Article:
Xu, R, Tang, Z, Ye, G et al. (4 more authors) (2022) Detecting Code Vulnerabilities by Learning from Large-Scale Open Source Repositories. Journal of Information Security and Applications, 69. 103293. ISSN 2214-2126

https://doi.org/10.1016/j.jisa.2022.103293

Detecting Code Vulnerabilities by Learning from Large-Scale Open Source Repositories*

Rongze Xu[a], Zhanyong Tang[a,*], Guixin Ye[a], Huanting Wang[a], Xin Ke[a], Dingyi Fang[a] and Zheng Wang[b]

[a]Northwest University, China
[b]University of Leeds, U. K

ARTICLE INFO

Keywords:
Code Vulnerability Detection
Deep Learning
Attention Mechanism
Software Vulnerability

ABSTRACT

Machine learning methods are widely used to identify common, repeatedly occurring bugs and code vulnerabilities. The performance of a machine-learned model is bounded by the quality and quantity of training data and the model's capability in extracting and capturing the essential information of the problem domain. Unfortunately, there is a storage of high-quality samples for training code vulnerability detection models, and existing machine learning methods are inadequate in capturing code vulnerability patterns.

We present DEVELOPER[1], a novel learning framework for building code vulnerability detection models. To address the data scarcity challenge, DEVELOPER automatically gathers training samples from open-source projects and applies constraints rules to the collected data to filter out noisy data to improve the quality of the collected samples. The collected data provides many real-world vulnerable code training samples to complement the samples available in standard vulnerable databases. To build an effective code vulnerability detection model, DEVELOPER employs a convolutional neural network architecture with attention mechanisms to extract code representation from the program abstract syntax tree. The extracted program representation is then fed to a downstream network - a bidirectional long-short term memory architecture - to predict if the target code contains a vulnerability or not. We apply DEVELOPER to identify vulnerabilities at the program source-code level. Our evaluation shows that DEVELOPER outperforms state-of-the-art methods by uncovering more vulnerabilities with a lower false-positive rate.

1. Introduction

Machine learning is an established technique for building predictive models to support code-related tasks like program similarity assessment [27] and code vulnerability detection [27]. By automatically learning the latent patterns indicative of vulnerable code, machine-learned models can exceed expert-crafted rules [30] and reduce development time for tools [29].

While promising, there are two significant drawbacks that limit the uptake of machine learning for software vulnerability detection: the lack of training samples and the insufficient model capability in reasoning about program semantics. Most machine-learning-based code vulnerability detection methods rely on training data from standard vulnerability databases like the Software Assurance Reference Dataset (SARD) or the national vulnerability database (NVD) [28, 31, 58, 29]. These datasets, however, cannot represent the diversity of real-world vulnerabilities and lag behind the software evaluation. The lack of training data negatively affect the quality of ML models, as they have very sparse training data for typical high-dimensional program space. Some methods of program synthesis may be able to alleviate the above dilemma[4]. However, synthetic programs are biased by the grammars, templates, or models used to generate the programs, and they may not reflect the diverse and evolving patterns of real-life programs. Therefore, machine learning models learned over synthetic data are hard to generalize to real-world code. To address the lack of training samples, we need to collect a large number of high-quality, real-world vulnerable code samples.

Insufficient reasoning about program semantics also hinders bug patterns learning for models. Existing approaches[42, 53] mostly rely on treating the source code as text or surface syntactic information like abstract syntactic tree (AST), which cannot catch well-defined structure information like control and data flow in programs. For example, LI *et al.* [29] based on data flow dependence and without control flow dependence, which means they cannot clarify execute path. Thus, flow sensitive vulnerability types like `Use After Free` cannot be classified correctly due to the fact that the model would mistakenly hold a view that one variable is released twice.

Furthermore, code embedding technique is also an important factor in the performance of training model. Existing methods [42, 53] process code snippets into flat sequences and then use WORD2VEC [37] to encode text sequences. These words correspond to vectors one-to-one so that vectors could represent code snippets. Unfortunately, this approach cannot capture the well-structured relationships and semantic information in programs, which will lead to the poor performance of bug detection. In addition to

*This work was supported in part by the National Natural Science Foun- dation of China (NSFC) under grant agreements 61972314 and 61872294, CCF-Huawei Populus Grove Fund, and the International Cooperation Projects of Shaanxi Province under grant agreements 2020KWZ-013 and 2021KW-04.

*Corresponding author
zytang@nwu.edu.cn (Z. Tang)
ORCID(s):

WORD2VEC, which treats code as text embedding, an embedding method similar to CODE2VEC [1] learns code semantics through distributed representation. But CODE2VEC shows in follow-up research [20] that embedding approaches cannot be easily used in downstream models, and the continuous representation of code tokens may not perform better than baselines which treat code tokens simply as symbols.

To solve these two problems, in this paper, we present DEVELOPER, a novel learning framework for building code vulnerability detection models. To address the data scarcity challenge, we gather real-world vulnerabilities from GitHub, the world's largest code hosting repository. GitHub contains many projects and a large number of programmers with a short review cycle and code edition iteration, which means that GitHub contains more vulnerabilities, faster updates, and a wider variety of vulnerabilities from the real world compared to SARD (refer to as Standard). We can get abundant code relationship information of vulnerabilities. Furthermore, detection models trained by the real-world dataset can better capture the semantic information and vulnerable patterns from the source code due to their various coding styles for the same function.

Another challenge is how to ensure the high quality of vulnerable examples collected from large-scale open source repositories. Some prior works simply use filter module [57, 2, 47, 15, 19] with keywords or white word lists. Nevertheless, these approaches usually collect vulnerability commits with more than one vulnerability type in one commit. To solve this problem, we perform syntactic analysis [19] based on RE (regular expression) rules filtering. By extracting the summary of commits message to obtain sentences' grammatical structure and dependency relationships between words. By doing so, we can ensure that the training data set will exclude Mixed-type commits [40] and belong to the specific vulnerability type. Then, we use these commits to obtain the security-relevant source code.

As mentioned above, the diversity and quality of vulnerable code samples during the training phase will impact the detection model results. Furthermore, code slicing and code embedding methods are beneficial to models to understand vulnerabilities' patterns and further reduce the false-positive rate of bug detection. To avoid the problem that existing approaches poorly capture the actual running path programs and semantic information (for example, the detection model mistakenly thought that statements in various control branches were executed in order). Our approach is based on data flow and control flow dependence to slice code on function level, which means we could improve the precision of bug location to execution path level. To better extract information on code snippets, inspired by CODE2VEC [1], we combine the AST path (All AST nodes between two AST terminal nodes.) and attention mechanism. Based on the AST path, code snippets are transformed into several AST node sequences, including structural information. Then, we convert sequences of nodes into numerical values and feed into the network which includes a convolutional layer and a fully connected layer, to learn the context representation

```
1  ...
2  reversedString[i] = '\0';
3  free(reversedString);
4  return reversedString;
5  ...
```

(a) Vulnerability sample from Standard

```
1  ...
2  if (errno == ENOENT) {
3      free(dir);
4      return ESP_ERR_NVS_NOT_FOUND;
5  }
6  wlerr("ERROR: Failed to get open %s\n", dir);
7  ...
```

(b) Fixed vulnerability sample from GitHub

Figure 1: Code (a) in Standard is an `Use After Free` vulnerability sample, and code (b) in GitHub is a sample which fixed the `use after free` vulnerability. Code sample (b) from real world contain more branches and control flow and data flow relationships than code (a).

of the path. To deal with the problem of learning one code snippet representation from different AST paths. DEVELOPER increases vulnerability weight based on the attention mechanism, combines different paths' features, and highlights important paths during vulnerability detection. After these steps, DEVELOPER can represent code snippets in a precise way.

We show advantages of DEVELOPER by applying it to detect source-code vulnerabilities. We thoroughly evaluate DEVELOPER on vulnerable programs from open source repositories written in C and Java. Comparing with two state-of-the-art (SOTA) vulnerability collection models [57, 51], two code embedding schemes [12, 1] and five SOTA vulnerability detection methods [30, 29, 52, 22, 8]. Experimental results show that DEVELOPER is superior to other competitive methods by discovering more code vulnerabilities with a lower false-positive rate as shown in Sec. 9.

In summary, our work has made the following contributions:

- We exploit a data collection method to collect high-quality vulnerabilities from open source code repositories. Extending Standard training dataset with more real-world vulnerabilities allows the model to detect more vulnerabilities in real-world scenarios.
- We introduced a code preprocessing method with attention mechanism, including a slicing and a embedding module. It means the semantic gap can be narrowed.
- We design and implement DEVELOPER, a system for source code vulnerability detection. Experiments show that DEVELOPER performs the best overall performance than other competing approaches.

2. Motivation

Deep learning (DL) techniques are proven to be effective for modeling the program structures and semantics [55, 48]. Recently, researchers have proposed a number of DL-based

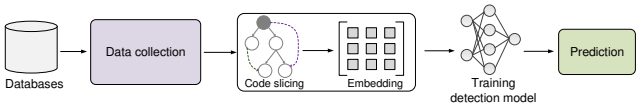


Figure 2: Overview of DEVELOPER. Our detection model takes code snippets from the Standard and GitHub on the path level as input.

models to detect software vulnerabilities [29, 26, 30, 56]. While promising, DL-based models are sensitive to the training samples, making existing methods that are trained by using samples from SARD and NVD suffering from lower accuracy when targeting the real-world software (e.g., programs from GitHub).

In this section, we give a motivating example shown in Figure 1, including two code snippets: one is an `use-after-free` vulnerable code snippet in Figure 1(a) that comes from SARD, the other is a benign code snippet in Figure 1(b) comes form GitHub.

In Figure 1(a), the variable `reversedString` at line 3 is freed before being returned at line 4, leading to an `use-after-free` vulnerability. Unlike this case, the variable `dir` in Figure 1(b) at line 3 is conditionally freed. This `use-after-free` vulnerability is not liable to be triggered when executing the statement at line 6.

However, poor code representation and slicing method make existing detection models give a wrong classification result when facing real-world programs like [10]. For example, WU *et al.* [53](treats the code as text) and VULDEEPECKER [29] (using code's data flow information) regard both example as `use-after-free` vulnerability, because the bug pattern they learned from Standard data set cannot fully handle complicated code relationships in GitHub.

Compared with Standard code samples, codes from real world contain more control and data flow information. Most of the code samples from Standard are similar to the sample in Figure 1(a), but in GitHub, the complexity of real-world programs is higher than Standard, (Sec. 9.1 can demonstrate our findings). It means that existing vulnerability detection models based on Standard [30, 29, 28] cannot identify vulnerabilities even with the same vulnerable type in the real-word due to the fact that Standard training samples do not have enough code relationship information.

Thus, DEVELOPER extends vulnerabilities from GitHub and exploits a useful code representation approach to capture enough code relationship information to reduce the false-positive rate of vulnerability detection models.

3. Overview of our approach

Figure 2 shows the high-level overview of DEVELOPER, including the following parts:

Data collection. The dominant open source platform, such as GitHub, helps us collect high-quality data for the training model. (Sec. 4)

Code slicing. Generating the code snippets by data flow dependence and control flow dependence. (Sec. 5)

Embedding. The embedding method uses the AST path and attention mechanism with vulnerability weight to represent code snippets. (Sec. 6)

Training detection model. Training vulnerability detection models through deep neural network BiLSTM with Highway network. (Sec. 7)

Prediction. Feeding code snippets into the detection model and finishing the vulnerability detection.

The rest of the paper is organized as follows. In Sec. 4 to 7, we describe our vulnerability detection model design, which respectively corresponds to different stages, among which, Sec. 4 describes the data collection, Sec. 5 describes the code slicing, Sec. 6 describes the data embedding, and Sec. 7 describes the training network. Sec. 8 and Sec. 9 describe our experimental evaluation of DEVELOPER and results. Sec. 10 illustrates the limitation of our work and future work. Sec. 11 describes the related prior work. Sec. 12 concludes this paper.

Table 1
Low-quality commits message in GitHub.

Commit-type	Type description	Example
How-type commit	Sufficient to indicate the intention to change.	Add a new line in 736, delete line in 690.
ID-type commit	Without description, only the commit ID.	Fixed-13f79535-47bb-ffa450edef68
Mixed-type commit	Containing more than one reasons for the modification, and it is impossible to intuitively see which part of the code was modified for what reason.	Fix compile error and fix stackoverflow and fix NPE.

4. Data collection

To collect high-quality vulnerability data in a real development environment, DEVELOPER uses commit messages from GitHub and constructs a list of RE rules. In this section, we introduce how to collect high-quality vulnerabilities from GitHub.

4.1. Data acquisition

Commits are changes to a file (or set of files). Commits usually contain a commit message, which is a brief description of what changes were made. We define a bug-fix file snapshot *Pair(bug, fixed)* as a code paired group of before and after fixed bugs. The difference between pre-bug-fix code and post-bug-fix code *Diff(bug, fixed)* is treated as a diff block.

For example, a commit with "`fix a bug in Line 456`" as the commit message, Bug-file is a file that has vulnerabilities in Line 456 before modification, and Fix-file is a file that does not contain vulnerabilities in Line 456 after modification. However, the application scenarios are far more complex than the example listed above; many commits contain imprecise information. The following three low-quality commit types shown in Table 1 lead a poor performance [40]:

- **How-type commits.** This type of commits does not indicate the change reason. They only show how and where

the change location;

- **Mixed-type commits.** This type of commits contains multiple modifications, which make models impossible to know which modification corresponds to which the changed lines of code;
- **ID-type commits.** This type of commits has no description of the change code but the ID.

At present, many researches use historical modification information on GitHub to collect data [40, 57, 2, 47, 15, 19]. They all use the GitHub API directly to filter the commit messages by keywords. However, the results obtained by this method alone are imprecise, not only with many low-quality commits, but also, keywords are distributed in different sentences, which results in no semantic relationship between these individual keywords. To collect high-quality commits, we first choose to collect commits from the highest-ranked repository.

Currently, most of the work related to collecting data from GitHub is sorted by *Star* [11]. In the experiment, the quality of the screened repositories using *Fork* as the sorting rule is much higher than screened repositories by *Star*. The reason is that when users *Fork* the repository, they are most likely to participate in code editing, and the *Star* repository is often used to show appreciation, so we sort the GitHub repositories by *Fork* and select the top 7% of the repositories to get commits. We need to clarify that our approach is designed to process commits where the bug is fixed in a single commit. This scenario commonly exists in GitHub. For example, we review 1,000 randomly selected code commits and found that 81.2% of vulnerabilities were fixed in one single commit.

4.2. Initial data filtering

based on keywords

Some RE rules and keywords are shown in Table 2 helps to avoid a low *Precision*. Specific steps are as follows:

(1) To filter commits that do not match keywords (mainly filter *How-type commits* and *ID-type commits*). DEVELOPER build RE rules by GitHub APIs. Refer to "*2020 CWE Top 25 Most Dangerous Software Errors*" in CWE [7], and select the five most common vulnerabilities in C/C++ (CWE-119, CWE-399, CWE-401, CWE-415, CWE-416) and JAVA (CWE-020, CWE-022, CWE-129, CWE-400, CWE-476). Then, we summarize a list of corresponding keywords and RE rules according to the vulnerability in Table 2) to filter unrelated commits.

(2) To filter commits that match keywords but are not related to vulnerabilities. To illustrate the necessity of this step, please refer to a commit after filtering through the previous step, `"Fix invalid helps and description of session UUID verification"`. Although `"invalid"` and `"verification"` form the keyword `"invalid verification"`, this commit has nothing to do with invalid verification vulnerabilities. So after using the Re rule constructed with the above keywords to filter irrelevant commits, we use the commit message characteristics to filter further. The first step is to construct RE rules which could delete merged and rolled back commits [19]. Next, constructing other RE rules to delete commits that contain keywords such as "delete, remove, discard, uninstall". Because these commits usually are not within the scope of vulnerability repair. The third step is to delete commits larger than 1 MB. According to our observations, most commits over 1 MB are not bug fixes. Finally, extracting the first sentence of the commit message as a summary of the entire commit message [15], and filter commits using RE rules and keywords. Our purpose is to ensure that the keywords in table 2 are present in the summary of the vulnerable commit message instead of unrelated commits.

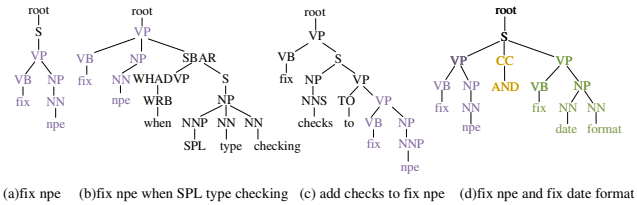
Table 2
Description and keywords of different vulnerability types collected from GitHub.

Bug type	Type description	Key words
CWE-119	This category is related to improper operation restrictions within the memory and buffer area.	buffer overflow/ buffer underflow/ integer overflow/ integer underflow/ buffer under-read/ incorrect buffer access
CWE-399	Vulnerabilities in this category are related to improper management of system resources.	unsafe reflection/ file descriptor leak/ insufficient resource pool/ uncontrolled memory allocation/ improper path traversal
CWE-401	This category is related to memory leaks.	memory leak
CWE-415	This category is called twice with the same parameter as the program Related to the program crash caused by free().	double free
CWE-416	This category is related to the program crash caused by re-referencing the memory after releasing the memory.	use after free/ dangling pointer/ wild pointer/ freed pointer dereference
CWE-020	This category is related to improper input validation.	improper input validation/ invalid verification
CWE-022	This category is related to not properly restricting file paths.	path traversal/ directory traversal
CWE-129	This category is related to unreasonable verification of array indexes.	array index underflow/ array index out of bound/ array index out of bounds exception
CWE-400	This category is related to uncontrolled resource consumption.	uncontrolled resource consumption/ resource leak
CWE-476	This category is related to null pointer references.	null pointer dereference/ null pointer exception/ null pointer/ null check/ npe/ unchecked return value

4.3. Secondary data filtering

based on syntactic analysis

Since *Mixed-type commits(see Table 1)* may submit fixes containing multiple vulnerability types or functions (this means that multiple files and codes have been modified), which will cause the description information in the commits message and the sets of fixed code lines to fail to correspond. Furthermore, Mixed-type commit causes the *bug* and *fixed* in *Pair(bug, fixed)* to fail to map one by one, and the correspondence between this ambiguous bug and fixed will confuse the vulnerability detection model.



(a)fix npe (b)fix npe when SPL type checking (c) add checks to fix npe (d)fix npe and fix date format

Figure 3: The grammatical structure of commit message. (a) (b) (c) have only one verb-object structure like "fix `npe`", which could exclude Mix-type vulnerability; (d) have two verb-object structures, which is a Mix-type commit message.

To filter out *Mixed-type commits*, our approach syntactically analyzes commit messages [19]. The primary task of syntactic analysis is to determine the grammatical structure and the dependence between words in a sentence. DEVELOPER used the natural language processing tool *Stanford CoreNLP* [34] to perform syntactic analysis. Taking the "null pointer dereference"(NPE) vulnerability as an example, we traverse the syntax tree of the commit and find only a verb-object construction subtree of "fix + noun" (fix `npe`) in Figure 3(a), which indicates that the commit has only one modification intention. As shown in Figure 3(b) and (c) both contain such subtrees (marked in purple). However, if the parent node of the subtree is found to have conjunction, and a subtree containing another verb-object construction, then the commit does not meet the requirements. As shown in Figure 3(d), the yellow mark represents the conjunction, and the green subtree represents another verb-object construction. It contains two verb-object construction, we cannot clearly correspond to the modification intention and lines of change code, so it does not meet the requirements.

5. Code slicing

After collecting data, DEVELOPER uses data and control flow dependence to slice *Pair(bug, fixed)*. This slicing method preserves the semantic information of source code [38]. It identifies the vulnerability on a specific execution path instead of the usual function level.

5.1. Parse *Pair(bug,fixed)*

We use commits to obtain the code *Pair(bug,fixed)* and delete unnecessary spaces as well as line breaks in the *Pair(bug,fixed)*, parse the *Pair(bug,fixed)* to obtain accurate information of code for adding and deleting lines. Figure 4 shows a commit that fixes the `null pointer dereference`. The commit deleted code line 5, and added code lines 5 to 8 to repair the vulnerability. Among them, the difference between Bug-file and Fixed-file in the commit is the collection of purple and green code lines in Figure 4, which is *Diff(bug, fixed)*.

5.2. Slicing and labeling

After parsing *Pair*, Figure 5 shows the process of slicing the Bug-file into three snippets (similar as fixed-file). Because line 4 involves an if conditional statement, the code would be divided into two branches, one branch meets the control condition, and the other does not. The code is divided into two snippets according to the number of branches: one is *snippet1*, the condition (line 4) and the statement (line 5) are not executed in this function, then the execution path to the end of the function. Another is *snippet2*, that is, codes of line 4 and line 5 are executed, and then executed until the execution path at the end of the function.

We label snippets containing the deleted code line as negative samples, and the label that does not have the deleted code line as positive samples. According to this method, the negative samples are *snippet1*, *snippet2*, in Figure 6. But during the slicing process, it was found that the positive and negative snippets were not balanced, and too many negative snippets would lead to poor model performance.

Therefore, for each Bug-file, we take all code lines from the beginning of the function to deleted code lines as a set and then generate a positive snippet. DEVELOPER expands the number of positive samples in this way. For example, generate *snippet3* (see Figure 6 (c)) from Bug-file (see Figure 4 (b)), it includes code lines from 1 to 5. Besides, We also remove duplicate samples in our datasets.

5.3. Normalization

When a developer writes a program, there are many different forms of variable names and function definitions that appear in the program according to the specification requirements. To ensure that subtle semantic differences in programs (such as the choice of variable names or the insertion of comments) do not affect the learned model, uniform symbols would replace user-defined variables and functions (user-defined nodes) in this paper. We delete non-ASCII characters and comments in the code because they are not related to the vulnerability. Further, replacing the same variables and functions with "`Var1`", "`Var2`" and "`Func1`", "`Func2`" to distinguish the variables and functions in the same snippets. When multiple variables or functions appear in different snippets, they would be mapped to the same symbol name. Figure7 shows the symbol replacement for *snippet2*. We replace the parameters `msg`, `ex`, `os`, `ignore`, and `handleException` with `Var1`, `Var2`, `Var3`, `Var4` and `Func1`.

6. Embedding

DEVELOPER uses AST and attention mechanism with vulnerability weight to embed code snippets. Many works have proved that AST can help us capture the semantic information of the program (code clone detection [45, 3, 50], code change patterns [35, 36]). We also introduce an attention mechanism with vulnerability weight to solve the problem of learning the embedding of entire snippet from multiple AST paths: integrate the feature information on these different paths to describe the correspondence between the path set and the label. The process of embedding is shown in Figure 8, it consists of two parts, the first is to use AST to extract the AST path, and the second is to use the AST path and the attention mechanism with vulnerability weight to learn the contextual representation of the source code.

6.1. AST path extraction

First, we generate the AST of the source code and then extract the AST path. We use the *Eclipse JDT and CDT* to

```
1.     private void handleException(String msg, Exception e){
2.     try{
3.         os.write(msg.getBytes());
4.         if (ex!= null){
5.   -         os.write(ex.getMessage().getBytes());
5.   +         String msg2 = ex.getMessage();
6.   +         if (msg2 != null){
7.   +             os.write(msg2.getBytes());
8.   +         }
6.  9.      }
7.  10.   }catch(IOException Ignore){}
8.  11.  }
```

(a) Commit

```
1.  private void handleException(String msg, Exception e){
2.  try{
3.      os.write(msg.getBytes());
4.      if (ex!= null){
5.          os.write(ex.getMessage().getBytes());
6.      }
7.  }catch(IOException Ignore){}
8.  }
```

(b) Bug-file

```
1.  private void handleException(String msg, Exception e){
2.  try{
3.      os.write(msg.getBytes());
4.      if (ex!= null){
5.          String msg2 = ex.getMessage();
6.          if (msg2 != null){
7.              os.write(msg2.getBytes());
8.          }
9.      }
10. }catch(IOException Ignore){}
11. }
```

(c) Fixed-file

Figure 4: Figure 4(a) is a commit on GitHub which fixes a NPE (NULL Pointer Dereference) vulnerability. The vulnerability in this commit is extracted to Figure 4(b) and the content of bug fixed is extracted to Figure 4(c).

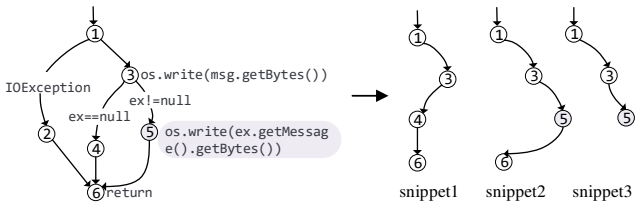


Figure 5: Data and control flow dependence are used to divide Bug-file into two execution paths, *snippet1* and *snippet2*. To extend the number of positive samples, *snippet3* is generated from line 1 to the deleted line.

generate source code AST. AST uniquely represents source code fragments with a given structure and syntax. AST's leaf nodes are called terminal nodes, which usually refer to user-defined values, such as identifiers or names in the source code. Non-leaf nodes are called non-terminal nodes representing some structures with special meaning in the source code, such as loops, expressions, and variable declarations.We extract the sets of AST nodes between two terminal nodes as AST paths. In other words, the AST path is a sequence of terminal nodes and non-terminal nodes. Finally, we learn the contextual representation of the code snippet from all the extracted AST paths. As shown in Figure 9, (a) shows the source code, (b) shows the AST structure, and the four extracted paths. For each path P, it can be represented as a set of node sequences. The AST Sequence 1 represents `Path2` in Figure 9(b).

$$Path2 = \{int, Parameter, method\ Declaration, Parameter, b\} \tag{1}$$

In particular, to prevent the problems that path from being too long and not conducive to the model's training, we selected three types of AST nodes: The first type node is associated with class instantiation and method invocation, uses the function names or class names of these nodes as token in this paper.The second type is the declaration class node, such as method declaration, type declaration, interface declaration, and enumeration declaration. The last type is control dependent node, such as condition control (`IfStatement`), cyclic control (`ForStatement, WhileStatement`), and abnormal control (`ThrowStatement, CatchClause`). For the node's content, we try our best to summarize the content of the node into one word. For example, during parsing of AST, the child node "{}" of the node `BlockStatement`, we regard it as a word.

6.2. The contextual representation of snippets

To learn the context representation vector of the code snippet from many AST paths, we first generate representation vectors of AST paths, add vulnerability weight for certain AST paths, then learn the contextual representation of each AST path. Finally, we use the attention mechanism to fuse multiple AST paths into a one-dimensional vector to represent the entire code snippet. The specific steps are as follows.

6.2.1. The representation vectors of AST path

To convert each AST path into a value that the neural network can input. We map each node in the AST path to a word. Use WORD2VEC to learn on this corpus, where we choose the CBOW model to learn the vector representation of these AST nodes [37]. The CBOW model is designed to predict the conditional probability of the central word w under a given context, such as w_{t-2}, w_{t-1}, w_{t+1}, w_{t+2}, and word vectors are generated by the training process. The model consists of three layers: an input layer, which reads word vectors in the context of word w; a projection layer, which sums vectors of input layer; and an output layer that uses softmax to predict the conditional probability of w. The model first randomly initializes vectors of all words in the corpus, and then these vectors are trained and updated together with the model's parameters. After training the CBOW model, it generates a mapping table of words and their corresponding vectors. Through the model, each node n_i of the path $P = n_1, n_2, ..., n_i$ can be mapped to a word vector $NodeV_i$. The loss function of the model is defined as for formula (2). The loss function of the model is defined as for formula (2). Where $Loss$ is the loss function of nodes in $P = n_1, n_2, ..., n_3$, NNS is a set of neighbor nodes of node n, $NodeV_k$ represents the feature vector of node N_k, $NodeV_j$ represents the feature vector of node N_j, HS$\{NodeV_k \mid NodeV_j\}$ stands for hierarchical softmax function.

$$Loss_i = \min_i \frac{1}{i} \sum_{j=1}^{i} \sum_{k \in NNS^r} -\log HS\{NodeV_k | NodeV_j\} \tag{2}$$

6.2.2. Adding the vulnerability weight

To integrate feature information about vulnerabilities into the representation vectors of the AST path, we refer to the method of LI *et al.*[25] and take the following principle in this paper: if any node n_i in the path $p = \{NodeV_i,$

```
1.  private void handleException(String msg, Exception e){
2.  try{
3.      os.write(msg.getBytes());
4.      if (ex!= null)
7.  }catch(IOException Ignore){}
8.  }
```

(a) *snippet1*

```
1.  private void handleException(String msg, Exception e){
2.  try{
3.      os.write(msg.getBytes());
4.      if (ex!= null){
5.          os.write(ex.getMessage().getBytes());
6.      }
7.  }catch(IOException Ignore){}
8.  }
```

(b) *snippet2*

```
1.  private void handleException(String msg, Exception e){
2.  try{
3.      os.write(msg.getBytes());
4.      if (ex!= null){
5.          os.write(ex.getMessage().getBytes());
```

(c) *snippet3*

Figure 6: Using the three execution paths in Figure 5 to divide the Bug-file into three snippets.

```
1.  private void Func1(String Var1, Exception Var2){
2.  try{
3.      Var3.write(Var1.getBytes());
4.      if (Var2!= null){
5.          Var3.write(Var2.getMessage().getBytes());
6.      }
7.  }catch(IOException Var4){}
8.  }
```

Figure 7: Normalization of the *snippet2* in Figure 6. Uniform symbols would replace user-defined variables and functions in *snippet2*, and delete non-ASCII codes and comments.

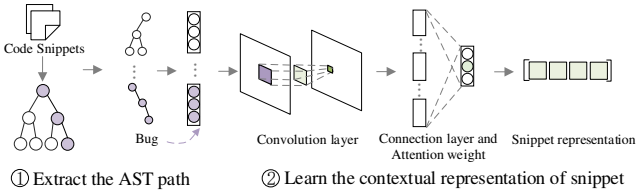


Figure 8: Embedding process. It consists of two steps. First, DEVELOPER extract the AST path, and then use AST and attention mechanism with vulnerability weight

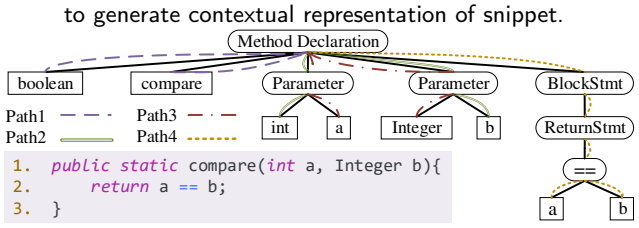


Figure 9: Source code and its four AST paths. AST path refers to the sets of AST nodes between two terminal AST nodes.

$NodeV_{i+1}$,..., $NodeV_n$} contains AST nodes of the vulnerable code line, the same weight w is applied to all AST nodes in the path, namely $p = w * \{NodeV_i, NodeV_{i+1},..., NodeV_n\}$. Conversely, if the path does not contain a vulnerable code line's AST nodes, the path would add no weight. (As shown in Figure 16 (a), only AST paths containing purple nodes are multiplied by w.) Thereby, we can distinguish the vulnerability and non-vulnerability AST path by whether it increases the weight.

6.2.3. The contextual representation of AST path

We extract the AST path features from a series of AST nodes by a convolutional layer. Specifically, the sequence of AST node vectors is represented as a matrix D of size $n * d$. n is the number of nodes, and d is the word vector's length mapped by each AST node. The core of the convolution is the filter. By linearly transforming the local information in different spatial ranges, it can mine the important features of different AST nodes in the path. Specifically, the local receptive fields and weights of the filter share the local space input vector. The information is convolved to extract features. The calculation of the fully connected layer is formula (4), where $\hat{c}_i$ is the output of the fully connected layer, W refers to the weight matrix, and the *tanh* function is a commonly used nonlinear activation function, its output range is (-1,1), each element of the vector would apply it. DEVELOPER takes such feature extraction operations for each path. Updating the weight parameters of the convolutional layer and the fully connected layer by training the network. For every AST path, the model focuses on different nodes.

$$c_i = \sigma(W \cdot x_{i:i+n-1} + b) \tag{3}$$

$$\hat{c}_i = \tanh(W \cdot c_i) \tag{4}$$

6.2.4. The contextual representation of snippets

We use the attention mechanism to emphasize certain AST paths with important vulnerability information. Specifically, we take the following inputs: a training target vector (T), input from fully connection layer after convolutional layer, namely V_i^I, and output from fully connection layer after convolutional layer, namely V_i^O . We define a target vector to have the same length as the input vector. Also, we set all values in the T as 1 for buggy and 0 for non-buggy. V_i^I and V_i^O are both obtained in the previous step in Section 6.2.3. Then we conduct a dot product between the T and the V_i^I , and scale the product result. We apply a softmax function on the result of the scaling process, so we get the attention weight α_i of each V_i^O (such as formula (5)). Finally, the calculation of the context vector of snippet v (as formula (6)) through V_i^O and α_i .

In natural language processing, the attention mechanism can make the model focus on some important words in the sentence. Similarly, in vulnerability detection, it is expected to help the vulnerability detection model give more weight to certain paths that contain important vulnerability information.

$$\alpha_i = \frac{\exp(V_i^I \cdot T)}{\sum_{j=1}^{n} \exp(V_i^I \cdot T)} \tag{5}$$

$$v = \sum_{i=1}^{n} \alpha_i \cdot V_i^O \tag{6}$$

7. Training detection model

7.1. Highway BiLSTM Neural Network

DEVELOPER uses a Highway BiLSTM to train the detection model. Compared with other neural networks such

as RNN and LSTM, BiLSTM has the main advantage of adding a two-way feedback mechanism. It has two LSTM hidden layers, forward and backward. The input vector of the forward hidden layer is from front to back. We use the forward information to detect the backward information to capture the contextual relationship. The input vector of the backward hidden layer is from back to front. We use the backward information to detect the forward information to capture the contextual relationship from another perspective. Finally, both hidden layers are connected to the same output layer. The LSTM hidden layer is composed of many LSTM units. Therefore, BiLSTM can better capture contextual information in both historical and future directions.

Furthermore, inspired by previous work in natural language processing [43, 23] that applies highway gates [46] to help the information propagation between cells, we also employ highway gates to our BiLSTM: With propagation depth M which means M sub layers in a single Highway LSTM layer. the outpur of sub layer m by s_m^t. y^t is read from the last sub-layer and fed into the first layer by letting: $s_0^t = y^{t-1}$. The layer is defined for $m = 1, ..., M$ by:

$$s_m^t = h_m^t \cdot T_m^t + s_{m-1}^t \cdot C_m^t \tag{7}$$

$$h_m^t = \tanh(W_H x^t \delta_{1,m} + R_{H_m} s_{m-1}^t + b_{H_m}) \tag{8}$$

$$T_m^t = \sigma(W_T x^t \delta_{1,m} + R_{T_m} s_{m-1}^t + b_{T_m}) \tag{9}$$

$$C_m^t = \sigma(W_C x^t \delta_{1,m} + R_{C_m} s_{m-1}^t + b_{C_m}) \tag{10}$$

$\delta_{i,j}$ is the Kronecker delta, and W, b are the matrix and bias terms that parametrize the connection. Thus the output of the first sub-layer, s_1^t is obtained from current input x^t, and the previous output, y^{t-1} and the rest of the computation is done in a feed-forward manner with internal Highway skip connections. This formulation shares an arbitrarily deep mapping for the layers' transition and transfer functions.

7.2. Building detection model

Highway BiLSTM helps us automatically learn vulnerability features, solve existing work shortcomings requiring a manual definition of vulnerability features. Figure 10 highlights the structure of the BiLSTM neural network, which includes a BiLSTM layer, a Highway gate layer, a dense layer, and a softmax layer. The BiLSTM layer consists of several BiLSTM unit, mainly used to capture the source code's data-dependent features and structural features. The Highway network layer can effectively alleviate the vanishing and exploding gradient problems during the training network, enable the model to capture much more semantic information [33]. The dense layer, the fully connected layer, aims to map the features learned in the previously hidden layers to the sample's label space and integrate the previously highly abstracted features. The Sigmoid layer is the activation layer. For deep neural networks, the output of the intermediate hidden layer must have an activation function.

For the target file to be detected, we follow Sec.5 to extract many function fragments in the file and generate sets

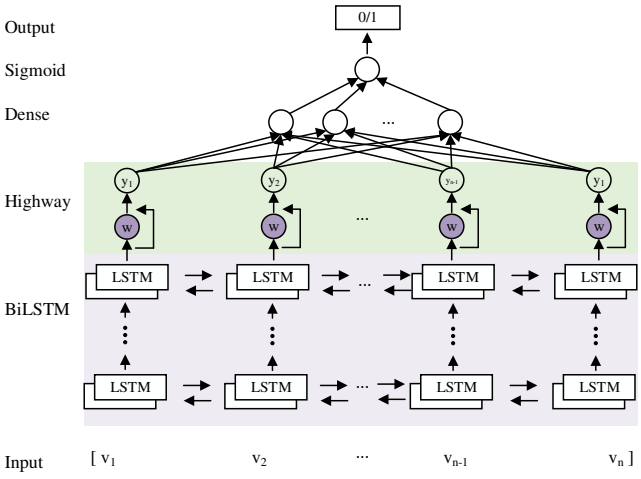


Figure 10: Overview of Highway BiLSTM neural network.

of snippets. Then use the trained detection model to classify it. For each snippet, the model can generate a label for it to achieve the detection purpose. DEVELOPER can detect vulnerabilities at source-code-level.

8. Evaluation

This section introduces preparations for DEVELOPER evaluation, including implementation platform, evaluation datasets, evaluation methodology, and competitive approaches.

8.1. Implementation

The hardware platform used for model training is NVIDIA GeForce GTX 1080 GPU, Intel Xeon E5-1620 CPU. In the Embedding phase, each word's vector dimension is 200 dimensions, and the maximum length of a sample is 500 words. In the training phase, the dimension of hidden layer nodes in BiLSTM is 300. Because there are many types of vulnerabilities, we adjust the hyperparameters of the model within a certain range and choose the best effect, such as size of epoch, the value range of is (100, 200, 300), the value range of size of batch is (32, 64, 128, 256), the value range of Learning rate is (0.005, 0.010, 0.015).

8.2. Datasets evaluation

Open source benchmark database: Benchmark. We generate Benchmark from two sources: NVD and LIU *et al.* [32]. NVD contains vulnerabilities in production software. The detailed page for each vulnerability may contain a mapping link to GitHub, containing *Diff(bug, fixed)* files that describe the difference code snippet between the vulnerable and fixed versions (see Section 5.1). So we search in NVD textual description by vulnerability keyword, get the candidate CVE list, and pay attention to whether the *Diff(bug, fixed)* can be obtained from the mapping link and further extract vulnerable code snippets. The above method is similar to the data collection method of [29, 27, 28]. LIU *et al.* builds their dataset based on five diversified large-scale open-source C projects that incorporate high complexity and variety of real source code instead of synthesis code used in previous works by the manual label. We obtain *Diff(bug, fixed)* files by

Table 3
Details of the experimental data in GitHub and Standard.

Bug type	Compliant commit(G)	Filtered commit(G)	Snippets(G)	Snippet(Standard)
CWE-119	68738	27840	50440	14937
CWE-399	48760	25070	37200	15297
CWE-401	34509	15640	27730	19320
CWE-415	15700	11426	24390	905
CWE-416	27000	17650	24490	459
CWE-020	57699	34350	48600	5637
CWE-022	65430	25470	32070	3290
CWE-129	210045	14230	52000	339
CWE-400	374783	20800	37000	2199
CWE-476	356729	20034	54200	2619

searching for the commit link provided by LIU *et al.* in GitHub. Then we process code files into snippets according to the DEVELOPER's data preprocessing method (slicing and code embedding).

Open source repositories: GitHub. Our approach collects 378,631 repositories from GitHub and 212,510 commits of 10 vulnerability types in these repositories. Slicing according to Sec. 5, we finally obtain 338,120 code snippets. To ensure the data is reliable, we set up a professional review team of 9 people to conduct sampling check on 10% of the data, dividing it into 50 partitions according to *fork* ranking, and randomly select an equal number of commits in each partition. Before the filtering phase in Sec. 4, there are 27.7% commits that do not meet the requirements; after filtering, only 0.3% of the commits did not meet the requirements in three months of manual inspection. The main reason is that the file modification is different from its description.

Standard vulnerability databases: Standard. We generate Standard from SARD. The dataset has been marked as positive, negative, or mixed samples. Finally, we obtained 41,625 vulnerability files from SARD. After slicing files to the function level with *Eclipse JDT and C*, we finally obtain 65,002 snippets.

Table 3 shows the number of data for each vulnerability type.

8.3. Evaluation Methodology

DEVELOPER is evaluated on a ten-fold cross-validation technique and uses the confusion matrix to generate metrics evaluation models for *Precision*, *Recall*, *F1-Score*, *FNR*, and *FPR*. *Precision* measures the correctness of the detected vulnerabilities; *Recall* measures the ratio of the true positive vulnerabilities to the entire population of vulnerable samples; *F1-Score* is the harmonic mean of *Precision* and *Recall*; *FNR* measures the ratio of false-negative vulnerabilities to the entire population of vulnerable samples; *FPR* measures the ratio of false-positive vulnerabilities to the entire population of invulnerable samples.

8.4. Competitive Approaches

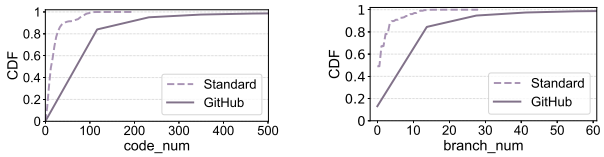
In this paper, DEVELOPER was compared with seven advanced work:

- **VULDEEPECKER** [29]
  VulDeepecker is a fine-grained, automated vulnerability detection model. The core idea of the model is to slice the source code through API call and library call. And collect all code lines related to it, named "Code Gadget." Then using the deep neural network, automatically learns these "Code Gadgets." Their work is based on the data set created by the vulnerability libraries NVD and SARD.
- **VUDDY** [22]
  VUDDY is an approach that uses hash function to scalable yet accurate code clone detection in C/C++. The design principle of VUDDY aims to detect vulnerable clone source software in the open-source repositories at the function level.
- **LIN *et al.*** [30]
  LIN *et al.* is a software vulnerability detection framework via Learning multi-domain knowledge bases. The framework uses Long-short Term Memory (LSTM) cells to detect software vulnerability from SARD and real-world at the function level.
- **FLAWFINDER** [52]
  FLAWFINDER is an open-source code static analysis tool that can detect vulnerabilities in C/C++ language. It mainly relies on vulnerability rules of user-defined to achieve vulnerability detection.
- **FINDBUGS** [8]
  FINDBUGS is an open-source code static analysis tool that could detect vulnerabilities in Java. It uses a variety of static analysis techniques and defines more than 300 different vulnerability rules.
- **TREE-LSTM** [5]
  TREE-LSTM is a generalization of LSTMs to tree-structured network topologies, which network in line with source code AST structure. It is superior to the LSTM in predicting the semantic relevance of the two sentences.
- **CODE2VEC** [1]
  CODE2VEC is a state-of-art code embedding method. The CODE2VEC uses a neural network model to represent a code snippet as a single fixed-length code vector, which can be used to predict semantic properties of the snippet.

9. Evaluation result

The main contribution of our method has proposed a method for collecting high-quality vulnerabilities from open source code repositories and a way of data preprocessing that tries to preserve the source code information. Our evaluation aims to answer the following questions:

- Why do we not just use Standard datasets for vulnerability detection? Is the distribution of vulnerable code differences between the real world and the standard vulnerability library (Sec. 9.1)?
- Can the high-quality vulnerability data we collect from the real world improve the model's detection performance (Sec. 9.2)?
- How well can our detection model on different datasets and different languages (Sec. B.1)?



(a) Distribution of code lines (b) Distribution of code branches

Figure 11: CDF (Cumulative Distribution Function) reflects the probability distribution of the number of control branches in the source code snippets on GitHub and Standard. Vulnerabilities on GitHub are more complex than Standard and contain more control information.

Table 4
Statistics of code lines and code branches in GitHub and Standard.

	Datasets	min	max	mean
code line	Standard	1	200	20
	GitHub	1	5698	176
branch	Standard	0	28	1
	GitHub	0	674	8

- Can the slicing method of source-code-level with data and control flow information capture the source code information more accurately? Can this slicing method be used for other models (Sec. 9.4)?
- Is it necessary to normalize code (Sec. 5.3)? How much does the semantic information of user-defined function names and method names affect the results of the detection model (Sec. 9.5)?
- Does the embedding method of using AST path with vulnerability weight and attention mechanism outperform other SOTA methods (Sec. 9.6)?
- Does the High-way BiLSTM Networks network perform better in vulnerability detection than other networks (Sec. 9.7)?

In addition, it should be noted that in each experiment, we train model and tune parameters with the same dataset using the parameter tuning tool NNI [39].

9.1. Analysis of vulnerability distribution

The purpose of this experiment is to illustrate the necessity of DEVELOPER's collection of GitHub data to detect real-world vulnerabilities. Figure 11 shows the distribution probability of code lines and control branch in code snippets on Standard and GitHub. The number of code lines in Standard is distributed in the range of 0 to 120, while in GitHub, it is distributed in the range of 0 to 500; The number of branches in Standard is distributed in the range of 0 to 4, and it is distributed in the range of 0 to 20 in GitHub. Compared with the real-world vulnerability, the Standard contains fewer code lines and branches. It means that the Standard's vulnerabilities are too simple to support the model to capture the vulnerability patterns well. Furthermore, We count the average, minimum, maximum number of code lines and branches on Standard and GitHub (see Table 4). The vulnerability from Standard contains less control flow information, which means it is not enough to support the predictive model to detect vulnerabilities of the real-world.

Table 5
Performance improvement over the baseline detection models when supplement different training samples.

	Precision	*Recall*	*F1*	*FNR*	*FPR*
Standard	64.2%	55%	59.2%	45.0%	31.4%
Wang	67.9%	57.2%	62.0%	42.8%	23.0%
Zhou	70.3%	62.4%	66.1%	37.6%	22.8%
Developer	76.2%	68.9%	72.4%	31.1%	17.7%

9.2. Evaluation on Data collection module

In this experiment, we evaluate whether the high-quality dataset collected by DEVELOPER could improve the performance of the vulnerabilities detection model. To isolate the impact of our neural network model, we test whether the training data set collected by data collection method could improve the performance of VULDEEPECKER [29], and we take 2,000 data in Benchmark dataset to learn the baseline detection model. Then, we use additional 1,000 code samples (with an equal positive-negative split) that are collected by DEVELOPER, ZHOU *et al.* (Zhou dataset) [57], WANG *et al.* (Wang dataset) [51] to the training dataset to learn a tuned model. We then apply 1,000 test samples from Benchmark, where half of the samples are vulnerabilities. Among them, the data instance of ZHOU *et al.* and WANG *et al.* are code fragments at the function level containing positive and negative samples. Then we process code files into snippets according to the DEVELOPER's data preprocessing method (slicing and code embedding).

As shown in Table 5, additional training data could improve the performance of the detection model. In addition, we notice that DEVELOPER delivers the best overall performance compared with the other three datasets with a 72.4% *F1-Score* and 17.7% *FPR*. Standard gives the lowest performance with a 59.2% *F1-Score*, the reason is that Standard samples usually give old-fashioned cases of vulnerabilities and insufficient amount. WANG *et al.* and ZHOU *et al.* give an improvement to original training samples with 62.0% *F1-Score* and 66.1% *F1-Score*. Due to the fact that our data collection method exploits strict filtering rules and manual review, our system could achieve the highest *F1-Score* than others.

9.3. Evaluation on Bug Detection

To evaluate the performance of DEVELOPER on both GitHub and Standard data sets. We compared three representative works, VULDEEPECKER, VUDDY, and LIN *et al.*, on the GitHub and four works, VULDEEPECKER, FLAWFINDER, LIN *et al.*, FINDBUGS, on Standard.

(1) Evaluation on GitHub.

Compared with VULDEEPECKER and VUDDY in C/C++. We use five kinds of vulnerabilities in C/C++ language for evaluation. Figure 13(a) reports the detection effect of DEVELOPER on each vulnerability type, and Figure 14(a) shows the average metric of three different vulnerability detection frameworks where the min-max bar shows the variance across experimental results.

Among the three different models, *Precision*, *Recall*, and *F1-Score* of DEVELOPER are the highest, *FNR* is the lowest. It means that DEVELOPER can accurately detect more vulnerabilities. A detection model with a lower *FPR* can reduce developers' workforce and material resources to check the vulnerabilities. VUDDY performs a high *Precision* with the lowest *Recall* 15% , which means VUDDY could only detect less than 15% of all vulnerable samples in the test set.

VULDEEPECKER shows a higher FPR since they only use the code data flow information when constructing code gadgets, which could not capture the in-depth code information. For instance, they detect `Double Free` samples based on the word 'free,' which neglects control flow information and causes a high FPR. The research from Sec. 9.1 shows that control dependence is significant for detecting real-world vulnerabilities. DEVELOPER uses data flow and control flow to disperse the structural information of a code fragment into the sequential information of multiple code snippets. This relieves the model's task of capturing structural information of the code and indirectly strengthens the model's ability to capture structural code information. Furthermore, DEVELOPER adds an attention mechanism with vulnerability weight based on word2vec in the embedding phase. At the same time, to consolidate network propagation and better learn the features of vulnerabilities, DEVELOPER has added a highway gate based on BiLSTM. For the above reasons, the *F1-Score* of DEVELOPER is higher than VULDEEPECKER, which ignores semantic code information such as control flow.

However, DEVELOPER has a low precision when facing CWE-401 (still higher than 75%), the memory leak. Because most memory leak vulnerabilities have nothing to do with the code structure, DEVELOPER cannot learn the vulnerable pattern through control dependencies and data dependencies. As shown in Figure 12, the variable `split` applies to allocate space on line 2 and releases space on line 5, but the structure information of the code before and after the repair has not changed. This is a problem with the existing static vulnerability detection model, which might be improved by adding dynamic features.

Compared with LIN *et al.* in JAVA. We collect five types of common vulnerability in JAVA from open source repositories to evaluate our system. Figure 13(b) reports the detection effect of DEVELOPER on each vulnerability type and Figure 14(b) shows the average detection performance of LIN *et al.* and DEVELOPER. We can observe from results that the detection effect of DEVELOPER is better than LIN *et al.* among these three types of metrics, *Precision*, *Recall*, and *F1-Score*, DEVELOPER is 15.9%, 14.1%, and 15.1% higher than LIN *et al.*, respectively. This means that DEVELOPER can detect more vulnerable samples while ensuring the *Precision* of the model. In terms of *FNR* and *FPR*, DEVELOPER is 14.1% and 18.0% lower than LIN *et al.*, which allow DEVELOPER with higher usability than LIN *et al.*.

Although LIN *et al.* uses AST to reserve syntax information, regards code as text directly, and applies WORD2VEC to encode source code, the code is only trained as words. The

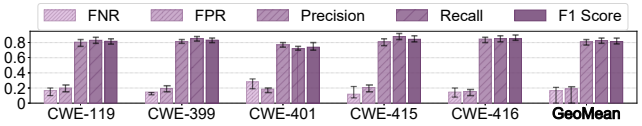
```
1  Continent& MapLoader::createContinent(string& line)
       {
2          vector<string>& split = splitInput(line, '␣
               ');
3          string name = split[0];
4          Continent* continent = new Continent(name);
5          return *continent;
6  }
```

(a) Vulnerability sample(CWE-401)

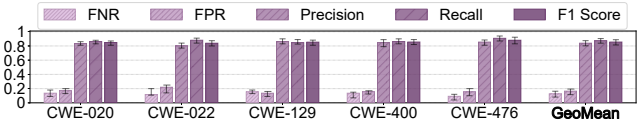
```
1  Continent& MapLoader::createContinent(string& line)
       {
2          vector<string>& split = splitInput(line, '␣
               ');
3          string name = split[0];
4          Continent* continent = new Continent(name);
5          delete& split;
6          return *continent;
7  }
```

(b) Fixed vulnerability sample(CWE-401)

Figure 12: Code (a) in GitHub is a `memory leak` vulnerability sample, and Code (b) in GitHub is a sample which fixed the vulnerability. Code (a) and Code (b), there is no change in the code structure.

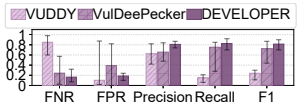


(a) Evaluation DEVELOPER on GitHub in C/C++

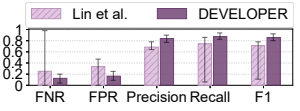


(b) Evaluation DEVELOPER on GitHub in JAVA

Figure 13: Evaluation DEVELOPER on GitHub in C/C++ and JAVA. DEVELOPER gives the better *Precision*, *Recall*, *FNR*, *FPR*, *F1-Score*.



(a) Evaluation in C/C++



(b) Evaluation in JAVA

Figure 14: Compared DEVELOPER with four SOTA vulnerabilities detection frameworks on GitHub. DEVELOPER gives the best *Precision*, *Recall*, *FNR*, *FPR*, *F1-Score*.

structure and semantic information of the source code will be ignored. In this paper, we not only slice the source code according to data and control flow dependence but also use a novel code representation method based on the AST path and attention mechanism. As a result, DEVELOPER extracts vulnerability features more precisely to train the detection model.

(2) Evaluation on Standard.

We evaluate the model's performance on the Standard vulnerability datasets. We selected four data types which have largest amount in SARD data set for experimental evalua-

Table 6
Evaluation VULDEEPECKER, FLAWFINDER and DEVELOPER on Standard in C/C++.

	Bug type	*Precision*	*Recall*	*F1*	*FNR*	*FPR*
VulDeepecker	CWE-119	91.7%	82.0%	86.6%	18.0%	2.9%
	CWE-399	94.6%	95.3%	95.0%	4.7%	2.8%
Flawfinder	CWE-119	26.9%	30.5%	28.6%	69.5%	83.1%
	CWE-399	24.1%	22.0%	23.0%	78.0%	69.5%
Developer	CWE-119	97.1%	91.5%	94.2%	8.5%	2.7%
	CWE-399	97.9%	96.1%	97.0%	3.9%	2.0%

Table 7
Evaluation LIN *et al.*, FINDBUGS and DEVELOPER on Standard in JAVA.

	Bug type	*Precision*	*Recall*	*F1*	*FNR*	*FPR*
Lin *et al.*	CWE-400	85.8%	82.9%	84.3%	17.1%	13.7%
	CWE-476	83.6%	86.4%	85.0%	13.6%	16.9%
Findbugs	CWE-400	24.6%	25.4%	25.0%	74.6%	78.0%
	CWE-476	25.0%	28.8%	26.8%	71.2%	86.4%
Developer	CWE-400	91.2%	91.2%	91.2%	8.8%	8.8%
	CWE-476	89.6%	91.5%	90.5%	8.5%	10.7%

tion.

Compared with VULDEEPECKER, FLAWFINDER in C/C++.
We compared with VULDEEPECKER, FLAWFINDER, and DEVELOPER in CWE-119 and CWE-399 of C/C++, the experimental results are shown in Table 6.

Compared with LIN *et al.*, FINDBUGS in JAVA. Compared LIN *et al.*, FINDBUGS, and DEVELOPER in CWE-400 and CWE-476 of JAVA, the experimental results are shown in Table 7.

Observing Table 6 and Table 7, it can be concluded that DEVELOPER can achieve higher *F1-Score* and lower false-positive rates on the four types of vulnerability, including CWE-119, CWE-399, CWE-400, and CWE-476, and better than VULDEEPECKER, LIN *et al.*, FLAWFINDER, FINDBUGS. The following facts can explain the high *FPR* and *FNR*: FLAWFINDER and FINDBUGS rely on manual-defined rules; they can only detect most of the vulnerabilities with prominent features, which leads to a very high rate of false negatives and false positives. The false-negative rate of DEVELOPER on CWE-119 is higher than that of CWE-399. After analyzing the false-negative samples of DEVELOPER detected, it is found that the main reason is that there is a small number of buffer overflow vulnerability codes that have nothing to do with control flow, which is also one of the limitations of DEVELOPER. In summary, DEVELOPER can maintain a high *F1-Score* on the Standard. At the same time, we achieved a low false-positive rate and a low false-positive rate.

9.4. Evaluation on code slicing

In this experiment we evaluate the impact of that slicing of source-code-level with data and control flow information, and function-level slicing without other information on model detection capabilities. we used the control and data flow dependence slicing method and the function block slicing method to compare the ten kinds of vulnerability collected from the GitHub (the data is collected according to the method in Sec. 4). The method function block slicing, called FUNCTION-SLICE, combined the source vector representation method based on the AST path and attention mechanism. Furthermore, to better illustrate the benefits of source code detection granularity, we introduce the *IoU* score [26]. As shown in formula 11, where A represents the number of lines of real vulnerabilities in the code slicing, and B represents the number of lines of vulnerabilities detected by the model. The closer the *IoU* is to 1, the more accurately the model can locate the vulnerability. Table 8 shows the average *Precision*, *Recall*, *F1-Score*, *FNR*, *FPR*, and *IoU* of the two methods.

Table 8
Evaluation slicing methods of FUNCTION-SLICE and DEVELOPER on GitHub.

	Precision	*Recall*	*F1*	*FNR*	*FPR*	*IoU*
Function-Slice	73.0%	74.9%	73.8%	25.1%	28.1%	25.7%
Developer	82.2%	86.2%	84.1%	13.8%	18.9%	61.3%

$$IoU = \frac{A \cap B}{A \cup B} \tag{11}$$

The Table 8 shows the average *Precision*, *Recall*, and *F1-Score* of DEVELOPER, which are respectively 9.2%, 11.3%, and 10.3% higher than those of FUNCTION-SLICE. *FNR* and *FPR* are 11.3% and 9.3% lower than FUNCTION-SLICE. And *IoU* of DEVELOPER is much higher than FUNCTION-SLICE.

In order to further verify whether our proposed slicing method is suitable for other vulnerability detection models, we select VULDEEPECKER with the best vulnerability detection effect in C/C++ and LIN *et al.* in JAVA. We verify the average effect of the model with and without the DEVELOPER's slicing method on GitHub. The results are shown in Table 9. Among them, the letter before "-" is the abbreviation of the model, L means LIN *et al.* and V means VULDEEPECKER, "S" after "-" indicates the slicing method of the model itself, and "D" indicates the DEVELOPER's slicing method. It can be seen that after using the DEVELOPER's slicing method, the false positive rate of VULDEEPECKER and LIN *et al.* dropped significantly by 19% and 22%, respectively. That shows that VULDEEPECKER only captures the information of API-related streams and cannot wholly summarize the structural information of code. LIN *et al.* only slices at the function-level, which hardly covers the structural information of the code, resulting in a high false-positive rate. DEVELOPER uses data flow and control flow to disperse the structural information of a code fragment into the sequential information of multiple code snippets. This relieves the model's task of capturing structural information of the code and indirectly strengthens the model's ability to capture structural code information. In summary, the slicing method on the source-code-level is superior to the the coarse-grained-level and improves the detection performance of other models.

9.5. Evaluation on Code Normalization

In order to evaluate the impact of the semantic information of the variable names and function names defined by developers (user-defined nodes) on the model, we con-

Table 9
Evaluation of the generalizability of the DEVELOPER slicing method on GitHub.

	Precision	*Recall*	*F1*	*FNR*	*FPR*
V-S	61.1%	69.5%	68.2%	31.1%	58.7%
V-D	65.7%	76.2%	72.9%	24.2%	39.8%
L-S	57.4%	65.3%	66.2%	34.3%	65.4%
L-D	68.1%	74.3%	71.6%	25.5%	34.1%

Table 10
Evaluation on code Normalization and Non-Normalization methods in open source Benchmark database.

	Precision	*Recall*	*F1*	*FNR*	*FPR*
B-N	83.7%	90.1%	86.5%	25.1%	10.7%
B-Non	74.5%	85.3%	73.1%	53.7%	14.2%

ducted validation experiments on the open-source Benchmark database (B). The experimental results are shown in Table 10. The "N" after "-" indicates that user-defined nodes are replaced. The comprehensive metrics F1 of the normalization method is 7% higher than the non-normalization method, which shows that model will be disturbed by noise due to uncertainty of real-world code quality. So, to reduce the negative impact of uncontrollable semantic information, code normalization is necessary. Furthermore, on the Benchmark dataset, which is closer to the real-world vulnerability situation, using the code normalization method to replace the semantic information of user-defined nodes is better than the non-normalized. It shows that DEVELOPER does not rely on the semantics of user-defined nodes for classification. At the same time, Tables 8, 9 in Section 9.4, and 11 in Section 9.6 indicate that DEVELOPER relies on structural code information for classification.

9.6. Compared with code embedding approaches

In this experiment we evaluate the embedding method's impact on the model detection ability. We evaluated the embedding methods of the token sequence (called WORD2VEC), CODE2VEC, a distributed represent of code, and DEVELOPER in GitHub. Among them, the embedding method of the token sequence uses the VULDEEPECKER processing method, which treats the code gadget as text, slices it, and uses WORD2VEC's CBOW model to vectorize the it. CODE2VEC first decomposes the code into a collection of paths in its AST, called path contexts. Then the network learns the atomic representation of each path contexts while simultaneously learning how to aggregate a set of them by attention mechanism. Table 11 shows the average *Precision*, *Recall*, *F1-Score*, *FNR*, and *FPR* of the comparison experiments. The table presents findings that the *F1-Score* of WORD2VEC is the lowest among the three approaches, and *FNR* and *FPR* are also the highest. The experimental effect of CODE2VEC is second. The main reason is WORD2VEC treats the code gadget as text. In that case, the connection between program semantics and vulnerability features is lost, resulting in poor detection results. The behavior of CODE2VEC embedding the AST path as the monolithic symbol will cause sparsity and make its effect poor [20]. Moreover, because CODE2VEC retains the natural language information of all terminal nodes, it overly depends on the quality of the code written by the programmer, resulting in a limited effect.

Table 11
Evaluate embedding methods of WORD2VEC, CODE2VEC and DEVELOPER on GitHub.

	Precision	*Recall*	*F1-Score*	*FNR*	*FPR*
Word2Vec	71.1%	73.7%	72.3%	26.3%	30.2%
Code2Vec	74.8%	84.6%	79.3%	15.4%	35.6%
Developer	82.2%	86.2%	84.1%	13.8%	18.9%

Table 12
Evaluate different networks on GitHub.

	Precision	*Recall*	*F1-Score*	*FNR*	*FPR*
CNN	71.1%	72.6%	71.8%	27.4%	30.6%
LSTM	74.4%	76.8%	75.6%	23.2%	28.1%
BiLSTM	76.2%	78.4%	77.3%	21.6%	26.1%
GRU	74.1%	75.3%	74.7%	24.7%	27.1%
BiGRU	76.7%	79.7%	78.1%	20.3%	26.5%
TreeLSTM	78.2%	81.5%	79.8%	18.5%	25.5%
Developer	84.1%	86.1%	85.1%	13.9%	18.1%

To further illustrate the advantages of DEVELOPER for code vulnerability detection, considering the two code snippet in Figure 15 from two different repositories hosted on GitHub. Both code samples contain an *null pointer dereference* vulnerability, where the program ignores null pointers when referencing pointers. Specifically, `snippet1` does not consider the case where `args` is null: the length of `args` is referenced in the for loop's control condition, and the `args`' value is referenced in the loop body. The same structure appears in the do-while of `snippet2`. Two snippets generate very different token sequences by WORD2VEC, resulting in different prediction results. DEVELOPER uses AST to consider the code's structural information (AST see Figure 16) and find that their AST is similar.

DEVELOPER adds vulnerability weights on the path(marked as purple) to help the model training.

9.7. Compared with different neural networks

In this experiment we evaluate the impact of Highway BiLSTM Networks on model detection capability. We chose six novel neural networks in the current field, including CNN [24], LSTM [17], BiLSTM [13], GRU [13], BiGRU [16], Tree-LSTM [5] to compare with our network. The experimental dataset is the data of ten vulnerabilities collected in the GitHub. Before using competition networks to learn vulnerability features, we use data flow and control flow to slice the code, and then use the AST path and attention mechanism to embed the code. Table 12 shows the average detection effect of seven different models.

We observe that Highway BiLSTM Networks gives the best performance of all SOTA networks in the ten types of vulnerabilities.

CNN shows the lowest *F1-Score* in all types of vulnerabilities with 71.8% *F1-Score*, which means the network ignores the relevance of the part to the global due to their limitations, such as translation invariance [21] in CNN. LSTM its ordinary variant like GRU, BiLSTM, BiGRU give similar *F1-Score* performance in several vulnerabilities between

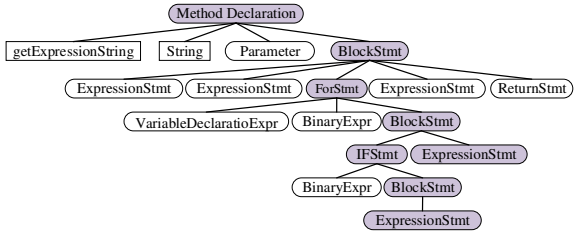
```java
public final String getExpressionString(final String[] args) {
    final StringBuilder sb = new StringBuilder(getExpressionName());
    sb.append("(");
    for (int i = 0; i < args.length; i++) {
        if (i > 0) {
          sb.append(",");
        }
        sb.append(args[i]);
    }
    sb.append(")");
    return sb.toString();
}
```

(a) code Snippet1 from GitHub.

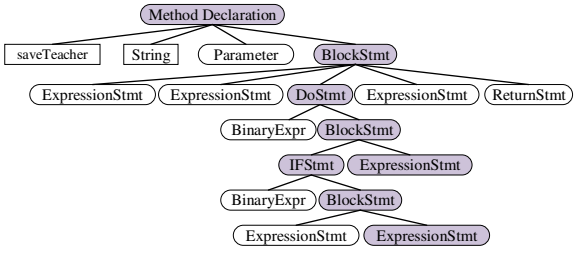
```java
public String saveTeacher(String[] selectClass,String[] selectSubject) {
    List<Integer> selectClassList = new ArrayList<>();
    List<Integer> selectSubjectList = new ArrayList<>();
    int i=-1;
    do{
        i++;
        if (!selectClass[i].equals("") && !selectSubject[i].equals("")) {
            selectClassList.add(Integer.parseInt(selectClass[i]));
            selectSubjectList.add(Integer.parseInt(selectSubject[i]));
        }
    }while( i < selectClass.length);
    return "teacher/add-teacher";
}
```

(b) code Snippet2 from GitHub.

Figure 15: Both (a) and (b) contain a NPE vulnerability. Although the programs are not similar grammar and from two different open source repositories, they have similar control and data flows that lead to a common vulnerability.



(a) AST of `snippet1`.



(b) AST of `snippet2`.

Figure 16: The AST of two snippets in Figure 15. DEVELOPER uses the attention mechanism to add weight to the vulnerability path(marked as purple.)

75.6% to 78.1%. However, relying on the minor update of these variants, their bug detection difference is minimal compared with TreeLSTM. The *Precision* of TreeLSTM is higher than these LSTM's variant networks about 4%, which directly matches the Abstract Syntax Tree representation of source code [5]. DEVELOPER gives the best performance of vulnerabilities detection task due to its networks so that it achieves an 85.1% *F1-Score*, higher about 10% than others. It means Highway Networks allow information to pass through each layer of the deep neural network at high speed without obstruction, which effectively alleviates the problem of gradient boom and allows deep neural networks to transform information with a deeper neural network [46].

10. Limitation and future work

DEVELOPER uses vulnerability data from open source repositories to build a deep learning detection framework. The experiments above show that this approach has a high accuracy rate for detecting vulnerabilities in real development environments, but there are still some limitations in this paper's work.

The framework relies on existing expert rules to collect vulnerability data, and the ruleset's quality impacts the collection results. Moreover, the current rules do not cover all cases on the open-source repository. In the future, we will explore the possibility of automatically collecting vulnerability data on open source repositories.

As mentioned above, to represent the structured information of the vulnerability, we utilized the AST node modeling program. However, we also found that DEVELOPER has a low accuracy rate on some vulnerabilities, such as API abuse and memory leaks. The reason is that these vulnerabilities are either related to a specific function or not related to the structural code information. At the same time, to reduce the dependence on the quality of the code written by developers, we ignore some semantic information about function and variable names. Next, we will consider how to preserve vital static features of the vulnerability code.

In addition, DEVELOPER uses Highway-BiLSTM in constructing the network model module. At present, deep learning technology has the problem of relying on black boxes. It is our future work to provide a theoretical explanation for the underlying working mechanism of DEVELOPER.

Finally, DEVELOPER's data set is mainly in C, C++, and Java. However, DEVELOPER does not limit the language type. Therefore, we can research multiple programming languages and the differences in the performance of models across programming languages in future work.

11. Related work

11.1. Classic vulnerability detection technology

The current mainstream thinking divides static vulnerability detection into two methods: detection based on pattern matching and code similarity [27, 22, 9, 18]. The detection method based on pattern matching generally uses machine learning technology, and its detection results largely rely on the accuracy of the features defined by human experts [27, 6, 14, 54]. Two popular examples of this type of approach are FLAWFINDER [52], and FINDBUGS [8] in the business world. However, these features cannot obtain the semantic information of the source code, the accuracy of the detection model is not high, and the *FNR* is high. A detection method based on code similarity uses known vulnera-

ble code instances to detect the same vulnerable code in the target file [44], the effect depends on the code similarity algorithm designed by human experts. Even if it takes a lot of workforce and material resources, high-risk code caused by code cloning cannot be detected.

11.2. Deep learning-based vulnerability detection

Since deep learning can automatically learn features, many using deep learning technology for vulnerability detection have slowly emerged. [49] extracts the token sequence from the program AST and then uses the deep belief network to learn semantic features from it. PANG *et al.* [41] uses the Long Short-Term Memory (LSTM) network learning vulnerability features in deep learning technology to realize automatic vulnerability detection on software components. LIN *et al.* also designed a deep-learning-based framework with LSTM cells. Their framework combines the heterogeneous data sources to learn unified representations of the patterns of the vulnerable source codes. DAM *et al.* [5] used the TREE-LSTM network to learn source code features to achieve defect prediction automatically. WU *et al.* [53] uses the method body as the granularity of vulnerability detection to study and compare the detection effect of different deep learning models and Multi-Layer Perceptron (MLP). PRADEL *et al.* [42] proposed a name-based vulnerability detection method, DeepBugs, which distinguishes semantically similar identifiers from semantically different identifiers, then uses deep neural networks to implement classifiers. The vulnerability detection granularity of the above several methods is very large. Most of the vulnerabilities are located in a file. LI *et al.* [29] proposed a BiLSTM-based fine-grained vulnerability detection method, which can detect vulnerabilities caused by improper library/API calls. Based on related work, our work has improved some of the shortcomings of existing work.

11.3. Vectorization technology

In the field of natural language processing, GOLDBERG *et al.* [12] has proposed a technology called word embedding. The result of word embedding generates a word vector, which can capture the semantic similarity between words. There are two main source vectorized representation techniques: vectorized representation based on token sequence and vectorized representation based on AST analysis. Based on the vectorized representation of the token sequence, the core idea is to treat the source code as text, slicing it, and generating the token sequence. The token sequence can be obtained from the source code or the AST. Then use the representation learning technology in natural language processing to process the token sequence into a vector. WANG *et al.* [49] arranged the tokens in the order in which they appeared in the source code and AST and then converted the tokens into the tokens' index subscripts in the bag of words. For example, existing works such as [29, 42, 40] flattened program fragments into sequences, and then used WORD2VEC's CBOW model to generate word vectors, and then replaced the tokens with corresponding vectors according to the table lookup method. The vectorized representation method based on the token sequence constructs the source code as an order-sensitive function of the token sequence. Still, it loses the hierarchical structure and grammatical information of the source code.

CODE2VEC [1]is a model based on the AST path and attention mechanism to predict the name of the Java method. Specifically, this method first parses the code into a set of AST paths, then learns the atomic representation of each path context through the network, and uses the attention mechanism to aggregate the AST paths to learn the representation of the code snippet. However, the generalization ability of the DL detection model may be weakened if only the atomic representation of the AST path is learned. In order to alleviate the sparsity of the model and data-hunger problems, DEVELOPER assembles the embedded AST node vectors into an AST path.

CODE2VEC also preserves all the information of terminal nodes. However, considering the noise that the imbalance of real-world code quality may bring, we use normalization (replacing used-define nodes with Func+Number/Var+Number) to ensure that used-define nodes do not affect the training of the model. It is worth noting that the used-define nodes in DEVELOPER and the terminal nodes in CODE2VEC are different. The latter refers to the leaf nodes of the AST path (only part of them are user-defined nodes), so the conclusion of the ablation experiment in CODE2VEC does not apply to DEVELOPER. Second, DEVELOPER relies on code structure information rather than node semantics for classification. The semantic information of terminal nodes is not helpful for all tasks [20]. Additionally, to help the model better focus on the vulnerability feature of the code, DEVELOPER also added vulnerability weights when embedding the AST paths.

12. Conclusion

We have presented DEVELOPER, which is a novel model for vulnerability detection at the source-code level. To obtain a sufficient amount of real-world vulnerability data, DEVELOPER uses syntactic analysis and multiple regular rules to filter high-quality vulnerability data based on GitHub. And we proposed a code preprocessing method that includes a slicing technique, which uses control flow and data flow information, and an embedding technique with an attention mechanism and vulnerability weight. Simultaneously, use the BiLSTM with highway network to train the vulnerability data so that we can accurately capture the grammatical and semantic features of the code. Experiments show that DEVELOPER has a higher *F1-Score* and a lower *FPR* than SOTA works.

CRediT authorship contribution statement

Rongze Xu: Implementation and writing of manuscript. **Zhanyong Tang:** Contribution of specific ideas. **Guixin Ye:** Revising the manuscript. **Huanting Wang:** Experiment. **Xin Ke:** Experiment. **Dingyi Fang:** Supervision of project. **Zheng Wang:** Revising the manuscript.

References

[1] Alon, U., Zilberstein, M., Levy, O., Yahav, E., 2019. code2vec: Learning distributed representations of code. Proceedings of the ACM on Programming Languages 3, 1–29.

[2] Berger, E.D., Hollenbeck, C., Maj, P., Vitek, O., Vitek, J., 2019. On the impact of programming languages on code quality: a reproduction study. ACM Transactions on Programming Languages and Systems (TOPLAS) 41, 1–24.

[3] Büch, L., Andrzejak, A., 2019. Learning-based recursive aggregation of abstract syntax trees for code clone detection, in: 2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER), IEEE. pp. 95–104.

[4] Cummins, C., Petoumenos, P., Murray, A., Leather, H., 2018. Compiler fuzzing through deep learning, in: Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis, pp. 95–105.

[5] Dam, H.K., Pham, T., Ng, S.W., Tran, T., Grundy, J., Ghose, A., Kim, T., Kim, C.J., 2018. A deep tree-based model for software defect prediction. arXiv preprint arXiv:1802.00921 .

[6] Du, X., Chen, B., Li, Y., Guo, J., Zhou, Y., Liu, Y., Jiang, Y., 2019. Leopard: Identifying vulnerable code for vulnerability assessment through program metrics, in: 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE), IEEE. pp. 60–71.

[7] Engineering, H.S.S., (HSSEDI), D.I., . CWE. https://cwe.mitre.org/.

[8] Findbugs, 1995. https://scan.coverity.com/.

[9] Ghaffarian, S.M., Shahriari, H.R., 2017. Software vulnerability analysis and discovery using machine-learning and data-mining techniques: A survey. ACM Computing Surveys (CSUR) 50, 1–36.

[10] GitHub, I., a. GitHub. https://github.com/.

[11] GitHub, I., b. GitHub Docs. https://docs.github.com/en/free-pro-team@latest/github.

[12] Goldberg, Y., Levy, O., 2014. word2vec explained: deriving mikolov et al.'s negative-sampling word-embedding method. arXiv preprint arXiv:1402.3722 .

[13] Graves, A., Schmidhuber, J., 2005. Framewise phoneme classification with bidirectional lstm and other neural network architectures. Neural networks 18, 602–610.

[14] Grieco, G., Grinblat, G.L., Uzal, L., Rawat, S., Feist, J., Mounier, L., 2016. Toward large-scale vulnerability discovery using machine learning, in: Proceedings of the Sixth ACM Conference on Data and Application Security and Privacy, pp. 85–96.

[15] Gu, X., Zhang, H., Zhang, D., Kim, S., 2016. Deep api learning, in: Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, pp. 631–642.

[16] Guo, J., Cheng, J., Cleland-Huang, J., 2017. Semantically enhanced software traceability using deep learning techniques, in: 2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE), IEEE. pp. 3–14.

[17] Hochreiter, S., Schmidhuber, J., 1997. Long short-term memory. Neural computation 9, 1735–1780.

[18] Jang, J., Agrawal, A., Brumley, D., 2012. Redebug: finding unpatched code clones in entire os distributions, in: 2012 IEEE Symposium on Security and Privacy, IEEE. pp. 48–62.

[19] Jiang, S., McMillan, C., 2017. Towards automatic generation of short summaries of commits, in: 2017 IEEE/ACM 25th International Conference on Program Comprehension (ICPC), IEEE. pp. 320–323.

[20] Kang, H.J., Bissyandé, T.F., Lo, D., 2019. Assessing the generalizability of code2vec token embeddings, in: 2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE), IEEE. pp. 1–12.

[21] Kauderer-Abrams, E., 2017. Quantifying translation-invariance in convolutional neural networks. arXiv preprint arXiv:1801.01450 .

[22] Kim, S., Woo, S., Lee, H., Oh, H., 2017. Vuddy: A scalable approach for vulnerable code clone discovery, in: 2017 IEEE Symposium on Security and Privacy (SP), IEEE. pp. 595–614.

[23] Kurata, G., Ramabhadran, B., Saon, G., Sethy, A., 2017. Language modeling with highway lstm, in: 2017 IEEE Automatic Speech Recognition and Understanding Workshop (ASRU), IEEE. pp. 244–251.

[24] Lawrence, S., Giles, C.L., Tsoi, A.C., Back, A.D., 1997. Face recognition: A convolutional neural-network approach. IEEE transactions on neural networks 8, 98–113.

[25] Li, Y., Wang, S., Nguyen, T.N., Van Nguyen, S., 2019. Improving bug detection via context-based code representation learning and attention-based neural networks. Proceedings of the ACM on Programming Languages 3, 1–30.

[26] Li, Z., Zou, D., Xu, S., Chen, Z., Zhu, Y., Jin, H., 2020. Vuldeelocator: a deep learning-based fine-grained vulnerability detector. arXiv preprint arXiv:2001.02350 .

[27] Li, Z., Zou, D., Xu, S., Jin, H., Qi, H., Hu, J., 2016. Vulpecker: an automated vulnerability detection system based on code similarity analysis, in: Proceedings of the 32nd Annual Conference on Computer Security Applications, pp. 201–213.

[28] Li, Z., Zou, D., Xu, S., Jin, H., Zhu, Y., Chen, Z., 2018a. Sysevr: A framework for using deep learning to detect software vulnerabilities. arXiv preprint arXiv:1807.06756 .

[29] Li, Z., Zou, D., Xu, S., Ou, X., Jin, H., Wang, S., Deng, Z., Zhong, Y., 2018b. Vuldeepecker: A deep learning-based system for vulnerability detection. In Proceedings of the 2018 Network and Distributed System Security Symposium URL: http://dx.doi.org/10.14722/ndss.2018.23158, doi:10.14722/ndss.2018.23158.

[30] Lin, G., Zhang, J., Luo, W., Pan, L., De Vel, O., Montague, P., Xiang, Y., 2019a. Software vulnerability discovery via learning multi-domain knowledge bases. IEEE Transactions on Dependable and Secure Computing .

[31] Lin, G., Zhang, J., Luo, W., Pan, L., De Vel, O., Montague, P., Xiang, Y., 2019b. Software vulnerability discovery via learning multi-domain knowledge bases. IEEE Transactions on Dependable and Secure Computing .

[32] Liu, B., Meng, G., Zou, W., Gong, Q., Li, F., Lin, M., Sun, D., Huo, W., Zhang, C., 2020. A large-scale empirical study on vulnerability distribution within projects and the lessons learned, in: 2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE), IEEE. pp. 1547–1559.

[33] Luo, X., Zhou, W., Wang, W., Zhu, Y., Deng, J., 2017. Attention-based relation extraction with bidirectional gated recurrent unit and highway network in the analysis of geological data. IEEE Access 6, 5705–5715.

[34] Manning, C.D., Surdeanu, M., Bauer, J., Finkel, J.R., Bethard, S., McClosky, D., 2014. The stanford corenlp natural language processing toolkit, in: Proceedings of 52nd annual meeting of the association for computational linguistics: system demonstrations, pp. 55–60.

[35] Martinez, M., Duchien, L., Monperrus, M., 2013. Automatically extracting instances of code change patterns with ast analysis, in: 2013 IEEE international conference on software maintenance, IEEE. pp. 388–391.

[36] Meqdadi, O., Aljawarneh, S., 2020. A study of code change patterns for adaptive maintenance with ast analysis. International Journal of Electrical and Computer Engineering 10, 2719.

[37] Mikolov, T., Chen, K., Corrado, G., Dean, J., 2013. Efficient estimation of word representations in vector space. arXiv preprint arXiv:1301.3781 .

[38] Nguyen, T.T., Nguyen, A.T., Nguyen, H.A., Nguyen, T.N., 2013. A statistical semantic language model for source code, in: Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering, pp. 532–542.

[39] NNI, 2018. https://github.com/microsoft/nni.

[40] Pang, Y., Xue, X., Namin, A.S., 2015. Predicting vulnerable software components through n-gram analysis and statistical feature selection, in: 2015 IEEE 14th International Conference on Machine Learning and Applications (ICMLA), IEEE. pp. 543–548.

[41] Pang, Y., Xue, X., Wang, H., 2017. Predicting vulnerable software components through deep neural network, in: Proceedings of the 2017 International Conference on Deep Learning Technologies, pp. 6–10.

[42] Pradel, M., Sen, K., 2018. Deepbugs: A learning approach to name-based bug detection. Proceedings of the ACM on Programming Languages 2, 1–25.

[43] Pundak, G., Sainath, T., 2017. Highway-lstm and recurrent highway networks for speech recognition .

[44] Rattan, D., Bhatia, R., Singh, M., 2013. Software clone detection: A systematic review. Information and Software Technology 55, 1165–1199.

[45] Sheneamer, A., Kalita, J., 2016. Semantic clone detection using machine learning, in: 2016 15th IEEE International Conference on Machine Learning and Applications (ICMLA), IEEE. pp. 1024–1028.

[46] Srivastava, R.K., Greff, K., Schmidhuber, J., 2015. Highway networks. arXiv preprint arXiv:1505.00387 .

[47] Turhan, B., Menzies, T., Bener, A.B., Di Stefano, J., 2009. On the relative value of cross-company and within-company data for defect prediction. Empirical Software Engineering 14, 540–578.

[48] Wang, H., Ye, G., Tang, Z., Tan, S.H., Huang, S., Fang, D., Feng, Y., Bian, L., Wang, Z., 2020a. Combining graph-based learning with automated data collection for code vulnerability detection. IEEE Transactions on Information Forensics and Security .

[49] Wang, S., Liu, T., Tan, L., 2016. Automatically learning semantic features for defect prediction, in: 2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE), IEEE. pp. 297–308.

[50] Wang, W., Li, G., Ma, B., Xia, X., Jin, Z., 2020b. Detecting code clones with graph neural network and flow-augmented abstract syntax tree, in: 2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER), IEEE. pp. 261–271.

[51] Wang, X., Sun, K., Batcheller, A., Jajodia, S., 2019. Detecting" 0-day" vulnerability: An empirical study of secret security patch in oss, in: 2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), IEEE. pp. 485–492.

[52] Wheeler, D.A., 2015. Flawfinder. http://www.dwheeler.com/flawfinder/.

[53] Wu, F., Wang, J., Liu, J., Wang, W., 2017. Vulnerability detection with deep learning, in: 2017 3rd IEEE International Conference on Computer and Communications (ICCC), IEEE. pp. 1298–1302.

[54] Yamaguchi, F., Lottmann, M., Rieck, K., 2012. Generalized vulnerability extrapolation using abstract syntax trees, in: Proceedings of the 28th Annual Computer Security Applications Conference, pp. 359–368.

[55] Ye, G., Tang, Z., Wang, H., Fang, D., Fang, J., Huang, S., Wang, Z., 2020. Deep program structure modeling through multi-relational graph-based learning, in: Proceedings of the ACM International Conference on Parallel Architectures and Compilation Techniques, pp. 111–123.

[56] Zhou, Y., Liu, S., Siow, J., Du, X., Liu, Y., 2019. Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks. arXiv preprint arXiv:1909.03496 .

[57] Zhou, Y., Sharma, A., 2017. Automated identification of security issues from commit messages and bug reports, in: Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, pp. 914–919.

[58] Zou, D., Wang, S., Xu, S., Li, Z., Jin, H., 2019. μvuldeepecker: A deep learning-based system for multiclass vulnerability detection. IEEE Transactions on Dependable and Secure Computing .

Supplemental Materials

In the supporting material, we show the detailed experimental results of Section 9. The data in the table is in the form of **[minimum value, maximum value] geomean value**.

A. Evaluation on Data collection module

Table 1 shows performance improvement over the baseline detection models when supplementing different training samples.

Table 1
Performance improvement over baseline detection models when supplement different training samples.

	Precision	*Recall*	*F1*	*FNR*	*FPR*
Standard	[0.61,0.69] 0.64	[0.51,0.63] 0.55	[0.51,0.62] 0.59	[0.33,0.49] 0.45	[0.25,0.36] 0.31
Wang	[0.63,0.75] 0.68	[0.54,0.61] 0.57	[0.57,0.69] 0.62	[0.39,0.49] 0.43	[0.11,0.36] 0.23
Zhou	[0.67,0.76] 0.70	[0.57,0.69] 0.62	[0.61,0.76] 0.66	[0.29,0.42] 0.37	[0.21,0.29] 0.23
Developer	[0.71,0.81] 0.76	[0.65,0.74] 0.69	[0.67,0.79] 0.72	[0.25,0.34] 0.31	[0.05,0.23] 0.18

B. Evaluation on Bug Detection

B.1. Evaluation on GitHub

Evaluation DEVELOPER on GitHub in C/C++.

Table 2 shows the evaluation result of DEVELOPER on GitHub in C/C++.

Table 2
Evaluation DEVELOPER on GitHub in C/C++.

	Precision	*Recall*	*F1*	*FNR*	*FPR*
CWE-119	[0.75,0.84] 0.81	[0.79,0.87] 0.83	[0.78,0.85] 0.82	[0.10,0.20] 0.16	[0.15,0.24] 0.20
CWE-399	[0.79,0.84] 0.82	[0.82,0.88] 0.85	[0.80,0.86] 0.83	[0.11,0.14] 0.15	[0.15,0.23] 0.19
CWE-401	[0.74,0.80] 0.77	[0.70,0.75] 0.72	[0.71,0.80] 0.74	[0.19,0.32] 0.28	[0.14,0.20] 0.18
CWE-415	[0.76,0.85] 0.81	[0.84,0.92] 0.88	[0.82,0.89] 0.84	[0.07,0.22] 0.12	[0.15,0.24] 0.21
CWE-416	[0.80,0.87] 0.85	[0.81,0.89] 0.85	[0.81,0.90] 0.85	[0.08,0.20] 0.15	[0.10,0.18] 0.15
GeoMean	[0.77,0.83] 0.81	[0.79,0.86] 0.83	[0.78,0.86] 0.82	[0.08,0.21] 0.16	[0.13,0.21] 0.18

Compared with VULDEEPECKER and VUDDY in C/C++.

Table 3 shows the evaluation result of three different vulnerability detection approaches in C/C++.

Table 3
Compared with VULDEEPECKER and VUDDY in C/C++.

	Precision	*Recall*	*F1*	*FNR*	*FPR*
VUDDY	[0.42,0.82] 0.62	[0.06,0.21] 0.15	[0.11,0.30] 0.24	[0.60,0.98] 0.85	[0.02,0.87] 0.10
VulDeePecker	[0.48,0.84] 0.65	[0.28,0.85] 0.75	[0.44,0.87] 0.72	[0.02,0.57] 0.24	[0.00,0.82] 0.39
Developer	[0.74,0.87] 0.80	[0.70,0.92] 0.82	[0.70,0.90] 0.81	[0.07,0.32] 0.16	[0.10,0.24] 0.18

Evaluation DEVELOPER on GitHub in JAVA.

Table 4 shows the evaluation result of DEVELOPER on GitHub in JAVA.

Table 4
Evaluation DEVELOPER on GitHub in JAVA.

	Precision	*Recall*	*F1*	*FNR*	*FPR*
CWE-020	[0.81,0.87] 0.83	[0.83,0.88] 0.86	[0.81,0.87] 0.84	[0.09,0.18] 0.14	[0.13,0.20] 0.17
CWE-022	[0.77,0.84] 0.80	[0.84,0.91] 0.88	[0.80,0.87] 0.84	[0.11,0.23] 0.20	[0.14,0.25] 0.22
CWE-129	[0.83,0.90] 0.86	[0.82,0.89] 0.84	[0.81,0.88] 0.85	[0.13,0.18] 0.15	[0.09,0.16] 0.14
CWE-400	[0.79,0.89] 0.85	[0.83,0.90] 0.86	[0.82,0.89] 0.86	[0.07,0.15] 0.14	[0.12,0.17] 0.17
CWE-476	[0.81,0.88] 0.86	[0.87,0.95] 0.91	[0.83,0.92] 0.88	[0.04,0.12] 0.09	[0.10,0.20] 0.15
GeoMean	[0.80,0.87] 0.84	[0.84,0.94] 0.87	[0.81,0.89] 0.86	[0.08,0.16] 0.13	[0.11,0.19] 0.16

Compared with LIN *et al.* in JAVA.

Table 5 shows the evaluation result of two different vulnerability detection frameworks in JAVA.

Table 5
Compared with LIN *et al.* in JAVA.

	Precision	*Recall*	*F1*	*FNR*	*FPR*
Lin *et al.*	[0.64,0.78] 0.68	[0.06,0.86] 0.74	[0.11,0.78] 0.71	[0.00,0.98] 0.25	[0.00,0.47] 0.33
Developer	[0.77,0.90] 0.83	[0.83,0.94] 0.87	[0.80,0.92] 0.85	[0.04,0.20] 0.12	[0.09,0.25] 0.16

B.2. Evaluation on Standard

Compared with VULDEEPECKER, FLAWFINDER in C/C++.

Table 6 shows the evaluation result of three different vulnerability detection frameworks in C/C++.

Table 6
Evaluation VULDEEPECKER, FLAWFINDER and DEVELOPER on Standard in C/C++.

	Bug type	*Precision*	*Recall*	*F1*	*FNR*	*FPR*
VulDeepecker	CWE-119	[0.81,0.98] 0.92	[0.77,0.87] 0.82	[0.81,0.92] 0.86	[0.12,0.26] 0.18	[0.00,0.05] 0.02
	CWE-399	[0.84,1.00] 0.95	[0.82,1.00] 0.95	[0.89,0.98] 0.95	[0.02,0.08] 0.05	[0.00,0.10] 0.03
Flawfinder	CWE-119	[0.05,0.36] 0.27	[0.03,0.35] 0.29	[0.19,0.32] 0.28	[0.59,0.74] 0.69	[0.71,0.82] 0.74
	CWE-399	[0.03,0.42] 0.24	[0.05,0.36] 0.23	[0.07,0.26] 0.23	[0.72,0.84] 0.78	[0.82,0.96] 0.93
Developer	CWE-119	[0.91,1.00] 0.97	[0.85,0.95] 0.91	[0.91,1.00] 0.94	[0.02,0.18] 0.08	[0.00,0.13] 0.03
	CWE-399	[0.88,1.00] 0.98	[0.91,1.00] 0.96	[0.92,1.00] 0.97	[0.00,0.11] 0.04	[0.00,0.09] 0.02

Compared with LIN *et al.*, FINDBUGS in JAVA.

Table 7 shows the evaluation result of three different vulnerability detection frameworks in JAVA.

Table 7
Evaluation LIN *et al.*, FINDBUGS and DEVELOPER on Standard in JAVA.

	Bug type	*Precision*	*Recall*	*F1*	*FNR*	*FPR*
Lin *et al.*	CWE-400	[0.79,0.93] 0.86	[0.76,0.84] 0.82	[0.81,0.80] 0.84	[0.12,0.24] 0.17	[0.08,0.20] 0.14
	CWE-476	[0.77,0.89] 0.84	[0.81,0.89] 0.86	[0.79,0.88] 0.85	[0.02,0.21] 0.13	[0.05,0.21] 0.17
FindBugs	CWE-400	[0.21,0.36] 0.25	[0.17,0.29] 0.25	[0.19,0.31] 0.25	[0.69,0.82] 0.74	[0.74,0.82] 0.78
	CWE-476	[0.19,0.31] 0.25	[0.15,0.36] 0.29	[0.15,0.31] 0.28	[0.63,0.79] 0.71	[0.77,0.89] 0.86
Developer	CWE-400	[0.87,1.00] 0.91	[0.88,0.95] 0.91	[0.88,1.00] 0.91	[0.04,0.12] 0.09	[0.01,0.12] 0.08
	CWE-476	[0.85,1.00] 0.89	[0.89,1.00] 0.92	[0.87,1.00] 0.92	[0.00,0.10] 0.08	[0.00,0.13] 0.10

C. Evaluation on code slicing

Table 8 shows the evaluation result of FUNCTION-SLICE and DEVELOPER on code slicing.

Table 8
Evaluation slicing methods of FUNCTION-SLICE and DEVELOPER on GitHub.

	Precision	*Recall*	*F1*	*FNR*	*FPR*	*IoU*
Function-Slice	[0.79,0.93] 0.73	[0.71,0.79] 0.75	[0.69,0.81] 0.74	[0.17,0.29] 0.25	[0.21,0.32] 0.28	[0.19,0.29] 0.26
Developer	[0.77,0.89] 0.82	[0.81,0.89] 0.86	[0.79,0.88] 0.84	[0.08,0.21] 0.14	[0.11,0.23] 0.19	[0.55,0.63] 0.61

Table 9 shows the evaluation result of the generalizability of the DEVELOPER slicing method on GitHub.

Table 9
Evaluation of the generalizability of the DEVELOPER slicing method on GitHub.

	Precision	*Recall*	*F1*	*FNR*	*FPR*
V-S	[0.57,0.65] 0.61	[0.63,0.73] 0.69	[0.61,0.73] 0.68	[0.27,0.42] 0.31	[0.51,0.62] 0.58
V-D	[0.59,0.68] 0.65	[0.71,0.82] 0.76	[0.68,0.79] 0.72	[0.13,0.27] 0.24	[0.24,0.47] 0.39
L-S	[0.55,0.61] 0.57	[0.62,0.67] 0.65	[0.62,0.69] 0.66	[0.19,0.44] 0.34	[0.45,0.69] 0.65
V-D	[0.62,0.72] 0.68	[0.71,0.80] 0.74	[0.65,0.76] 0.71	[0.11,0.18] 0.25	[0.28,0.39] 0.34

D. Evaluation on Code Normalization

Table 10 shows the evaluation result of code Normalization and Non-Normalization methods in Standard and Benchmark datasets.

Table 10
Evaluation on code Normalization and Non-Normalization methods in Standard and Benchmark datasets.

	Precision	*Recall*	*F1*	*FNR*	*FPR*
S-N	[0.90,1.00] 0.95	[0.90,1.00] 0.93	[0.85,0.96] 0.93	[0.00,0.05] 0.03	[0.00,0.12] 0.07
S-Non	[0.95,1.00] 0.98	[0.96,1.00] 1.00	[0.88,1.00] 0.97	[0.00,0.03] 0.01	[0.00,0.08] 0.04
B-N	[0.75,0.86] 0.83	[0.82,1.00] 0.90	[0.79,0.89] 0.86	[0.10,0.30] 0.25	[0.05,0.16] 0.10
B-Non	[0.62,0.76] 0.74	[0.79,0.91] 0.85	[0.69,0.77] 0.73	[0.34,0.62] 0.53	[0.07,0.23] 0.14

E. Compared with code embedding approaches

Table 11 shows the evaluation result of WORD2VEC, CODE2VEC and DEVELOPER on GitHub.

Table 11
Evaluate embedding methods of WORD2VEC, CODE2VEC and DEVELOPER on GitHub.

	Precision	*Recall*	*F1-Score*	*FNR*	*FPR*
Word2Vec	[0.67,0.75] 0.71	[0.71,0.79] 0.74	[0.69,0.76] 0.72	[0.21,0.37] 0.26	[0.22,0.34] 0.30
Code2Vec	[0.71,0.76] 0.75	[0.75,0.89] 0.85	[0.73,0.82] 0.79	[0.10,0.21] 0.15	[0.31,0.41] 0.35
Developer	[0.79,0.84] 0.82	[0.82,0.89] 0.86	[0.77,0.87] 0.84	[0.05,0.18] 0.14	[0.09,0.23] 0.19

F. Compared with different neural networks

Table 12 shows the evaluation result of different networks on GitHub.

Table 12
Evaluate different networks on GitHub.

	Precision	*Recall*	*F1-Score*	*FNR*	*FPR*
CNN	[0.67,0.75] 0.71	[0.65,0.79] 0.72	[0.64,0.76] 0.72	[0.21,0.37] 0.27	[0.22,0.34] 0.30
LSTM	[0.71,0.81] 0.74	[0.72,0.85] 0.77	[0.69,0.82] 0.76	[0.10,0.26] 0.23	[0.16,0.35] 0.28
BiLSTM	[0.69,0.82] 0.76	[0.72,0.83] 0.78	[0.69,0.82] 0.77	[0.12,0.29] 0.22	[0.21,0.31] 0.26
GRU	[0.67,0.79] 0.74	[0.71,0.79] 0.75	[0.69,0.77] 0.74	[0.21,0.33] 0.25	[0.22,0.34] 0.27
BiGRU	[0.69,0.82] 0.76	[0.75,0.89] 0.80	[0.73,0.82] 0.78	[0.15,0.26] 0.20	[0.21,0.41] 0.27
TreeLSTM	[0.66,0.84] 0.78	[0.76,0.89] 0.81	[0.77,0.87] 0.80	[0.11,0.25] 0.18	[0.15,0.29] 0.25
Developer	[0.77,0.92] 0.84	[0.75,0.91] 0.86	[0.78,0.91] 0.85	[0.11,0.23] 0.14	[0.05,0.21] 0.18