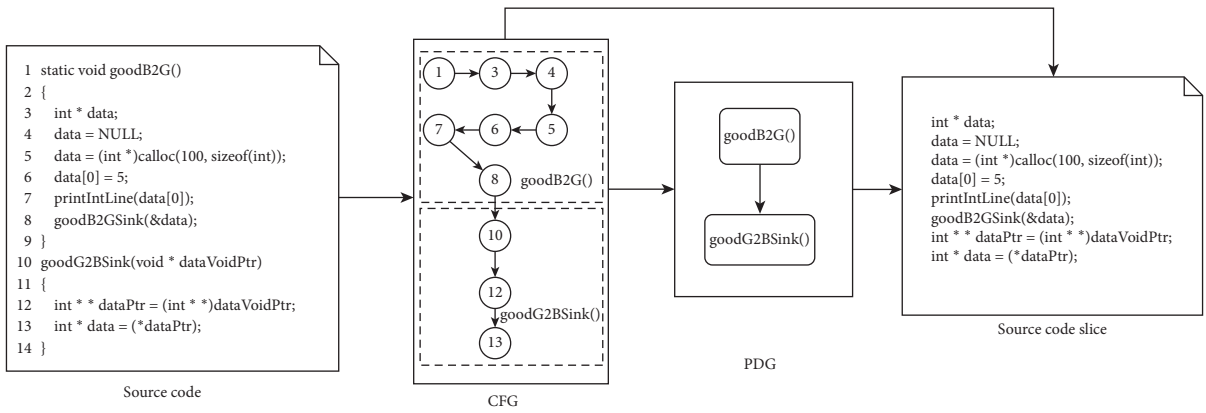

Figure 4: Example of extracting a source code slice.

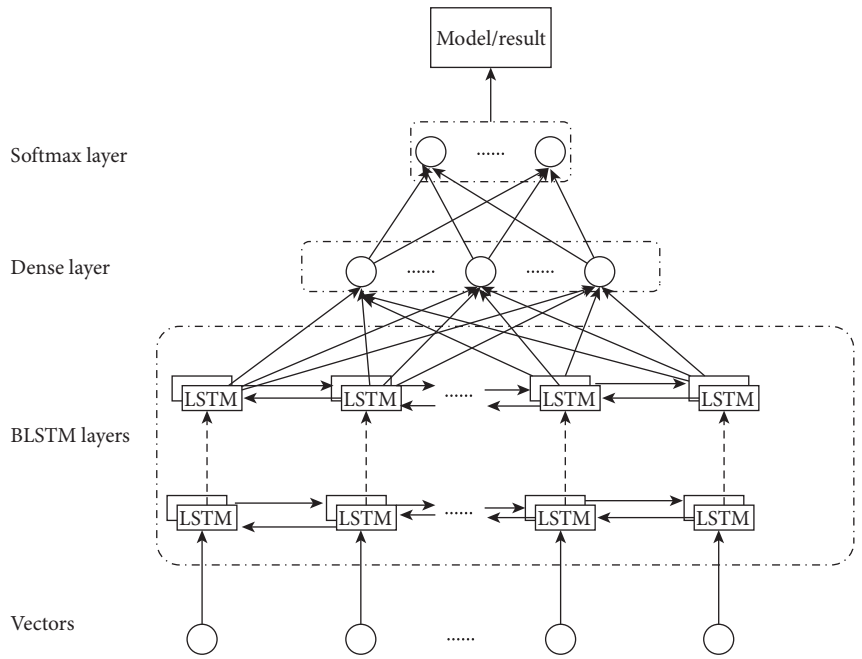

Figure 5: The structure of the individual network.

Table 2: Settings of the proposed network.

Layer name	Activation func.	Node num.
BLSTM	None	600
Dense 1	"LeakyReLU"	300
Dense 2	"LeakyReLU"	300
Softmax	"Softmax"	2

function used is categorical cross-entropy, and the optimizer used is Adamax [30]. During the training phase, we set the batch size with 64 and the epoch number with 30, which gives a good performance. More settings are detailed in Table 2.

3.2.3. Late Fusion Model. The decisions of the three networks may be different. As a result, at the end of our system, these decisions are passed through a voting layer to get the final result. Majority voting is employed in this layer. It is known that a late fusion model usually gets better results than those from a single model. It is expected to avoid erroneous decisions in individual models.

3.3. Training Phase. The training phase is highlighted in Figure 2. It has 5 steps:

Step 1. Source Code Processing.

(1) Extract source code slices: extract source code slices from a training code according to some vulnerability syntax characteristics.

(2) Add labels: give each source code slice a label as the indicator of vulnerabilities ("1" means the presence of vulnerabilities, and "0" means the absence of vulnerability). The method on how to determine the vulnerability of a code slice is detailed in Section 3.6.

Step 2. Program Compiling

(1) Generate an assembly code from the training code using the GCC compiler.

Step 3. Assembly Code Processing

(1) Extract assembly code slices: use the code slice alignment algorithm detailed in Section 3.5 to analyze the assembly code, and extract assembly code slices corresponding to source code slices.

(2) Add labels: similar to the processing of source code slices, each assembly code slice is assigned a label indicating whether it contains vulnerabilities ("1" means yes and "0" means no).

Step 4. Convert Source Code Slices and Assembly Code Slices into Vectors

(1) As shown in Figure 6, the system uses word2vec to convert source code slices, assembly code slices, and hybrid code slices into vector forms, which will be input into the deep learning model.

Step 5. Model Training

(1) Use the data generated from Steps 1–4 to train the network presented in Section 3.2. The parameters and design of the model have been detailed in Section 3.2.

3.4. Testing Phase. As highlighted in Figure 2, the testing phase has 5 steps.

Steps 1–4. Similar to Steps 1–4 in the training phase, except that the label adding is in no need.

Step 5. Automatic Vulnerability Detection.

(1) Input the data generated from Step 1 to Step 4 into the trained model, and the system will give a detection result. Result "1" means that vulnerabilities exist in this code slice, and "0" means there is no vulnerability.

3.5. Code Alignment. In this section, we provide the algorithm employed in data alignment, align_data. The top level of the algorithm is listed in Algorithm 1.

The whole algorithm can be summarized into three stages, namely, (1) pseudocode generation, (2) collection of candidate sets of matched assembly codes, and (3) best match finding. At stage (1), we use IDA Pro [31] to generate pseudocodes, pe_code, from the assembly code (i.e., Line 3 in Algorithm 1), where each statement in pe_code corresponds to several statements $d_{iu}, \ldots, d_{iw}$ in assembly_code. At stage (2), we search for the candidate set of assembly code statements that match s_{ij} (i.e., Lines 4–9 in Algorithm 1). For each p_i, if its statement type (e.g., loop statement and assignment statement) is as same as s_{ij}., the corresponded $d_{iu}, \ldots, d_{iw}$ can be considered as candidate matches of s_{ij}. At stage (3), the Hungarian algorithm [32] is used to get a slice $(d_{i1}, \ldots, d_{ij})$ from the candidate set D_i'. It is considered as a potential match for S_i and combined into D_i. Then, we use string and integer constants, function and library calls, and function declaration information to compute the similarity between D_i and S_i. If this similarity is bigger than a threshold, a satisfactory match is achieved. Otherwise, repeat this stage until a satisfactory match is found (i.e., Lines 11–16 in Algorithm 1).

3.6. Slice Labeling. A simple tool is developed here to give each code slice a label indicating the presence of vulnerabilities. All the vulnerability functions have been defined and given by the NIST Software Assurance Reference Dataset (SARD) [33]. This paper manually populates these vulnerability functions according to their vulnerability codes provided by SARD. The procedure of the tool is described as follows:

(1) Obtain the file names, vulnerability locations, vulnerability types, and other information from the vulnerability file.

(2) Traverse the training codes and the vulnerability information to obtain the vulnerability code and vulnerability function names. Collect this information to populate the vulnerability function library.

(3) Traverse each code slice and vulnerability function library to determine whether the code slice has vulnerability functions or call vulnerability functions.

(4) Divide each code slice into two categories:

   (1) If a code slice has vulnerability functions or call vulnerability functions, it will be labeled with "1."
   (2) Otherwise, it will be labeled with "0."

4. Experimental Results

Our experiments are conducted on a computer with an NVIDIA GeForce GTX 2080 Ti GPU and an Intel Xeon $E5-2678$ v3 CPU operating at 2.50 GHz. The employed program compiler is GCC 9.3.0.

4.1. Evaluation Metrics. In this paper, we use five standard indicators suggested in [34] to measure the performance of the vulnerability detection system. We denote TP as the number of vulnerable samples that are detected as vulnerable (i.e., true-positives), FP as the number of samples that are not vulnerable but are detected as vulnerable (i.e., false-positives), TN as the number of samples that are not vulnerable and are not detected as vulnerable (i.e., true-negatives), and FN as the number of vulnerable samples that are not detected as vulnerable (i.e., false-negatives). Then, the five indicators, namely, accuracy (A), false positive rate (FPR), false negative rate (FNR), precision (P), and F1-measure (F1), are defined as

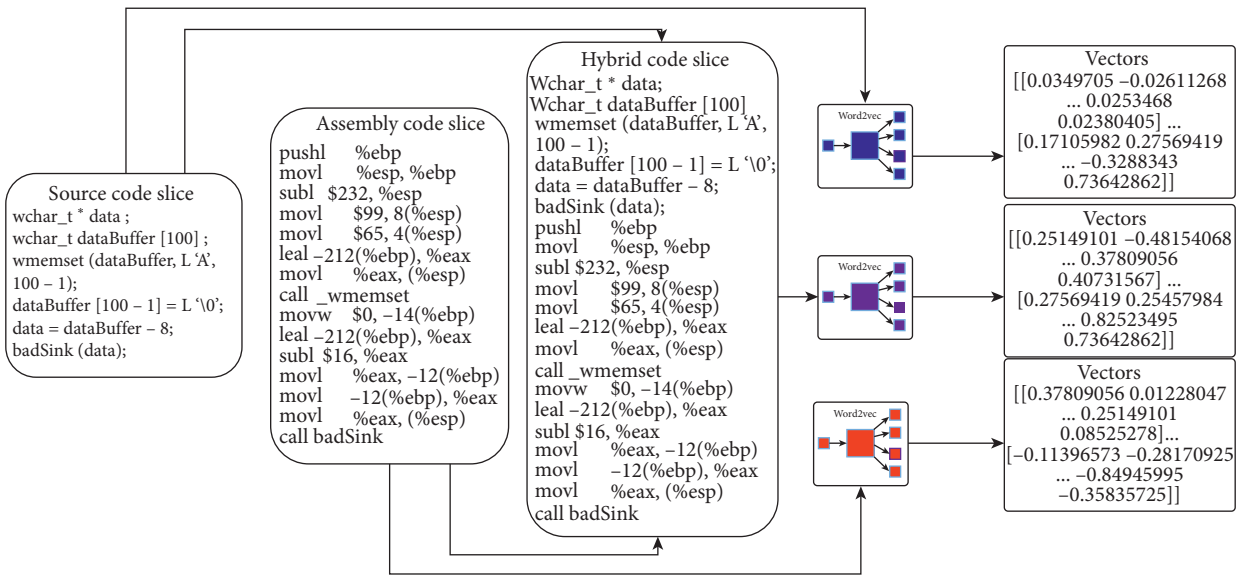

Figure 6: Convert code slices into vectors.

(1) **function** ALIGN_DATA(S_i, assembly_code)
(2) **for** s_{ik} in S_i **do**
(3) pe_code ⟵ IDA_PRO (assembly_code)//p_i in pe_code corresponds to $d_{iu}, \ldots, d_{iw}$ in assembly_code
(4) $D_i' \longleftarrow \varnothing$
(5) **for** p_i in pe_code **do**
(6) **if** STATEMENT_TYPE (p_i) ==STATEMENT_TYPE s_{ik} **then**
(7) $D_i' \longleftarrow D_i' \cup (d_{iu}, \ldots, d_{iw})$
(8) **end if**
(9) **end for**
(10) **end for**
(11) similarity_score ⟵ 0
(12) **while** similarity_score < GOAL **do**
(13) $(d_{i1}, \ldots, d_{ij}) \longleftarrow$ HUNGARIAN $((s_{i1}, \ldots, s_{ij}), D_i')$
(14) $D_i \longleftarrow$ GEN_ SLICE $((d_{i1}, \ldots, d_{ij}))$
(15) similarity_score ⟵ SIMILARITY (S_i, D_i)
(16) **end while**
(17) **return** D_i
(18) **end function**

Algorithm 1: Align_data algorithm.

$$FPR = \frac{\text{FP}}{\text{FP} + \text{TN}},$$

$$FNR = \frac{\text{FN}}{\text{TP} + \text{FN}},$$

$$A = \frac{\text{TP} + \text{TN}}{\text{TP} + \text{TN} + \text{FP} + \text{FN}}, \quad (1)$$

$$P = \frac{\text{TP}}{\text{TP} + \text{FP}},$$

$$F1 = \frac{2 \cdot P \cdot (1 - \text{FNR})}{P + (1 - \text{FNR})}.$$

4.2. Dataset Preparing

4.2.1. Input Preparing. In this paper, we collect a large number of source codes from the NIST Software Assurance Reference Dataset (SARD) [33]. Then, these codes are compiled with GCC on Windows X64 to obtain the corresponded assembly codes. We randomly select 80% of the source codes and corresponded assembly codes as the training dataset, and the remaining 20% are used for the testing.

To granularly evaluate the proposed system, we divide the test cases into the following four categories:

(1) Library/API function call (FC), namely, syntax vulnerability characteristics related to library/API function calls

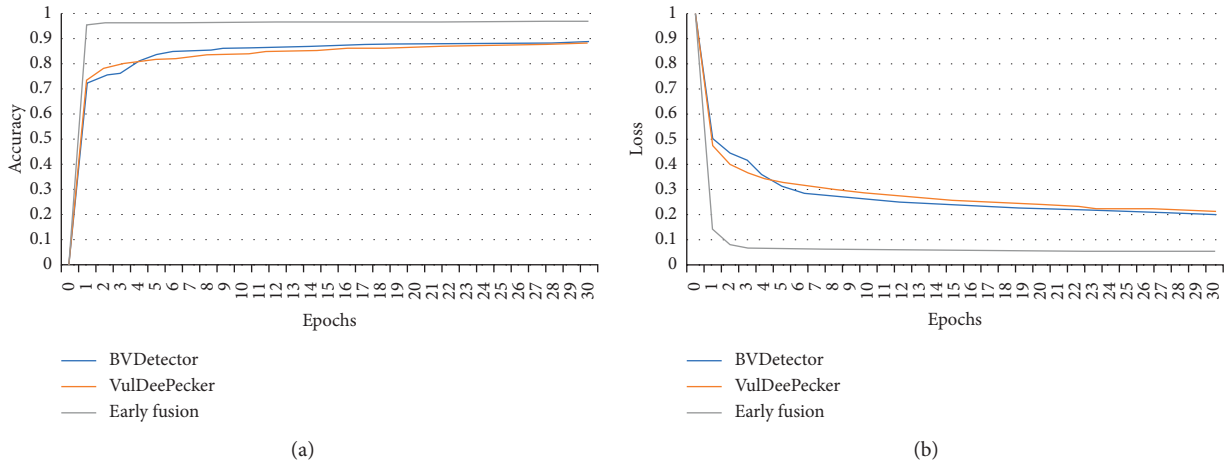

Figure 7: The training accuracy curves (a) and training loss curves (b) of compared systems.

(2) Array usage (AU), namely, vulnerability characteristics related to array element access

(3) Pointer usage (PU), namely, syntax vulnerability characteristics related to pointer usage

(4) Arithmetic expression (AE), namely, syntax vulnerability characteristics related to improper arithmetic expressions

4.2.2. Extracting Code Slices and Adding Labels

(1) Extracting Code Slices. Source code slices are extracted first. We obtain CFG and PDG by parsing the source code. After that, all the statements affected by the library/API function calls are extracted based on CFG and PDG. These statements form source code slices. Then, we use Algorithm 1 to collect assembly code slices corresponding to the source code. In total, 42938 source code slices and assembly code slices are extracted.

(2) Add Labels. We obtain file names, vulnerability locations, vulnerability types, and other information by analyzing program documents in the SARD project. Then, they are used to build a vulnerability function database, where all the code slices are labeled by the tool developed in Section 3.6. In total, we label 16478 code slices with "1," and 26460 code slices with "0."

To ensure effective training of the detection system, 16478 code slices are randomly selected from the 26460 code slices labeled with "0." They are then combined with all the code slices labeled with "1" to form a dataset in each training/testing round, of which 13182 code slices labeled with "0" and 13182 code slices labeled with "1" are randomly selected to train the model, and the rest are used for testing.

4.2.3. Convert Datasets into Vectors. The neural network only accepts vectors, thus we need to convert the data into the vector form, which is carried out by word2vec. Since the effectiveness of our system depends largely on the quality of the word embeddings produced, we first use the constructed dataset as a corpus to train word2vec. It can give a trained word2vec model more suitable for our task. After that, code slices are divided into a series of tokens. Each token is converted through the trained word2vec to obtain a fixed-length vector that can be recognized by the network. In this experiment, the length of each code slice is 50 and the dimension of the word embedding is 100.

4.3. Training and Results

4.3.1. Network Training. In this experiment, we use Keras with TensorFlow to implement the system. The length of the input layer is fixed to be 200, and the number of nodes of each BLSTM is 300. Two fully connected layers with LeakyReLU are employed as its activation functions. The number of nodes is 300, and the dropout is 0.5. The model is optimized by Adamax [30] with a learning rate of 0.002.

4.3.2. Result. We compare the proposed system with the approaches presented in [35] (denoted as VulDeePecker) and [15] (denoted as BVDetector). VulDeePecke detects vulnerabilities at the source code level, while BVDetector detects vulnerabilities at the assembly code level. The training accuracy and training loss curves for these systems are compared in Figure 7. It can be observed that all the compared systems converge fast. The average accuracy of compared systems is close to 90%, while the accuracy of our early fusion system can reach 97%.

Table 3 lists the number of marked code slices by different schemes. It can be observed that some vulnerabilities can be detected by the model based on source codes, but not by the model based on assembly codes. In contrast, some vulnerabilities can only be detected by the model based on assembly codes. Take the code fragment shown in Figure 8 as an example, which is a vulnerability related to out-of-bounds array access. The system, which uses source code slices, may incorrectly detect the

Table 3: Comparison of different metrics on the number of marked code slices.

Metrics	VulDeePecker [35]	BVDetector [15]	Early fusion	Hybrid fusion
Original number of code slices		6591		
Number of code slices marked as safe	2977	3102	3156	3202
Number of code slices marked as vulnerable	3614	3489	3435	3389
Number of correctly marked code slices	5665	5829	6373	6421
Number of incorrectly marked code slices	926	762	218	170

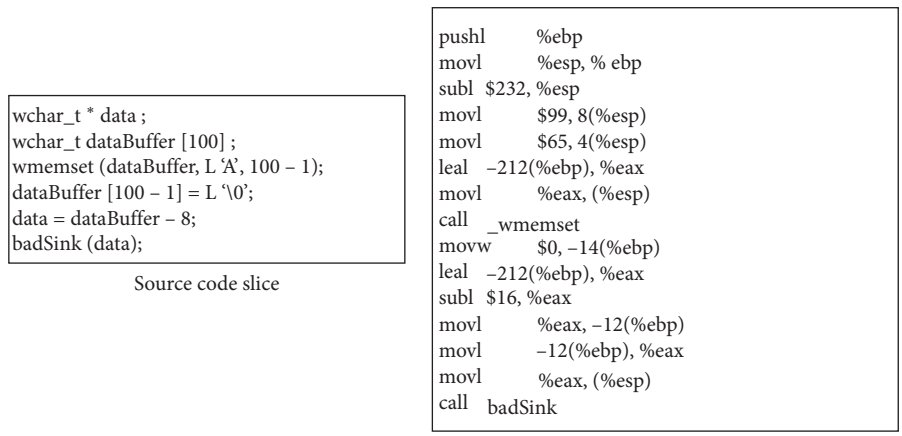

Figure 8: Example of a vulnerability in a source code slice and the corresponded assembly code slice.

Table 4: Test results for different categories.

Kind		FPR (%)	FNR (%)	A (%)	P (%)	F1 (%)
FC	VulDeePecker [35]	9.6	17.5	85.9	91.3	86.6
	BVDetector [15]	7.7	13.9	88.9	92.8	89.3
	Early fusion	**1.1**	5.0	96.8	**98.9**	96.8
	Hybrid fusion	1.3	**4.5**	**97.0**	98.7	**97.0**
AU	VulDeePecker [35]	9.4	17.4	86.1	91.5	86.8
	BVDetector [15]	7.4	13.8	89.1	93.1	89.4
	Early fusion	**1.1**	5.1	96.7	**98.9**	96.8
	Hybrid fusion	1.3	**4.5**	**96.9**	98.7	**97.0**
AE	VulDeePecker [35]	9.7	17.5	85.9	91.2	86.6
	BVDetector [15]	7.2	15.4	88.3	93.0	88.5
	Early fusion	**1.0**	5.9	96.4	**98.9**	96.4
	Hybrid fusion	1.4	**5.0**	**96.6**	98.4	**96.6**
PU	VulDeePecker [35]	6.7	18.1	86.9	93.8	87.4
	BVDetector [15]	7.8	13.9	88.8	92.7	89.2
	Early fusion	**1.1**	5.0	96.8	**98.9**	96.8
	Hybrid fusion	1.3	**4.5**	**97.0**	98.6	**97.0**

vulnerability as normal pointer arithmetic because it is difficult to detect out-of-bounds array access in a calculation. By contrast, the system using assembly code slices can easily detect this vulnerability via memory addresses. As a result, fusing source codes and assembly codes in our model can accurately detect a wider range of vulnerabilities.

The comparison results on the test set are shown in Tables 4 and 5, where a representative static vulnerability mining tool on the market, Flawfinder [36], is also included for comparison. It indicates that our system achieves better results in all aspects compared to other systems. Compared with VulDeePecke and BVDetector, our hybrid fusion-based system can raise the F1 scores by 10%. The score of our

Table 5: Test results for different systems.

	Kind	FPR (%)	FNR (%)	*A* (%)	*P* (%)	F1 (%)
Flawfinder [36]	ALL	10.2	81.3	60.6	48.3	24.6
VulDeePecker [35]	ALL	9.7	17.4	85.9	91.3	86.6
BVDetector [15]	ALL	7.7	13.9	88.9	92.8	89.3
Early fusion	ALL	**1.1**	5.0	96.8	**98.9**	96.8
Hybrid fusion	ALL	1.3	**4.5**	**96.9**	98.6	**97.0**

system is also well above the one obtained by Flawfinder. This confirms the effectiveness of the proposed vulnerability detection system.

5. Conclusion

This paper proposes a system for detecting vulnerabilities in software through deep learning. It combines the vulnerability features at the source code level and assembly code level to improve the ability of vulnerability detection. The system extracts code slices from both the source code and the assembly code of a program. Then, a code alignment algorithm based on string, integer constants, function and library calls, etc., is suggested to align these code slices. The hyper fusion-based deep learning model considers early fusion and late fusion. Early fusion combines aligned source code slices and assembly code slices to generate a new dataset, while late fusion combines the decisions of the deep learning models on the source dataset, assembly dataset, and fused dataset. We implement a prototype and perform systematic experiments. The experimental results show the effectiveness of the proposed system. In future research, we can extend this method to cross languages and cross platforms. Moreover, we could consider additional data besides source codes and assembly codes.

Data Availability

The source code and dataset have been deposited in the GitHub repository (https://github.com/onstar99/VulnerabilitySystem).

Conflicts of Interest

The authors declare that they have no conflicts of interest.

Acknowledgments

This work was supported by the Key R&D Program of Guangdong Province (Grant no. 2019B010136003), National Natural Science Foundation of China (Grant nos. 61802145 and 61932010), Natural Science Foundation of Guangdong Province, China (Grant no. 2019B010137005), Science and Technology Program of Guangzhou, China (Grant no. 202007040004), Fundamental Research Funds for the Central Universities, Opening Project of State Key Laboratory of Information Security, and Opening Project of Guangdong Key Laboratory of Intelligent Information Processing and Shenzhen Key Laboratory of Media Security.

References

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

[2] E. Stepanov and K. Serebryany, "Memorysanitizer: fast detector of uninitialized memory use in c++," in *Proceedings of the 2015 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pp. 46–55, San Francisco, CA, USA, March 2015.

[3] M. Vimpari, "An evaluation of free fuzzing tools," Master's thesis, University of Oulu, Oulu, Finland.

[4] S. M. Ghaffarian and H. R. Shahriari, "Software vulnerability analysis and discovery using machine-learning and data-mining techniques," *ACM Computing Surveys*, vol. 50, no. 4, pp. 1–36, 2017.

[5] S. Kim, S. Woo, H. Lee, and H. Oh, "Vuddy: a scalable approach for vulnerable code clone discovery," in *Proceedings of the 2017 IEEE Symposium on Security and Privacy*, pp. 595–614, San Jose, CA, USA, June 2017.

[6] J. Jang, A. Agrawal, and D. Brumley, "Redebug: finding unpatched code clones in entire os distributions," in *Proceedings of the 2012 IEEE Symposium on Security and Privacy*, pp. 48–62, San Jose, CA, USA, July 2012.

[7] N. H. Pham, T. T. Nguyen, H. A. Nguyen, and T. N. Nguyen, "Detection of recurring software vulnerabilities," in *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*, pp. 447–456, Antwerp, Belgium, September 2010.

[8] A. Hovsepyan, R. Scandariato, W. Joosen, and J. Walden, "Software vulnerability prediction using text analysis techniques," in *Proceedings of the 4th International Workshop on Security Measurements and Metrics*, pp. 7–10, Lund, Sweden, September 2012.

[9] Y. Pang, X. Xue, and A. S. Namin, "Predicting vulnerable software components through n-gram analysis and statistical feature selection," in *Proceedings of the 2015 IEEE 14th International Conference on Machine Learning and Applications*, pp. 543–548, Miami, FL, USA, December 2015.

[10] Z. Li, D. Zou, S. Xu, H. Jin, Y. Zhu, and Z. Chen, "Sysevr: a framework for using deep learning to detect software vulnerabilities," *IEEE Transactions on Dependable and Secure Computing*, vol. 2021, Article ID 3051525, 1 page, 2021.

[11] Z. Li, D. Zou, S. Xu, Z. Chen, Y. Zhu, and H. Jin, "Vuldeelocator: a deep learning-based fine-grained vulnerability detector," arXiv preprint arXiv:2001.02350.

[12] G. Grieco, G. L. Grinblat, L. Uzal, S. Rawat, J. Feist, and L. Mounier, "Toward large-scale vulnerability discovery using machine learning," in *Proceedings of the 6th ACM Conference on Data and Application Security and Privacy*, pp. 85–96, New Orleans, LA, USA, March 2016.

[13] A. Younis, Y. Malaiya, C. Anderson, and I. Ray, "To fear or not to fear that is the question: code characteristics of a vulnerable functionwith an existing exploit," in *Proceedings of the 6th ACM Conference on Data and Application Security and Privacy*, pp. 97–104, New Orleans, LA, USA, March 2016.

[14] Q. Feng, R. Zhou, C. Xu, Y. Cheng, B. Testa, and H. Yin, "Scalable graph-based bug search for firmware images," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pp. 480–491, Vienna, Austria, October 2016.

[15] J. Tian, W. Xing, and Z. Li, "Bvdetector: a program slice-based binary code vulnerability intelligent detection system," *Information and Software Technology*, vol. 123, Article ID 106289, 2020.

[16] G. Papandreou, A. Katsamanis, V. Pitsikalis, and P. Maragos, "Multimodal fusion and learning with uncertain features applied to audiovisual speech recognition," in *Proceedings of the 9th Workshop on Multimedia Signal Processing*, pp. 264–267, Chania, Crete, Greece, October 2007.

[17] S. Bedoya and T. H. Falk, "Laughter detection based on the fusion of local binary patterns, spectral and prosodic features," in *Proceedings of the 18th International Workshop on Multimedia Signal Processing*, pp. 1–5, Montreal, QC, Canada, September 2016.

[18] G. A. Ramirez, T. Baltrušaitis, and L.-P. Morency, "Modeling latent discriminative dynamic of multi-dimensional affective signals," in *Proceedings of the International Conference on Affective Computing and Intelligent Interaction*, pp. 396–406, Memphis, TN, USA, October 2011.

[19] G. Castellano, L. Kessous, and G. Caridakis, "Emotion recognition through multiple modalities: face, body gesture, speech," in *Affect and Emotion in Human-Computer Interaction*, pp. 92–103, Springer, Berlin, Heidelberg, 2008.

[20] S. Cao, X. Sun, L. Bo, Y. Wei, and B. Li, "Bgnn4vd: constructing bidirectional graph neural-network for vulnerability detection," *Information and Software Technology*, vol. 136, Article ID 106576, 2021.

[21] X. Xu, C. Liu, Q. Feng, H. Yin, L. Song, and D. Song, "Neural network-based graph embedding for cross-platform binary code similarity detection," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pp. 363–376, Dallas, TX, USA, October 2017.

[22] S. Liu, M. Dibaei, Y. Tai, C. Chen, J. Zhang, and Y. Xiang, "Cyber vulnerability intelligence for internet of things binary," *IEEE Transactions on Industrial*, vol. 16, no. 3, pp. 2154–2163, 2019.

[23] D. Zou, S. Wang, S. Xu, Z. Li, and H. Jin, "μVulDeePecker: a deep learning-based system for multiclass vulnerability detection," *IEEE Transactions on Dependable and Secure Computing*, vol. 2019, Article ID 2942930, 1 page, 2019.

[24] word2vec, 2018, https://radimrehurek.com/gensim/models/word2vec.html.

[25] T. Le, T. Nguyen, T. Le et al., "Maximal divergence sequential autoencoder for binary software vulnerability detection," in *Proceedings of the International Conference on Learning Representations*, Vancouver, Canada, May 2018.

[26] F. Zuo, X. Li, P. Young, L. Luo, Q. Zeng, and Z. Zhang, "Neural machine translation inspired binary code similarity comparison beyond function pairs," arXiv preprint arXiv: 1808.04706.

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

[28] Z.-Z. Lan, L. Bao, S.-I. Yu, W. Liu, and A. G. Hauptmann, "Multimedia classification and event detection using double fusion," *Multimedia Tools and Applications*, vol. 71, no. 1, pp. 333–347, 2014.

[29] F. Yamaguchi, N. Golde, D. Arp, and K. Rieck, "Modeling and discovering vulnerabilities with code property graphs," in *Proceedings of the 2014 IEEE Symposium on Security and Privacy*, pp. 590–604, Washington, DC, USA, May 2014.

[30] D. P. Kingma, J. Ba, and Adam, "Adam: a method for stochastic optimization," arXiv preprint arXiv:1412.6980.

[31] "IDA pro multi-processor disassembler and debugger," http://www.hexrays.com/products/ida/index.shtml.

[32] J. Munkres, "Algorithms for the assignment and transportation problems," *Journal of the Society for Industrial and Applied Mathematics*, vol. 5, no. 1, pp. 32–38, 1957.

[33] "Software assurance reference dataset," 2018, https://samate.nist.gov/SRD/index.php.

[34] M. Pendleton, R. Garcia-Lebron, J.-H. Cho, and S. Xu, "A survey on systems security metrics," *ACM Computing Surveys*, vol. 49, no. 4, pp. 1–35, 2016.

[35] Z. Li, D. Zou, S. Xu et al., "Vuldeepecker: a deep learning-based system for vulnerability detection," arXiv preprint arXiv:1801.01681.

[36] Flawfinder: http://www.dwheeler.com/flawfinder.