Article

A Flexible Code Review Framework for Combining Defect Detection and Review Comments

Xi Chen [1,2,3,†], Lei Dong [1,2], Hong-Chang Li [1,2,†], Xin-Peng Yao [1,2], Peng Wang [1,2,*] and Shuang Yao [4]

1 Key Laboratory of Civil Aircraft Airworthiness Technology, Tianjin 300300, China; chen_x@cauc.edu.cn (X.C.)
2 School of Safety Science and Engineering, Civil Aviation University of China, Tianjin 300300, China
3 Key Laboratory of Civil Aviation Intelligent Flight, Beijing 100085, China
4 School of Artificial Intelligence, Hebei University of Technology, Tianjin 300130, China
* Correspondence: pwang@cauc.edu.cn
† These authors contributed equally to this work.

Abstract: Defects and errors in code are different in that they are not detected by editors or compilers but pose a potential risk to software operation. For safety-critical software such as airborne software, the code review process is necessary to ensure the proper operation of software applications and even an aircraft. The traditional manual review method can no longer meet the current needs with the dramatic increase in code sizes and variety. To this end, we propose Deep Reviewer, a general and flexible code review framework that automatically detects code defects and correlates the review comments of the defects. The framework first preprocesses the data using several methods, including the proposed D2U flow. Then, features are extracted and matched by the detector, which contains a pair of twin LSTM models, one for code defect type detection and the other for review comment retrieval. Finally, the review comment output function is implemented based on the masks generated by the code defect types. The method is validated using a large public dataset, SARD. For the binary-classification task, the test results of the proposed are 98.68% and 98.67% in terms of precision and F1 score, respectively. For the multi-classification task, the proposed framework shows a significant advantage over other methods.

Keywords: code defect; software security; code slice; code review; deep neural networks

Citation: Chen, X.; Dong, L.; Li, H.-C.; Yao, X.-P.; Wang, P.; Yao, S. A Flexible Code Review Framework for Combining Defect Detection and Review Comments. *Aerospace* **2023**, *10*, 465. https://doi.org/10.3390/aerospace10050465

Academic Editors: Mihaela A. Mitici, Matteo Davide Lorenzo Dalla Vedova, Adam F. Abdin and Anne Barros

Received: 21 March 2023
Revised: 12 May 2023
Accepted: 12 May 2023
Published: 16 May 2023

Copyright: © 2023 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (https://creativecommons.org/licenses/by/4.0/).

1. Introduction

Software defects have caused many catastrophic accidents, which include the Ariane 5 explosion on 4 June 1996 [1]. The cause of this disaster was an overflow condition induced in a program that converted 64-bit floating-point numbers to 16-bit signed integers, which eventually caused the computer to crash. In recent decades, although more academic and industrial efforts have been made, due to the proliferation of software size and volume, the threat posed by software defects is increasing.

Code review is an inspection procedure of the source code by developers other than the author and is recognized as a valuable tool for detecting code defects and improving the quality of software projects [2]. As an important part of airworthiness review, the review of airborne software is an important step for the safety of the aircraft. Given that defects are inevitable, it is critical to detect them in the code at an early phase. Conceptually, a defect is different from a bug. The latter represents an error in the source code that can be detected by the compiler or even the program editor, while the former indicates the abuse of variables, functions, etc., resulting in a potential operational risk or risk of attack on the software. Static code analysis techniques [3–5] can help developers and reviewers quickly locate defects in the source code. As determined by what we are about to present in Section 2, machine-learning-based techniques are the research direction that currently seems to have the most potential for development and practical application.

However, there are some shortcomings in the existing methods. The first is that most of them only perform several experiments for binary-classification tasks, i.e., detecting the presence of a specified defect in a test code. For an input, these methods can simply determine whether it is defective. They do not give details of the type of defect. Therefore, the binary classification results are almost useless in practical application scenarios. Reviewers want to know not only if the input contains a defect but also what the specific type of defect is and even how to fix it. Thirdly, the experiments of these models are often carried out in a highly standardized, artificially written database. In the actual software application, there is a precipitous decline in accuracy because the model may not learn the true defect features but rather the features of the database (e.g., specific variable/function names, etc.) [6]. As a result, this causes a gap between the practical application needs of these methods in the code review process.

Driven by the practical demands, we designed a code defect detection framework called Deep Reviewer. We used a twin deep model to learn two different but interrelated features, namely code defect features and the corresponding review comments described by natural language. When performing defect detection, after vectorizing the source code, the first model identified the type of defect while giving a mask that serves to specify the comments associated with the detected type. Then, the second model output the most relevant review comment(s) based on the textual similarity and the resulting mask. The flowchart of Deep Reviewer is shown in Figure 1.

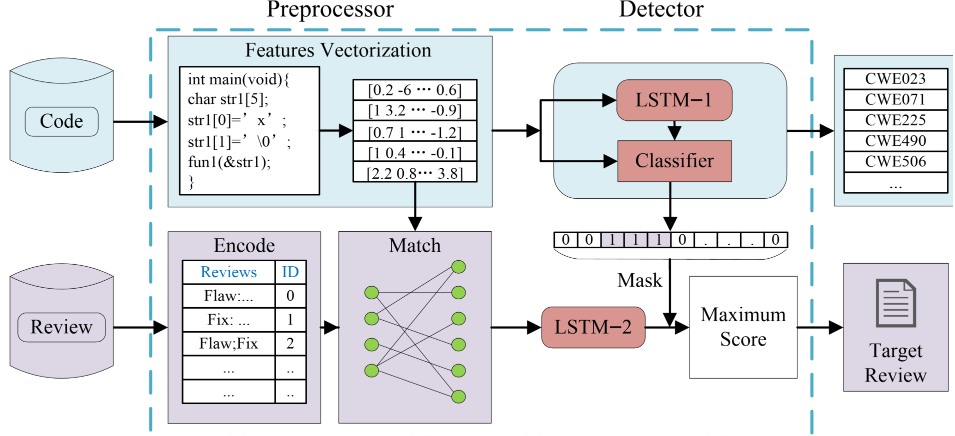

Figure 1. The flowchart of the Deep Reviewer framework.

Deep Reviewer is flexible because the various methods in the two main modules of the whole framework, namely the **Preprocessor** and the **Detector**, are flexible and replaceable according to developers and future researchers. For example, in the preprocessor, we use the D2U flow proposed in this paper for code slicing, which can be replaced by the method in Ref [7,8] according to practical needs; in the detector, we use two LSTM models, one or even all of which can be replaced by other detection models such as BGRU [9], BLSTM [10]. The method design and selection presented in this paper are based on our experimental results.

The main contributions of this paper are summarized as follows:

1. We propose a high-precision and flexible framework, Deep Reviewer. For the first time, the framework associates can detect defect types with corresponding pre-edited review comments, which allows reviewers and programmers to easily make corrections to detected defects, thereby improving the efficiency of the code review process.
2. In the preprocessor part of Deep Reviewer, we propose a code-slicing method called D2U flow. It extracts the life cycle of a variable that is complete and free of running conflicts from Definition-to-Usage. D2U flow avoids the problems caused by different branch statements appearing in the same slice. At the cost of consuming more offline computational resources, shorter slices are obtained, resulting in more accurate defect

localization results. In the detector part, we designed a depth model called twin LSTM. It can correlate two kinds of outputs, i.e., code defect types and review comments.
3. We evaluate the performance on a large-scale database SARD. We compare the proposed method with state-of-the-art approaches. The results demonstrate the apparent advantages of the proposed method over previous work.

The remainder of this paper is organized as follows. Section 2 briefly introduces the related work. Then, we focus on the differences between this topic and previous approaches to explain the particularity of our task. Section 3 presents the preprocessor part of the proposed framework, including a detailed introduction of D2U flow method for code slicing. Section 4 describes the structure of the detector part, in which the solution to output review comments while detecting code defects is introduced in detail. Section 5 gives our experimental results and analysis. Section 6 offers our conclusion.

2. Related Work

The methods for detecting software defects can be categorized as either dynamic or static, depending on their implementation. Dynamic methods are concerned with identifying logic errors and functional losses while the software is running, but they may not cover all possible code paths and statements. These methods should use code instrumentation to cover specific program run paths. Code instrumentation is the method of inserting test code into the program, and the inserted test code is called probes. In order to cover paths as much as possible, dynamic methods require experienced inspectors to design these probes, which are very prone to misses, errors, and inefficiencies. Static methods examine the features of the source code text to identify the type and location of defects and can comprehensively cover all possible paths of the software. Therefore, we focus on static methods in this paper, also known as pattern-based methods.

Existing research on pattern-based defect detection can be roughly classified into two categories: text rule-based and machine learning-based methods [9]. Text rule-based methods are also known as expert experience-based methods. Many pieces of static analysis software such as Flawfinder [6], RATS [11], and Checkmarx [12] use this type of approach. Using such tools require following a strict programming specification or/and writing style. Otherwise, it will suffer from high false-positive rates or/and high false-negative rates. Since human experts generated the pattern of defects [13], these methods can often only be applied to code review processes with the same specific development background (e.g., the same company or organization). A brief comparison of these types of code defect detection methods is given in Table 1.

Table 1. Software defect detection method.

Method	Dynamic Analysis	Static Analysis	
		Text Rule Based	Machine Learning Based
Full Coverage	×	✓	✓
Rule Dependence	✓	✓	×
Code Instrumentation	✓	×	×

In contrast, another class of methods is based on machine learning for automatic defect detection. Compared with natural languages, computer programming languages have similar but stricter and simpler syntaxes and structures [14]. Inspired by high-accuracy achievements in domains such as computer vision (CV) and natural language processing (NLP), machine learning, especially deep learning, has sparked interest in detecting software defects automatically. Given that a source code can be treated as a special case of NLP problems, some researchers [15–17] tried to settle this task by using NLP solutions.

For segment-level code defect detection approaches, Liu et al. [10] used bidirectional long short-term memory (BLSTM) to learn invariant and discriminative code representations from labeled defective and non-defective codes. Cao et al. [18] proposed a bidirectional graph neural network structure named BGNN4VD to detect function-level defects and evaluated it in a database collected from four real C/C++ projects. However, these methods cannot pin down the precise locations of defects because programs are represented in coarse-grained granularity (e.g., program [19], file [20], and function [18,21]).

Some researchers [22,23] have tried to obtain statement-level defect localization so that the defect type is given along with the exact line where the defect occurs. However, the performance indicators were still relatively low (e.g., the metric *Accuracy* was lower than 0.30). We will not carry out further comparisons with such methods.

Slice-level methods have finer granularity for code defect location than the segment-level and better accuracy than the statement-level. Therefore, we also concentrate our study in such a way. In 2018, Li et al. proposed VulDeePecker [7] as the first system showing the feasibility of using deep learning to detect defects at the slice level, which was much finer than the function level. Then, they improved it as µVulDeePecker [8] in 2021. It extended VulDeePecker to detect multiclass defects. Recently, an outstanding work named SySeVR [9] was proposed, which is the first systematic framework for using deep learning to detect defects. SySeVR represented programs as vectors that accommodate the syntax and semantic information. SySeVR first slices the code and then converts the text (source code) into digital sequences via vectorization methods; then, uses neural networks for feature advancement and learning; and finally, outputs defect detection results. It performs well in terms of accuracy but cannot output review comment related to detected defects. Meanwhile, its slicing method may fail in situations such as the ones we describe in Section 3.1.1. Wang et al. [24] used TextCNN to extract the features in the text in 2021 and proposed a software defect detection method based on software code slices that performed a multi-classification detection task on 60 types of CWE defects. The experiment proves that the method is effective, but there is still a lot of room for improvement in indicators such as precision and recall.

In the Sections 3 and 4, we describe the key idea underlying the Deep Reviewer framework and rigorously define the terminology used in the present paper. As highlighted with the dashed rectangle in Figure 1, the proposed framework comprises a preprocessor and detector. Each part is divided into some steps, as elaborated below.

3. Preprocessor

3.1. D2U Flow

3.1.1. Basic Ideas

Step I: For slice-level detection approaches, which are the finest methods at present, dividing the source code into slices is a fundamental and necessary step for feature extraction or pattern detection, just as long sentences are cut into phrases in natural language processing.

It seems reasonable to use an abstract syntax tree (AST) [25] to obtain the overall structure of functions in source code, i.e., SyVC in Ref. [9]. Then, data-dependency and control-dependency [26], which are defined over Control Flow Graph (CFG), are used to obtain code slices [3,7–9,24]. This is because it would give each slice a complete closure of control flow or data flow. Figure 2 shows an example (the path of this sample is ..000\022\771\CWE23_Relative_Path_Traversal_char_Environment_w32CreateFile_03.c) in the SARD [27] database.

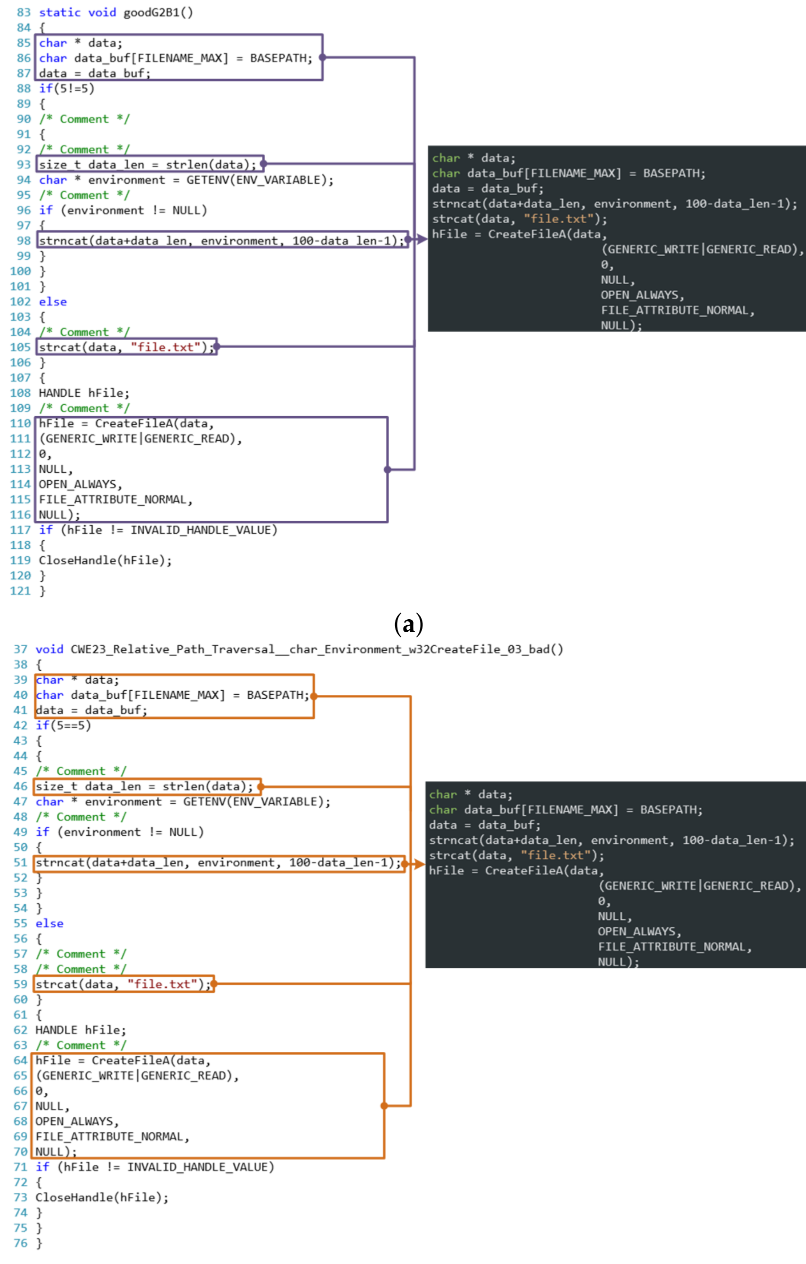

Figure 2. (**a**) "Good" code, (**b**) "Bad" code and their slices corresponding to variable $char * data$.

The line numbers associated with the variable $char * data$ in Figure 2a are 85, 86, 87, 93, 98, 105, and 110. As a result, the code slice of $char * data$ was generated from the Good function, as shown on the right. The SARD database refers to defective code as Bad and vice versa as Good. Similarly, lines 39, 40, 41, 46, 51, 59, and 64 are selected from the Bad code to form a slice about $char * data$, as shown in Figure 2b. Since a manually labeled defect in the database is a statement-level defect, all slices containing the defect are labeled as a specific defect. Note that the slices from Figure 2a,b are identical. It is catastrophic if the latter, extracted from the Bad code, is labeled as defective, while the former is labeled as non-defective.

The occurrence of this phenomenon can contaminate the training set and cause a decrease in the discriminative power of the detection model. Unfortunately, however, it seems to be unavoidable. Essentially, slicing in this way does not take into account the actual execution flow of variables. In code with local scope control keywords such as *if*, *switch*, not all the statements in the slice will be executed at the same runtime.

This motivates us to use a more efficient method to improve the accuracy of the labels in the databases.

3.1.2. Code Slicing Method

Existing dynamic methods [28–30] can obtain the full lifecycle of a variable, but they may fail to solve the problems mentioned in Section 3.1.1. Therefore, the purpose of the proposed method is straightforward: extract the life cycle of a variable that is complete and free of running conflicts from Definition-to-Usage (D2U flow for short).

Algorithm 1 introduces the detailed steps of D2U flow. It searches the source code C to find a variable definition d (line 4 in Algorithm 1), and then, it selects the usages of d, statement by statement. If a usage statement for d is contained within a local scope, we use a temporary vector Sp to store it (lines 7 to 12 in Algorithm 1) and insert Sp into S. Otherwise, this statement is added to S directly. Finally, D2U flow repeats these steps until all the statements of C are checked.

Algorithm 1 D2U flow.

Input: source code C;
Output: slice vector$S$

```
1:  initial S = ∅;
2:  scope-level slice vector Sp = ∅;
3:  repeat
4:      find a variable definition d;
5:      insert d into S;                              ▷ find the definition
6:      find an usage of d as u;
7:      if u in a local scope control scp then        ▷ u is a branch usage
8:          insert scp into Sp;
9:          insert u into Sp;
10:         repeat
11:             find an usage of d as u*;
12:             insert u* into Sp;
13:         until (u* out of scp)                     ▷ completed the search for all branches
14:         insert Sp into S;
15:     else
16:         insert u into S;                          ▷ u is a public usage
17:     end if
18: until (reach the end of C)
```

The most prominent insight from the proposed method is the separation of statements branches into different slices. Each slice constructs a complete and runtime conflict-free D2U flow. We offer detailed steps of the example in Figure 2a, as follows, and the results are given in Figure 3.

- **Step I.1**: Find the definition of ***char * data*** in line 85, and pick its usages in lines 86 and 87 as public usages. After this step, there is a candidate slice $S = [line85, line86, line87]$.
- **Step I.2**: Obtain the branch usage in line 93, contained in a local scope with the keyword ***if*** in line 88. After this step, there is a global slice S and a local slice $Sp_1 = [line8, line93]$.
- **Step I.3**: Obtain the branch usage in line 98, contained in another local scope with keyword ***if*** in line 96. After this step, there is a global slice S and two local slices: Sp_1 and $Sp_2 = [line93, line96, line98]$.
- **Step I.4**: Obtain the branch usage in line 105, contained in the local scope with keyword ***else*** in line 102. After this step, there is a global slice S and three local slices: Sp_1, Sp_2, and $Sp_3 = [line102, line105]$.

- **Step I.5**: Find the public usage in line 110 without a local scope. Insert local slices into S to make three global slices $S_1 = S + Sp_1$, $S_2 = S + Sp_2$, $S_3 = S + Sp_3$. Then, add line 110 into these slices.
- **Step I.6**: Complete the source code search and output the slice vector $S = [S_1, S_2, S_3]$, as shown in Figure 3.

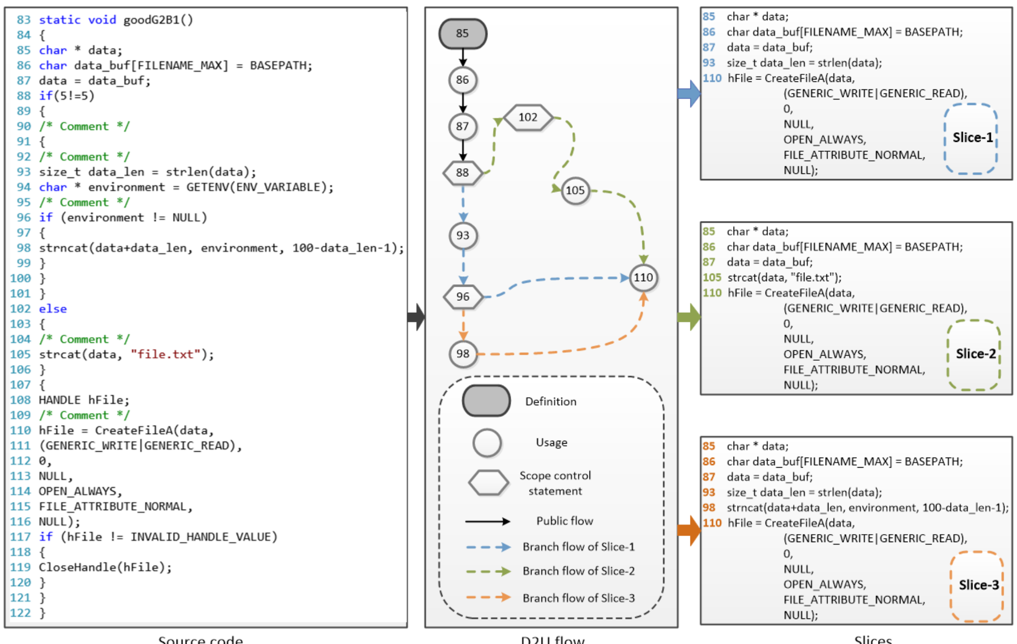

Figure 3. An example of code slicing based on D2U flow.

3.1.3. Slice Labeling

Step II: Labeling slices correctly is a crucial step to ensure that features can be reasonably represented and detected.

Although we also obtain the exact same three slices in Figure 2a as in Figure 3, we can then take full advantage of the statement-level labels in the database to filter and relabel slices. For the example of Figure 2a, the tag /∗Dead Code∗/ in the comments can be used to remove the miscellaneous items Slice-1 and Slice-3 and keep Slice-2 labeled as non-defective. Simultaneously, we relabel the Slice-1 and Slice-3 of Figure 2b as defective based on the original statement-level label and remove Slice-2. Consequently, the slice vectors of Figure 3 corresponding to variable *char* data* are non-defective $S_G = \{Slice - 2\}$ and defective $S_B = \{Slice - 1, Slice - 3\}$, respectively.

There are two main advantages of the D2U flow for slicing as follows:

1. Since slice-level detection does not use the original labels in the database during the training phase but needs relabel slices, it solves the problem that the same content slices are labeled as different types, thus improving the accuracy of slice labeling and the objectivity of the test results based on the slices.
2. Compared with previous work in Ref. [9], for the same database, since D2U flow splits data flow according to the execution conditions. It will obtain more slices, but the average slice length is shorter. Considering that a defective statement is contextualized, it has not been possible to implement statement-level detection so far, i.e., we cannot determine whether a single statement is defective by itself. Therefore, shorter slices indicate that D2U flow can provide more accurate defect localization.

3.2. Feature Representation

Deep neural networks need numerical inputs to learn target features, whereas code text inputs are not supported. Therefore, the preprocessor needs to encode text-based

code into groups of word vectors, known as word embedding. Once word embedding is performed, the textual string will transform into a collection of numerical vectors, with each vector representing a specific word. "Word" refers to the fundamental component of the sentence, encompassing both words and symbols. Several vectorization methods are commonly used, including word2vec, fast-text, and doc2vec [31,32]. In particular, Deep Reviewer utilizes word2vec for text vectorization.

The pre-training model of Word2vec is created by training the thesaurus, and then, the model is fed with the words that require vectorization to acquire the most appropriate word vector. There are two training techniques used in Word2vec to obtain the word vector: CBOW and skip-gram. The former utilizes context to fill in missing words, while the latter predicts context using known words. Although it may not be the optimal word embedding method, Word2vec is acknowledged for its quick and adaptable characteristics, making it satisfactory for vectorizing code and review comment.

3.3. Binding of Review Comments to Defective Samples

We generated a collection of review comments that covers all the relevant reviews for the defect types involved in the experiment. One review comment is encoded in a one-hot code, i.e., each review comment has a corresponding and unique code. This procedure is shown as **Encode** in the **Preprocessor** part of Figure 1.

For review comments belonging to the same defect type, their encoding is more similar in the vector (feature) space. We gave different encoding weights based on the relevance of the review and defect analogues.

Consider that, for a given defective code f with defect type $type(f)$ and its comment rc, as f and rc have a one-to-one correspondence, in the training phase, for f, all comments will be negative samples except for rc. It is impossible to train the model of review comments. Therefore, we regard all reviews with the same defect type $type(f)$ as candidate reviews $[can_1, can_1, ..., can_N] \in type(f)$. Since the one-to-one relationship poses the problem of under-sampling of positive samples. We assign a fuzzy label to the other candidates, with x, $(0 < x < 1)$. The similarity of f and can_i determines the value of x. Reviews that do not pertain to the same defect type are marked as 0. Then, the softmax function is applied to the output layer of the review comments model, limiting output values in the range of [0, 1]. The proximity of a value to 1 indicates a high similarity between the input and the labeled code, and vice versa.

For output review comments, the Deep Reviewer framework is essentially intended to assist coders and reviewers in modifying the code, not for them to be obligated to follow the output. Therefore, we select the top N review comments with the top N similarities as the output to provide a precise and broader reference. This not only increases the number of available review samples but also ensures that the matched review comments are all related to the code under test. Then, the one-hot codes of review comments are matched with the defective samples in the training database via manual labeling. The process is to bind the defective samples and the review comments, which is shown as **h** in the **h** part of Figure 1.

4. Detector Based on Twin LSTM

4.1. Twin LSTM Structure

In processing sequence data features, LSTM (Long Short-Term Memory)-like networks outperform CNN (Convolutional Neural Network) and classical RNN (Recurrent Neural Network) models. Benefiting from their gating mechanism, they are also better than a GRU (Gate Recurrent Unit) network in extracting complex data features. However, for long sequence code data, there are still limitations in learning deep features. In this paper, we propose a deeper LSTM network structure containing three layers, namely, triple-cell LSTM. By using the text features extracted from the single-layer LSTM as input data, the three LSTM networks are connected to learn the deep data features of the code defect because we believe a deeper LSTM can build more complex feature representations

compared with a shallow one, so as to have a better extraction effect on deep complex features for a better long sequence feature memory effect.

Before choosing the proposed triple-cell LSTM as the backbone, we tried to use other models, including CNN, BGRU [9], and BLSTM [10] in the Deep Viewer framework. In the multi-classification experiments with different numbers of defect types (experiments in Section 5.2.2), regarding indicator ACC, the comparison result is less than 2%, while for the indicator MCC, the comparison result is no more than 5%. The proposed triple-cell obtained the best overall performance. Meanwhile, the training time required for the models in Refs [9,10] are much more than ours.

For the detector in the proposed Deep Reviewer framework, it uses two triple-cell LSTM models in parallel. The first triple-cell LSTM(LSTM-1) is used to extract and detect the defect type. If the input is judged to be defective, LSTM-2 retrieves the database, selecting the most similar one to the input sample and outputting the corresponding review comments. These two structurally similar models are independent yet interact with each other in messages. We call this combined structure twin LSTM, as illustrated in Figure 4.

In the Section 4.2, we describe in detail this twin LSTM model passing messages to ensure that the output review comments are more informative.

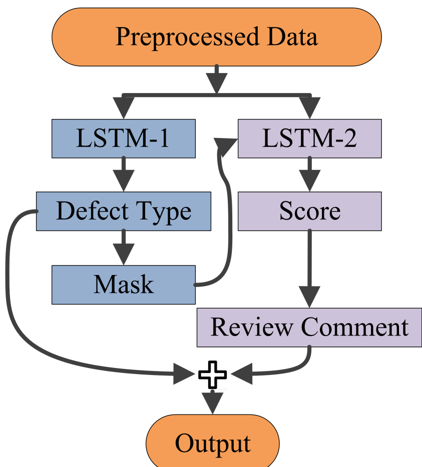

Figure 4. Twin-LSTM structure.

4.2. Detector Workflow

The proposed **Preprocessor** first performs vectorized coding to convert the text data into vector data that can be computed and learned. The code is then manually labeled to correspond to the reviews, as shown in Figure 1. For the LSTM-1 in Figure 4, it is used to learn the code vector data and outputs two results: the defect type and an N-hot mask used to assist the LSTM-2 output review comment. Two judgments are drawn from our experiment results and the analysis:

- If the text similarity of the code is used to train and discriminate the defect types, it tends to cause overfitting of the detection model and focuses the model's attention on the surface text features rather than the mechanism of defect generation.
- The higher the textual similarity of two pieces of code, the more similar their details are and the more similar the causes of the defects are, within a limited range of comparison, provided that they clearly have the same defect type. Therefore, the more similar the review comments are for them.

Based on the above judgments, the detector first detects the defect type using one model in the twin LSTM and, at the same time, gives a one-hot mask based on the detection result, which limits the matching range of the text similarity matching process to a maximum extent, thus increasing the confidence in the output of review comments.

The mask is superimposed on the output of the review comments model (LSTM-2), thus eliminating the problem of the second model finding high similarity text features among other error types.

LSTM-2 takes the code vector as the sample input and the review code as the label input. As we introduced in Section 3.3, in order to obtain a more objective review matching results, we assign weights to the review codes and generate fuzzy labels, i.e., unique hot code labels with values between 0 and 1. Finally, the output of LSTM-2 is filtered with the obtained N-hot mask to obtain several scores, and the highest scoring review comment is selected as the output result. Since the N-hot mask is generated based on the predicted results of LSTM-1, the review comments under the predicted defect types in the matching results of LSTM2 are retained as the candidates. For others, the scores are set as 0. Therefore, the candidate comments belong to the defect types predicted by LSTM-1, which improves the result confidence.

In the actual detection task, the input source code is first vectorized, then the defect classification result and the corresponding N-hot mask are obtained by LSTM-1 prediction. Then, each review comment in the database is given a score based on the prediction of the LSTM-2 filtered by the mask. The highest-scoring review comment is selected as the output and combined with the defect types output by LSTM-1 as the overall output of the framework.

5. Experimental Results

In this section, we carry out experiments to evaluate the proposed framework in terms of defect detection performance. In order to make a fair comparison with state-of-the-art methods, i.e., TextCNN [24] and SySeVR [9], we select the same databases Software Assurance Reference Dataset (SARD) [27] as those methods.

The SARD database is a growing collection of test programs with documented weaknesses. Test cases vary from small synthetic programs to large applications. The programs are in C, C++, Java, PHP, and C# and cover over 150 classes of weaknesses. In this paper, we use the November 2022 version of the dataset and perform the experiments on C C++ language files. In theory, the proposed approach is applicable to any programming code in text form, attributed to the learning ability of the deep learning model. The reason we chose C/C++ programing language for our experiments is its widespread use in the field of airborne software. To ensure sufficient training samples, we selected 80,782 samples with 68 classes of defects. Each class contains over 100 samples. Among them, there are 20 categories with more than 1000 samples. For the programs collected from SARD, we randomly selected 80% of them as the training set (i.e., for training and validation) and the remaining 20% of programs as the test set (i.e., for testing).

The training process is divided into two LSTM models training. The training process is divided into triple-cell LSTM training and word2vec model pre-training. The LSTM model has three hidden layers with 256 neurons and a dropout rate of 0.3. It uses Adam as the optimizer, softmax as the activation function for the output layer, and cross-entropy loss as the chosen loss function. The word vector dimension of the Word2vec pre-training model is set to 64. The sliding window size is 5, the learning rate is 0.001, and the number of iterations is 10.

Experiments are carried out on a computing server with Intel Xeon 4309Y, 2.8G Hz and 32G RAM, NVIDIA Tesla T4, and only one GPU is used. The software environment is Linux ubuntu 22.04, TensorFlow 2.4.0, and Keras 2.4.3. All the testing programs are written in Python 3.10.

5.1. Performance Indicators

To ensure the objectivity and fairness of the comparison experiments as much as possible, we choose the same performance indicators as our comparison methods. In addition, we use multiple ways to calculate these performance indicators to show the comparative performance of each comparison method with different evaluation focuses. In the exper-

iments, True-Positive (TP) denotes the number of defective samples that are detected as defective. False-Positive (FP) denotes the number of samples that are not defective but are detected as defective. True-Negative (TN) denotes the number of samples that are not defective (dubbed non-defective) and are detected as not defective, and False-Negative (FN) denotes the number of defective samples that are detected as not defective. The effectiveness of defect detectors can be evaluated by the following widely used performance indicators:

1. **ACC**: It measures the proportion of correctly detected among all samples and can be calculated using Equation (1).

$$ACC = \frac{TP + TN}{TP + FP + TN + FN}. \tag{1}$$

2. **MCC**: MCC is the abbreviation of Matthews correlation coefficient. The true-class and predicted-class are considered two binary variables, and their correlation coefficients are calculated as Equation (2). A prediction will yield a high score only if it obtains good results in all four confusion matrix classes (TP, TN, FN, and FP).

$$MCC = \frac{TP \times TN - FP \times FN}{\sqrt{(TP + FP)(TP + FN)(TN + FP)(TN + FN)}}. \tag{2}$$

3. **Macro Average**: Assuming that there are common L classes, each class is given the same weight, which characterizes the overall accuracy performance of the database, including the following indicators.
P_{Marco}: It is the precision of truly defective samples among the detected (or claimed) defective samples. P_{Marco} can be calculated using Equation (3).

$$P_{Marco} = \frac{\sum_{i=1}^{L} Precision_i}{|L|}. \tag{3}$$

R_{Marco}: It is the recall value within Marco. It refers to the proportion of samples that are predicted to be positive in a true positive class. R_{Marco} can be calculated using Equation (4).

$$R_{Marco} = \frac{\sum_{i=1}^{L} Recall_i}{|L|}. \tag{4}$$

Then, we obtain the final Macro F1 calculation as follows:

$$F1_{Marco} = \frac{2 \times P_{Marco} \times R_{Marco}}{P_{Marco} + R_{Marco}}. \tag{5}$$

4. **Weight Average**: Unlike the Marco average, the weight average gives different weights for each class based on the amount of data it contains, so that classes with smaller amounts of data are not overlooked. This method includes the following indicators, in which the parameter w_i refers to the normalized weights, indicating the percentage of sample size in ith class.

$$P_{weight} = \frac{\sum_{i=1}^{L} Precision_i \times w_i}{|L|}. \tag{6}$$

$$R_{weight} = \frac{\sum_{i=1}^{L} Recall_i \times w_i}{|L|}. \tag{7}$$

$$F1_{weight} = \frac{2 \cdot P_{weight} \cdot R_{weight}}{P_{weight} + R_{weight}}. \tag{8}$$

5.2. Comparisons and Analysis

There are potentially many kinds of defects in the process of developing software. Our testing datasets also provided details of the type-code for these defects. Unfortunately, previous studies offered experimental results only for binary classification task. That is, these experimental results can only analyze whether the code has a specific type of defect or is correct. It is still far from the goal of automatically identifying defects in the practical software applications, much less providing developers with correction suggestions.

In this section, experiments are carried out for three different tasks to evaluate the comparison methods. Firstly, Section 5.2.1 presents the comparison results of the binary task, in which similar experiments and evaluation indicators are given with previous work. Secondly, the methods are further evaluated in a multi-classification task to demonstrate the performance of the comparison methods in Section 5.2.2. Thirdly, we offer an introduction about our output of review comments in Section 5.2.3.

5.2.1. Binary Classification Task

In the experiment, similar metrics are adopted as previous methods for a fair comparison. For an input, all methods output one of two detection labels, i.e., defective or non-defective. We randomly select a defect type in the database for testing, then repeat the random selection and testing ten times. The obtained experimental results given in Table 2 are the average of the ten random tests.

Table 2. Binary classification result (%).

Method	R	P	ACC	F1
SySeVR	94.15	93.35	93.32	93.39
TextCNN	93.30	94.15	93.03	93.41
Proposed	98.68	98.87	98.68	98.67

In the Deep Reviewer framework, since the detector used in this paper is the well-known LSTM model, we believe that the better results obtained by the proposed method cannot be mainly attributed to the advantages of the chosen deep network mode. Analyzed via the progressive change in the dynamics of the input data in different steps, the difference in the results in this experiment may be more attributable to the differences in the preprocessor. More explicitly, the proposed D2U flow offers more accurate code slicing results.

For the actual process of code review, the binary result of “this code file is defective” or “this code file is not defective” is almost meaningless. Both the reviewer and the coder want more specific answers, including where the defect exists, the type of defect, and how to fix the defect. Therefore, we extended the experiment to the multi-classification task.

5.2.2. Multi Classification Task

Our objective is to design and compare the proposed framework for multi-classification task. Let labels $L = \{0, 1, 2, ..., n\}$ denote the set of defect types. In the following experiments, $L = 0$ means the detection result is “non-defective” and other values of $L = i, 0 < i \leq n$, correspond to a Common Weakness Enumeration IDentifier (CWE-ID), which is the outcome of a community effort at categorizing defects [33].

Set up a hypothesis: as the defect types increase, the distinguishability of features decreases, causing a diminution in model detection performance. To confirm this, we compared the trend of loss decline in different classification tasks when the proposed model was trained. The results are given in Figure 5.

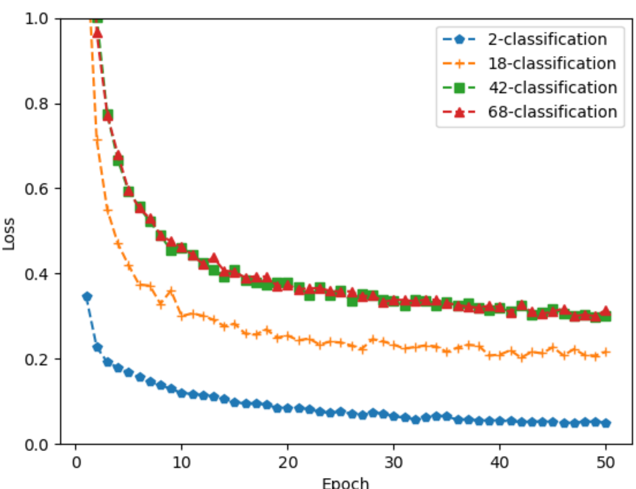

Figure 5. Loss value curves for different quantitative defect types.

The curve of the two-classification model achieved the minimum loss value at the earliest convergence epoch. The 10-classification model shown the same superiority as the other two models, while the curves of the 50-classification model and 100-classification model almost coincide. We suppose that, when the number of defect types is less than 50, the above hypothesis can be partially verified. Furthermore, the results in Figure 6 proved that the hypothesis seems valid. We can see that, as the classification task increases from 2 to 68 classifications, a series of characterization accuracy metrics, including *ACC* and *P*, decreases, indicating that the multi-classification problem is more complex and challenging.

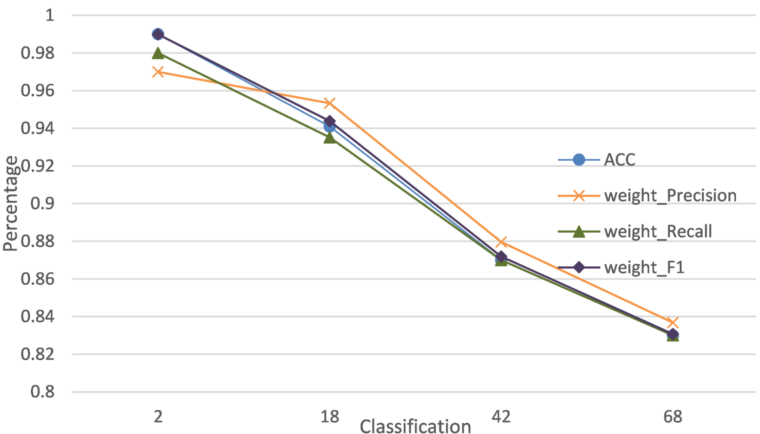

Figure 6. The trend curve of accuracy metrics decreases with the increasing number of classifications.

Therefore, we tried to compare the performance of the methods in the face of different numbers of defect types. Among them, 18 classification results are given in Table 3, and 42 and 68 classification results are given by Tables 4 and 5, respectively.

Table 3. Results of the 18-classification model (%).

Method	Macro			Weight			MCC	ACC
	R	P	F1	R	P	F1		
TextCNN	90.75	83.11	86.30	93.03	94.15	93.39	73.46	93.03
SySeVR	87.24	85.47	86.32	93.62	93.80	93.70	72,69	93.62
Proposed	98.67	98.68	98.67	98.67	98.68	98.67	97.34	98.67

Table 4. Results of the 42-classification model (%).

Method	Macro			Weight			MCC	ACC
	R	**P**	**F1**	**R**	**P**	**F1**		
TextCNN	47.61	62.73	51.68	92.22	91.24	91.38	58.10	93.03
SySeVR	47.14	47.20	51.10	92.61	91.56	91.70	60.32	92.61
Proposed	94.81	95.07	94.83	98.66	98.68	98.67	97.25	96.27

Table 5. Results of the 68-classification model (%).

Method	Macro			Weight			MCC	ACC
	R	**P**	**F1**	**R**	**P**	**F1**		
TextCNN	58.45	67.47	60.70	92.68	92.70	92.52	61.29	92.68
SySeVR	42.22	64.50	48.12	92.82	91.93	91.97	58.40	92.82
Proposed	91.21	91.64	90.97	95.22	95.76	95.26	95.22	95.22

The sample size of each category in the SARD database varies greatly, with some containing more than 2000 samples and others containing only a dozen samples. Therefore, it can be seen that the results under the Marco method and the Weight method are very different. However, either way, the method in this paper shows a clear advantage in the multi-classification task.

Due to the unbalanced database samples, only calculating the accuracy of the full sample set sometimes indicates not only the goodness of the model but also the result category statistical information for comparative analysis. For example, if the negative samples in a binary sample set are only 5%, then the overall accuracy of my test results can reach 95% even if all samples are identified as positive. The confusion matrix is to put the predicted results of all categories and the real results into the same graph according to categories, as shown in Figure 7. The number of correct identification and the number of incorrect identifications of each category can be clearly seen in the figure.

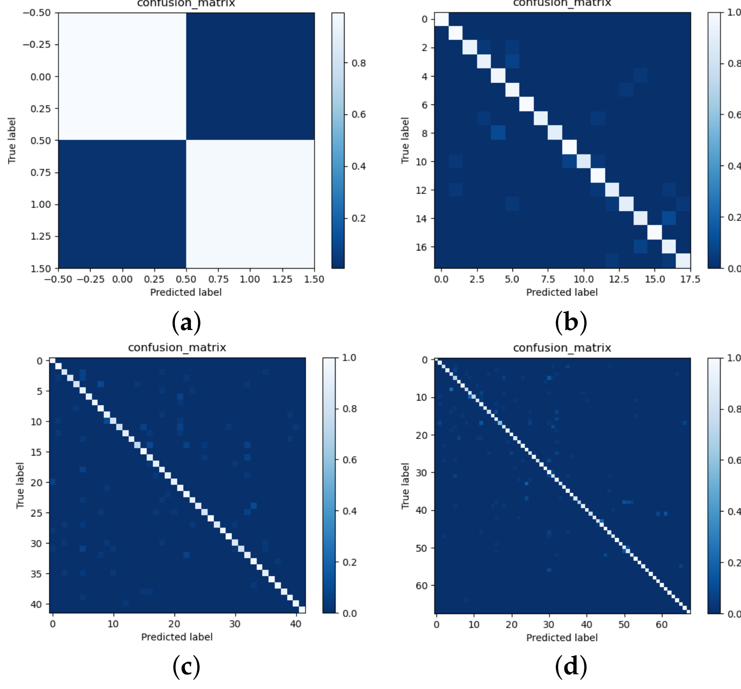

Figure 7. Confusion matrices of 2 (**a**), 18 (**b**), 42 (**c**) and 68 (**d**) classification task obtained via the proposed method for different classification tasks.

5.2.3. Review Comment Output

Purposefully, the review comments output by the proposed framework do not give the user a modification order but serve as a more intuitive reference opinion for the user to modify the code. Therefore, we output several review comments based on the top N similarities between the input code and the defective code in the database, as shown in Figure 8.

FILES	CWEx	REVIEW-1	REVIEW-2
File1	CWE253	FLAW: Call RegOpenKeyExA() with HKEY_LOCAL_MACHINE violating the least privilege principal FIX: Call RegOpenKeyExA() with HKEY_CURRENT_USER	FLAW: fputws() might fail, in which case the return value will be WEOF (-1), but we are checking to see if the return value is 0 FIX: check for the correct return value
File2	CWE23	FLAW: Possibly opening a file without validating the file name or path FIX: File name does not contain a period or slash	__NULL__
…	…	…	
File3	CWE680	FLAW: Read data fro fgets() FIX: Set data to a rela greater than zero	

```
... ...
{
    size_t data_len = strlen(data);
    FILE * pFile;
    if(FILENAME_MAX-data_len > 1)
    {
        pFile = FOPEN("C:\\temp\\file.txt", "r");
        if (pFile != NULL)
        {
            fgets(data+data_len, (int)(FILENAME_MAX-data_len), pFile);
            fclose(pFile);
        }
    }
}
... ...
```

Figure 8. Defect-related review comment output.

In this case, users can revise their code directly based on the review comments. In this experiment, we use $N = 2$ for similarity threshold $\alpha > 0.65$. Therefore, In the second row of the figure, there is only one comment shown. Unfortunately, we have not yet found a suitable evaluation metric to quantitatively assess the similarity of our output to users' real needs. Therefore, we chose to use the same method as the SRAD dataset to give the introductions of defects, in the form of detailed descriptions of defect types and caveats.

5.2.4. Analysis and Discussion

As mentioned briefly above, the procedure of code review is committed to solving several problems. This section analyzes the strengths and weaknesses of the proposed framework in terms of illustrating how well it addresses these issues.

Q1: Is the method able to detect code defects?

Since our framework, more specifically, the proposed twin LSTM model, is based on supervised learning in a closed-set training dataset, if an unknown defect occurs in the software, our method may fail. Fortunately, we can follow the CWE (Common Weakness Enumeration) [33] criteria to classify defects. The CWE database is a community-developed project that provides a catalog of common vulnerabilities in the software and hardware of an organization's technology stack. The database includes detailed descriptions of common defects and guides secure coding standards. Since it is a constantly self-updating database, when a new defect type appears, it assigns a unique CWE-ID to that defect and includes it. Therefore, we can roughly assume that if we use the latest CWE database, the training dataset of the model contains all currently known defect types.

From the experimental results in Sections 5.2.1 and 5.2.2, more than 98.6% (from the indicator R) defects can be found, and more than 98.8% (from the indicator P) of which can be detected correctly for the binary classification task. Additionally, the results on the multi classification tasks are more than 91.2% and 91.6%, respectively.

Q2: Can the method locate the detected defect?

From coarse to fine, defect location can be classified into segment-level (block-level), slice-level, and statement-level. From the current study [22,23], the results of the detection accuracy at the statement-level were relatively low and far from the maturity level that can be applied.

Benefiting from the proposed code slicing method D2U flow, we can obtain the shortest slices from preprocessor because the D2U flow divides the data stream of the same variable into different control-branches (e.g., controlled by the keywords *if* and *switch* in C/C++ language) into different slices. Although additional computational overhead is required, the effect is negligible for the offline training process. From this perspective, our slice-level approach gives the least candidates of defective statements, i.e., the most fine-grained.

Q3: Is the method able to distinguish the specific type of detected defects?

Comparative experimental results of the proposed Deep Reviewer and other methods for multiclass detection have been presented in Section 5.2.2. Each output result associates a CWE-ID (or 0 if the input is non-defective). Therefore, the method in this paper can provide the type of defect at the time of detection.

However, it is worth stating that the latest version of the SARD database contains 118 defect categories (CWE-IDs). The experiments in Section 5.2.2 show that we have only selected at most 67 of them for the comparative test. This is because the sample size of each category in this database varies greatly, from a maximum of more than two thousand entries to a minimum of less than ten. This causes these small-sample categories to not only be easily overfitted during training but also have a negligible impact on the overall results when evaluating the framework. Therefore, we selected 67 categories with a sample size greater than 100, and at the same time, we used three different calculation methods (Macro Average and Weight Average, which have been introduced in Section 5.1) for the evaluation indicators P, R, and $F1$. Aiming at an objective and fair comparison test.

Q4: Can the method provide modification suggestions for defects?

From a practical point, our framework resembles a dictionary structure. In this dictionary structure, when an entry, i.e., a defect type indexed by CWE-ID, is found, the framework can further give a detailed introduction of it or suggestions for fixing it.

Initially, we designed a simpler detector structure with not $N + 1$ but $N' + 1$ nodes in the full linkage layer, where N and N' denote the number of defect types and the number of training samples, respectively. In other words, we have tried to widen the full linkage layer so that a single deep network can be used to find the training sample that is most similar to the input. In this way, the output of this training sample CWE-ID can be accompanied by the output of pre-edited review comments. However, the experimental results have shown that this simple and crude way cannot achieve satisfactory accuracy.

Therefore, we use the first depth model (LSTM-1 in Figure 1) to detect the CWE-ID and delimit the range of candidates of the review comment. The second depth model (LSTM-2 in Figure 1) is then used to retrieve the target among the candidates that is most similar to the input sample and outputs its pre-edited review comments.

Q5: What are the limitations of this framework?

Although the superiority of the proposed framework in terms of accuracy is demonstrated by the experiments above, the method can also output relevant reviews comments. However, we are aware of some shortcomings:

1. Although the proposed framework innovatively outputs defect categories and review comments jointly, we are unable to quantitatively evaluate the compliance of the output review comments, which somewhat reduces the objectivity of the conclusions.
2. As the backbone, LSTM is a relatively outdated neural network model. Although our selection is based on experiments compared with other state-of-the-art methods, there may be a better model as the backbone of the detector.
3. Although the proposed approach is theoretically applicable to other programming languages, we have not been able to conduct relevant experiments to verify it due to the lack of training data. This is the next step in our research direction: adapting the preprocessor to achieve support for different programming languages.

6. Conclusions

In this paper, we proposed a flexible and generic framework, Deep Reviewer, for high-precision code defect detection. In the preprocessor module, a novel code slicing method D2U flow was proposed. It extracts the life cycle of a variable that is complete and free of running conflicts from definition to usage. In the detector module, we applied a twin LSTM model to extract and match features. We call this framework flexible because all of methods we use in the paper can be replaced by others, including the code slicing method D2U flow, feature vectorization method Word2Vec, and the deep neural network LSTM. To help reviewers and coders quickly locate and fix defects in the code, the most significant advantage of this framework is that it can highly accurately identify multiple types of defect types while associating the corresponding review comments. The comparative study has demonstrated the effectiveness of the proposed method over other state-of-the-art methods. We will work on evaluating the practical effects of this approach in real engineering code, especially in airborne software, in future work.

Author Contributions: Conceptualization, X.C., H.-C.L. and X.-P.Y.; Methodology, X.C. and H.-C.L.; Software, H.-C.L.; Validation, L.D., H.-C.L. and S.Y.; Investigation, P.W.; Resources, H.-C.L. and X.-P.Y.; Data curation, H.-C.L.; Writing—original draft, X.C., L.D., X.-P.Y. and P.W.; Writing—review and editing, X.C., H.-C.L. and S.Y.; Visualization, H.-C.L.; Supervision, L.D. and P.W.; Project administration, P.W.; Funding acquisition, X.C. All authors have read and agreed to the published version of the manuscript.

Funding: This work was supported in part by the Tianjin Education Commission Scientific Research Plan Project under Grant 2022KJ058, in part by the Fundamental Research Funds for the Central Universities under Grant 3122022QD07, and in part by the Key Laboratory of Civil Aviation Intelligent Flight Open Subject under Grant B2022JS32055.

Data Availability Statement: Publicly available datasets were analyzed in this study. This data can be found here: [www.nist.gov/itl/ssd/software-quality-group/samate/software-assurance-reference-dataset-sard, accessed on 5 May 2022].

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Lions, J.L.; Lennart Lübeck, L. *ARIANE 5 Failure-Full Report*; European Space Agency: Paris, France , 1996.
2. Sadowski, C.; Söderberg, E.; Church, L.; Sipko, M.; Bacchelli, A. Modern Code Review: A Case Study at Google. In Proceedings of the IEEE/ACM 40th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP), Gothenburg, Sweden, 27 May–3 June 2018; pp. 181–190.
3. Heckman, S.; Williams, L. A systematic literature review of actionable alert identification techniques for automated static code analysis. *Inf. Softw. Technol.* **2011**, *53*, 363–387. [CrossRef]
4. Zhioua, Z.; Short, S.; Roudier, Y. Static code analysis for software security verification: Problems and approaches. In Proceedings of the 38th International Computer Software and Applications Conference Workshops, Vasteras, Sweden, 21–25 July 2014; pp. 102–109.
5. Muske, T.; Serebrenik, A. Survey of Approaches for Postprocessing of Static Analysis Alarms. *ACM Comput. Surv. (CSUR)* **2022**, *55*, 1–39. [CrossRef]
6. FlawFinder. 2018. Available online: www.dwheeler.com/flawfinder (accessed on 5 May 2022).
7. Li, Z.; Zou, D.; Xu, S.; Ou, X.; Jin, H.; Wang, S.; Deng, Z.; Zhong, Y. Vuldeepecker: A deep learning-based system for vulnerability detection. *arXiv* **2018**, arXiv:1801.01681.
8. Zou, D.; Wang, S.; Xu, S.; Li, Z.; Jin, H. μVulDeePecker: A Deep Learning-Based System for Multiclass Vulnerability Detection. *IEEE Trans. Dependable Secur. Comput.* **2021**, *18*, 2224–2236. [CrossRef]
9. Li, Z.; Zou, D.; Xu, S.; Jin, H.; Zhu, Y.; Chen, Z. SySeVR: A Framework for Using Deep Learning to Detect Software Vulnerabilities. *IEEE Trans. Dependable Secur. Comput.* **2022**, *19*, 2244–2258. [CrossRef]
10. Liu, S.; Lin, G.; Han, Q.L.; Wen, S.; Zhang, J.; Xiang, Y. DeepBalance: Deep-Learning and Fuzzy Oversampling for Vulnerability Detection. *IEEE Trans. Fuzzy Syst.* **2020**, *28*, 1329–1343. [CrossRef]
11. Rough Audit Tool for Security. 2014. Available online: code.google.com/archive/p/rough-auditing-tool-for-security (accessed on 5 May 2022).
12. Checkmarx. 2014. Available online: www.checkmarx.com/ (accessed on 5 May 2022).
13. Yamaguchi, F. Pattern-based methods for vulnerability discovery. *It-Inf. Technol.* **2017**, *59*, 101–106. [CrossRef]

14. Lin, G.; Xiao, W.; Zhang, J.; Xiang, Y. Deep learning-based vulnerable function detection: A benchmark. In *Proceedings of the International Conference on Information and Communications Security, Beijing, China, 15–17 December 2019*; Springer: Berlin/Heidelberg, Germany, 2019; pp. 219–232.
15. Wu, F.; Wang, J.; Liu, J.; Wang, W. Vulnerability detection with deep learning. In Proceedings of the 3rd International Conference on Computer and Communications (ICCC), Chengdu, China, 13–16 December 2017; pp. 1298–1302. [CrossRef]
16. Fidalgo, A.; Medeiros, I.; Antunes, P.; Neves, N. Towards a Deep Learning Model for Vulnerability Detection on Web Application Variants. In Proceedings of the International Conference on Software Testing, Verification and Validation Workshops (ICSTW), Porto, Portugal, 24–28 October 2020; pp. 465–476. [CrossRef]
17. Wartschinski, L.; Noller, Y.; Vogel, T.; Kehrer, T.; Grunske, L. VUDENC: Vulnerability Detection with Deep Learning on a Natural Codebase for Python. *Inf. Softw. Technol.* **2022**, *144*, 106809. [CrossRef]
18. Cao, S.; Sun, X.; Bo, L.; Wei, Y.; Li, B. BGNN4VD: Constructing Bidirectional Graph Neural-Network for Vulnerability Detection. *Inf. Softw. Technol.* **2021**, *136*, 106576. [CrossRef]
19. Grieco, G.; Grinblat, G.L.; Uzal, L.; Rawat, S.; Feist, J.; Mounier, L. Toward large-scale vulnerability discovery using machine learning. In Proceedings of the 6th ACM Conference on Data and Application Security and Privacy, New Orleans, LA, USA, 9–11 March 2016; pp. 85–96.
20. Moshtari, S.; Sami, A. Evaluating and comparing complexity, coupling and a new proposed set of coupling metrics in cross-project vulnerability prediction. In Proceedings of the 31st Annual ACM Symposium on Applied Computing, Pisa, Italy, 4–8 April 2016; pp. 1415–1421.
21. Yamaguchi, F.; Lottmann, M.; Rieck, K. Generalized vulnerability extrapolation using abstract syntax trees. In Proceedings of the 28th Annual Computer Security Applications Conference, Orlando, FL, USA, 3–7 December 2012; pp. 359–368.
22. Hin, D.; Kan, A.; Chen, H.; Babar, M.A. LineVD: Statement-level Vulnerability Detection using Graph Neural Networks. *arXiv* **2022**, arXiv:2203.05181.
23. Li, Y.; Wang, S.; Nguyen, T.N. Vulnerability detection with fine-grained interpretations. In Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Athens, Greece, 23–28 August 2021; pp. 292–303.
24. Wang, X.; Guan, Z.; Xin, W.; Wang, J. Source code defect detection using deep convolutional neural networks. *J. Tsinghua Univ. Technol.* **2021**, *61*, 1267–1272.
25. Noonan, R.E. An algorithm for generating abstract syntax trees. *Comput. Lang.* **1985**, *10*, 225–236. [CrossRef]
26. Ferrante, J.; Ottenstein, K.J.; Warren, J.D. The program dependence graph and its use in optimization. *ACM Trans. Program. Lang. Syst. (TOPLAS)* **1987**, *9*, 319–349. [CrossRef]
27. NIST. SARD Database. 2020. Available online: www.nist.gov/itl/ssd/software-quality-group/samate/software-assurance-reference-dataset-sard (accessed on 5 May 2022).
28. Bayer, U.; Moser, A.; Kruegel, C.; Kirda, E. Dynamic analysis of malicious code. *J. Comput. Virol.* **2006**, *2*, 67–77. [CrossRef]
29. Ball, T. The concept of dynamic analysis. *ACM SIGSOFT Softw. Eng. Notes* **1999**, *24*, 216–234. [CrossRef]
30. Schütte, J.; Fedler, R.; Titze, D. Condroid: Targeted dynamic analysis of android applications. In Proceedings of the 29th International Conference on Advanced Information Networking and Applications, Gwangju, Republic of Korea, 24–27 March 2015; pp. 571–578.
31. Muhammad, P.F.; Kusumaningrum, R.; Wibowo, A. Sentiment Analysis Using Word2vec And Long Short-Term Memory (LSTM) For Indonesian Hotel Reviews. *Procedia Comput. Sci.*f **2021**, *179*, 728–735. [CrossRef]
32. Adipradana, R.; Nayoga, B.P.; Suryadi, R.; Suhartono, D. Hoax analyzer for Indonesian news using RNNs with fasttext and glove embeddings. *Bull. Electr. Eng. Inform.* **2021**, *10*, 2130–2136. [CrossRef]
33. CWE. 2023. Available online: http://cwe.mitre.org/ (accessed on 5 May 2022).

Disclaimer/Publisher's Note: The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.