2022 IEEE 46th Annual Computers, Software, and Applications Conference (COMPSAC) | 978-1-6654-8810-5/22/$31.00 ©2022 IEEE | DOI: 10.1109/COMPSAC54236.2022.00142

# Can Deep Learning Models Learn the Vulnerable Patterns for Vulnerability Detection?

Guoqing Yan[1], Sen Chen[1*], Yude Bai[1], Xiaohong Li[1*]
[1]*College of Intelligence and Computing, Tianjin University, Tianjin, China*
{*guoqingyan, senchen, baiyude, xiaohongli*}@*tju.edu.cn*

***Abstract*—Deep learning has been widely used for the security issue of vulnerability prediction. However, it is confusing to explain how a deep learning model makes decisions on the prediction, although such a model achieves a good performance. Meanwhile, it is also difficult to discover which part of the source code is concentrated on by this black-box model. To this end, we present an empirical evaluation to explore how the deep learning model works on predicting vulnerability and whether it precisely captures the critical code segments to represent the vulnerable patterns. First of all, we build a new vulnerability dataset, called Juliet+, in which vulnerability-related code lines of both positive (bad) and negative (good) samples are labeled manually with substantial efforts, based on the Juliet Test Suite. After that, four deep learning models by leveraging attention mechanisms are empirically implemented to detect vulnerability through mining vulnerable patterns from the source code. We conduct extensive experiments to evaluate the effectiveness of such four models and to analyze the interpretability with evaluation metrics such as Hit@k. The empirical experiment results reveal that the deep learning models with attention, to some extent, can focus on the vulnerability-related code segments that are profitable to interpret the result of vulnerability detection, especially when we adopt the graph neural network model. We further investigate what factors affect the interpretability of models including the class distribution, the number of samples, and the differences of sample features. We find the graph neural network model performs better on part of the dataset which contains balanced and sufficient samples with obvious differences between vulnerable and non-vulnerable patterns.**

***Index Terms*—Vulnerability detection, Deep learning, Attention, Model interpretability**

## I. Introduction

As the popularity of deep learning (DL) in the fields of computer vision and natural language processing (NLP), it has also been adopted to automatically predict software vulnerability. Many studies [1]–[7] verify the effectiveness when taking Recurrent Neural Networks (RNNs), Graph Neural Networks (GNNs), or Deep Belief Neural Networks (DBN) [8], to recognize software vulnerability. However, all of them ignore the interpretability due to the black-box property of the DL models which lowers the reliability of the predicted results [9]. Moreover, Chakraborty et al. [5] claim that DL models learn certain irrelevant features which are harmful to improve the performance of vulnerability detection [1], [2]. To this end, it is urgent to systematically investigate whether DL models can distinguish and understand the differences between vulnerable and non-vulnerable patterns from the source code.

In this study, we employ the attention mechanisms [10], which are advantageous for enhancing performance on tasks such as text classification [11], [12] and machine translation [13], to uncover and understand the predictions of the DL models. By appending attention, we obtain the attention value for code segments to represent which part of the code is focused on by the DL models, instead of just using attention to raise the detection accuracy [6], [14]. However, due to the lack of vulnerability datasets in which vulnerability-related code lines are first labeled, it is difficult to perform an attention-value-based interpretability analysis of the models.

Generally, there are three types of vulnerability datasets [5]: real-world, semi-synthetic, and synthetic. The real-world vulnerability dataset indicates the development of software while there is no widely-used dataset. This mainly derives from that security experts collect vulnerability datasets in different ways, the granularity of vulnerability detection is various and the public available real-world datasets are generally small in size [15], [16]. Moreover, it is complex and time-consuming even for security experts to mark the important vulnerability-related code segments from real-world vulnerability datasets. Since the semi-synthetic vulnerability datasets like Draper [17] are still for real-world software, it is also difficult to label the accurate vulnerability-related code lines. We finally choose the synthetic vulnerability dataset Juliet Test Suite for Java 1.3 (Juliet[1]). The special kind of designs in Juliet is beneficial for us to quickly and accurately find the vulnerability-related code lines from the source code. We thus manually construct a well-labeled vulnerability dataset Juliet+ depending on Juliet.

Two categories of DL models are applied for vulnerability detection, the sequence-based models (e.g., Bidirectional Long Short-Term Memory Networks (BLSTM) [1] and Bidirectional Gated Recurrent Unit (BGRU) [2]) and the GNN-based models [3]–[6]. The former models are similar to sequence modeling in NLP and they take flat sequences of code as input so that attention is easy to add into those models. The latter ones consider the structural and logical information of the source code. We finally implement four attention-based models to demonstrate the interpretability of them by considering the similarity of source code and natural language, i.e., Hierarchical Attention Network (HAN) [12], Single Attention Network (SAN), Attention-based Bidirectional Long Short-Term Memory Network (BLSTM-att) [11], and Relation

[*]Sen Chen and Xiaohong Li are the corresponding authors.

[1]Juliet Test Suite for Java 1.3, https://samate.nist.gov/SRD/testsuite.php.

Graph Convolution Network with attention (R-GCN-att) [6] (see Section II-E for details). The first three models are the sequence-based models that regard source code as the natural language directly. The R-GCN-att, as a GNN-based model, learns the information from the graph structure (nodes, edges, etc.) of the source code before calculating the attention score of each node in this graph.

Extensive experiments are conducted to investigate how the attention-based DL models predict software vulnerability and interpret the prediction results. We first compare the detection results of vulnerability across such four DL models. They all gain a high F1-score of more than 95% on Juliet+. This leads to the followed evaluation about attention whether the DL models have learned the vulnerable patterns or not. After the analysis on the effects of attention (by attention score), we observe that the GNN-based R-GCN-att outperforms other models and can explain the results of vulnerability detection. We further explore which factors affect the attention score of R-GCN-att more. It is shown that the R-GCN-att will achieve better performance when adopting the dataset that contains more balanced and sufficient samples with obvious differences between vulnerable and non-vulnerable patterns. Note that all of the dataset, the source code of models, and the experimental results are available at https://github.com/Ng13oTy/Interpretability.

In summary, we make the following main contributions:

- To the best of our knowledge, this is the first work to quantitatively and intuitively analyze the interpretability of the DL models for vulnerability detection based on a new dataset named Juliet+ that vulnerability-related code lines of each sample are manually labeled.
- We implement four attention-based DL models, including both the sequence-based models and the graph neural network-based model, to validate the problem of interpretability for software vulnerability detection.
- We define two metrics to evaluate the interpretability of such four DL models. The empirical results reveal that the GNN-based DL model R-GCN-att better interprets the predictions of software vulnerability because it can more precisely focus on the critical code segments that represent the vulnerable patterns.

## II. Methodology

In this section, we formulate the issue of vulnerability detection in Section II-A and then introduce the proposed framework in Section II-B that includes data preprocessing (Section II-C), model training and testing (Section II-D). Section II-E presents the four attention-based DL models.

### A. The Definition of Vulnerability Detection

According to the definition of vulnerable function detection in [3], we further formulate the issue of vulnerability detection as follows. The dataset with label is defined as $\{(c_i, y_i)\,|c_i \in C, y_i \in Y\}$, $i \in \{1, 2, \cdots, n\}$, where $C$ denotes the set of samples in the form of source code, $Y = \{0, 1, type-1\}^n$ refers to the label set with $type$ as the number of sample types, and $n$ is the number of samples in the dataset. The goal of vulnerability detection is to learn a mapping from $C$ to $Y$, i.e., $\Phi : C \mapsto Y$. This mapping $\Phi$ represents the process that turns the source code from initial (low-level) features to high-level features and finally obtain the classification result via minimizing the loss function of DL models below,

$$min\sum_{i=1}^{n} L(\Phi(ll_i, hl_i, y_i\,|c_i)) + \lambda\omega(f)\,, \tag{1}$$

where $L(\cdot)$ means the cross entropy loss function, $\lambda\omega(f)$ is regarded as the penalty parameter, $ll_i$ and $hl_i$ imply the initial (low-level) and high-level features of the $i$-th sample $c_i$ respectively.

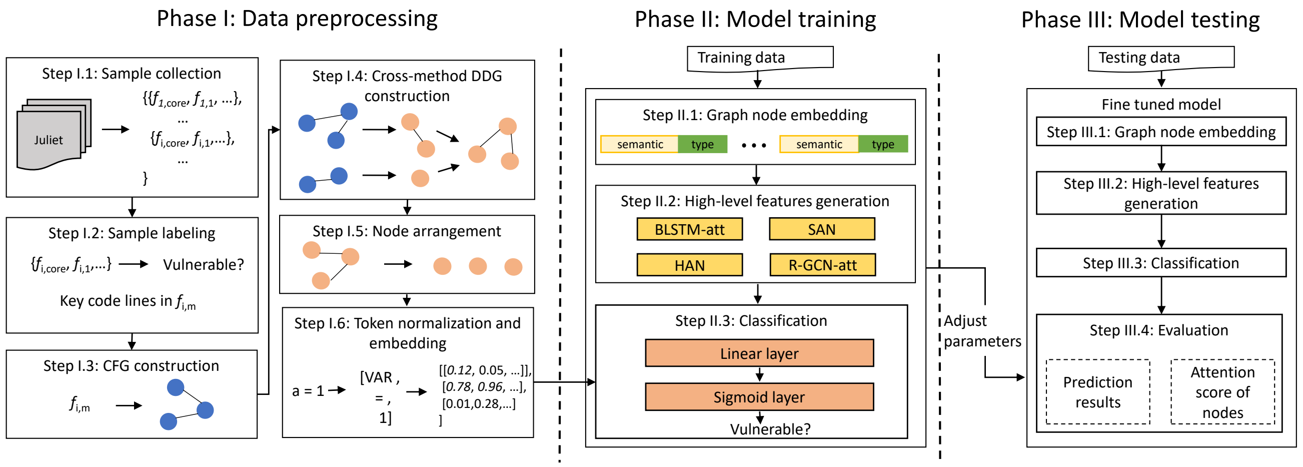
### B. Overview of the Framework

As shown in Fig. 1, the proposed framework contains three phases to validate the interpretability of DL models for vulnerability detection.

*1) Data Preprocessing:* In this step, we make an effort of marking the vulnerability dataset Juliet+ collected from Juliet at first. We not only label whether a sample in Juliet+ is vulnerable or not but also label the corresponding vulnerability-related code lines of this sample. Then the well-labeled source code is turned into a data dependency graph (DDG). We choose DDG as the initial input due to two reasons. The first one is that software vulnerabilities often happen to the data flow when it is performed during vulnerability detection. Secondly, DDG more precisely illuminates the information about source code since it derives from the control flow graph (CFG) describing all the possible execution paths of the source code. DDG also removes much redundant information that is irrelevant to the vulnerabilities. We show the specific sub-steps (six sub-steps in total) of data preprocessing in Section II-C.

*2) Model Training:* We train the attention-based DL models through learning the high-level features different from the initial features in this step. There are three sub-steps including the graph node embedding, the generation of high-level features, and classification. In the process of graph node embedding, each node in a DDG is embedded after combining the text information (semantic information) and type information of such node. Four attention-based DL models learn knowledge from the embedded node of DDG to generate its high-level features. Then we conduct a binary classifier to discriminate whether there is a vulnerability in the code via decoding the high-level features given by the DDG.

*3) Model Testing:* With the well-trained DL models, the DDG of a new sample is detected to determine whether it is vulnerable or not. Moreover, we analyze the interpretability of these models based on the attention score of each node in the DDG of the source code. Since the attention effects of the four models only need to be output during testing, only the vulnerability-related code lines of the samples in the test set are manually labeled. We customize two criteria Hit@k, Hit@k% and several hit types to comprehensively evaluate the interpretability of the models.



Phase I: Data preprocessing
Step I.1: Sample collection
Juliet
{{$f_{1,core}$, $f_{1,1}$, …},
…
{$f_{i,core}$, $f_{i,1}$,…},
…
}
Step I.2: Sample labeling
{$f_{i,core}$, $f_{i,1}$,…}
Vulnerable?
Key code lines in $f_{i,m}$
Step I.3: CFG construction
$f_{i,m}$
Step I.4: Cross-method DDG construction
Step I.5: Node arrangement
Step I.6: Token normalization and embedding
a = 1
[VAR , = , 1]
[[0.12, 0.05, …]], [0.78, 0.96, …], [0.01,0.28,…] ]
Phase II: Model training
Training data
Step II.1: Graph node embedding
semantic
type
semantic
type
Step II.2: High-level features generation
BLSTM-att
SAN
HAN
R-GCN-att
Step II.3: Classification
Linear layer
Sigmoid layer
Vulnerable?
Adjust parameters
Phase III: Model testing
Testing data
Fine tuned model
Step III.1: Graph node embedding
Step III.2: High-level features generation
Step III.3: Classification
Step III.4: Evaluation
Prediction results
Attention score of nodes

Fig. 1: The framework to validate the interpretability of DL models for vulnerability detection.

## C. Data Preprocessing

We build the vulnerability dataset Juliet+ based on the initial dataset Juliet, which comes from the Software Assurance Metrics And Tool Evaluation [18] (SAMATE). According to the official document [19], the mixed-type test cases which constitute the vast majority of Juliet should contain one bad execution and at least one relevant good execution. We thus need to collect samples firstly by splitting the bad and good executions. Besides, each test case should have a primary file that contains a primary bad method and a primary good method (see the test case 128562[2] as an example). This primary good method consists of one or more secondary good methods which refer to the start of the good executions. Furthermore, this primary bad method represents the start of the bad execution directly. In addition, we adopt the concepts of "source" and "sink" to express the data flow of the source code [19]. The "source" means how to acquire the value of a variable defined. The "sink" denotes how to use the variable defined. We specifically describe the generation of the initial features of the source code in the following six sub-steps.

*1) Sub-step 1: Sample collection.* Depending on the powerful language analysis tool *JavaParser* [20], we firstly parse the source code of the test cases in Juliet and then extract both the methods and all of the related calling information in each test case. The primary bad method and the secondary good method are formed as the core method $f_{i,core}$. Then a sample $c_i$ of Juliet+, a collection of methods, is defined as $c_i = \{f_{i,core}, f_{i,1}, f_{i,2}, \cdots\}$, where $f_{i,m}$, $m \in \{1, 2, \cdots\}$, is a method that is called by $f_{i,core}$ directly or indirectly. Fig. 2 implies the source code of three samples extracted from the test case 128562 in Juliet.

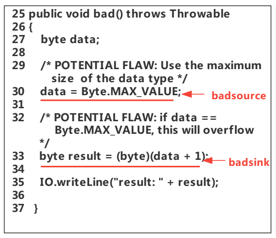
*2) Sub-step 2: Sample labeling.* we describe how to label a sample (such as the ones given in Fig. 2) as vulnerable or not and how to mark the corresponding vulnerability-related code lines in this sub-step.

① **Labeling the sample.** To the sample with a core method $f_{i,core}$, we label it as a positive or negative sample by the method of the regular expression. If the name of $f_{i,core}$ matches the pattern "^bad$", the relevant sample is regarded as a bad sample (vulnerable). On the contrary, if the name of $f_{i,core}$ meets the pattern "^good(\d+|G2B\d*|B2G\d*)$", this sample is a good one (non-vulnerable). In this way, the test case in Fig. 2 is comprised of one bad sample (Fig. 2 (a)) and two good samples (Fig. 2 (b) and Fig. 2 (c)).

② **Labeling the vulnerability-related code lines.** Although Juliet has already marked certain code lines that are related to the vulnerabilities, we find that some of those lines are mislabeled. Moreover, it does not provide the repaired code lines about the good executions and the lines of vulnerable codes which are not at the primary file. Therefore, it is necessary to relabel the vulnerability-related code lines at first.

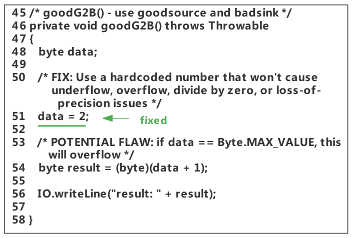
According to [19], a bad sample that has both "badsource" (a piece of code) and "badsink" (a piece of code) contains at least one software vulnerability related to the data flow. We thus relabel all bad samples with two properties, "bad_source_lines" (the code lines of "badsource") and "bad_sink_lines" (the code lines of "badsink"). For example, as is shown in Fig. 2 (a), the code line 30 are labeled as the "bad_source_lines" due to that the variable "data" gets the maximum value, while the code line 33 is "bad_sink_lines" because it leads to an overflow problem after adding one to the "data" in this line. When the vulnerable code of the bad sample is repaired, it changes into a good sample. Specifically, two ways are given to remedy the vulnerable code. The one is to give the variable a "goodsource" and the other is to use the variable with a "goodsink". We define the good samples with a property of "fixed_lines". For instance, the vulnerable code of a bad sample in Fig. 2 (a) becomes a good sample in Fig. 2 (b) when the overflow problem is solved by setting "data" with a constant value 2 (the way to give the variable a "goodsource"). This bad sample is also turned into a good sample by using the variable with a "goodsink", which adds a check (the "if" statement at line 69 in Fig. 2 (c)) to prevent the overflow. To ensure the validity of the labeled code, we employ not only the lines of code given by Juliet but also the comments in the

[2]The test case 128562, https://samate.nist.gov/SRD/view_testcase.php?tID=128562.



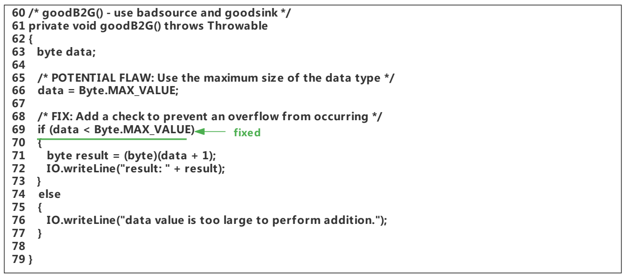
```
25 public void bad() throws Throwable
26 {
27    byte data;
28
29    /* POTENTIAL FLAW: Use the maximum
         size  of the data type */
30    data = Byte.MAX_VALUE;
31
32    /* POTENTIAL FLAW: if data ==
          Byte.MAX_VALUE, this will overflow
      */
33    byte result = (byte)(data + 1);
34
35    IO.writeLine("result: " + result);
36
37  }
```

(a) A bad sample with badsource and badsink



```
45 /* goodG2B() - use goodsource and badsink */
46 private void goodG2B() throws Throwable
47 {
48   byte data;
49
50   /* FIX: Use a hardcoded number that won't cause
        underflow, overflow, divide by zero, or loss-of-
          precision issues */
51   data = 2;
52
53   /* POTENTIAL FLAW: if data == Byte.MAX_VALUE, this
       will overflow */
54   byte result = (byte)(data + 1);
55
56   IO.writeLine("result: " + result);
57
58 }
```

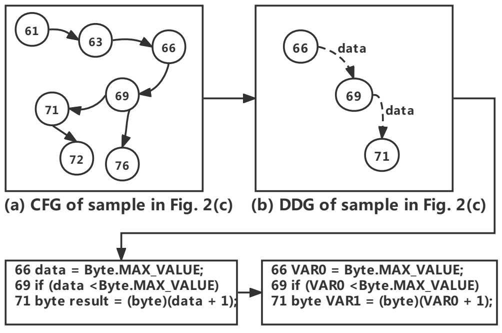
(b) A good sample with goodsource and badsink



```
60 /* goodB2G() - use badsource and goodsink */
61 private void goodB2G() throws Throwable
62 {
63   byte data;
64
65   /* POTENTIAL FLAW: Use the maximum size of the data type */
66   data = Byte.MAX_VALUE;
67
68   /* FIX: Add a check to prevent an overflow from occurring */
69   if (data < Byte.MAX_VALUE)
70   {
71     byte result = (byte)(data + 1);
72     IO.writeLine("result: " + result);
73   }
74   else
75   {
76     IO.writeLine("data value is too large to perform addition.");
77   }
78
79 }
```

(c) A good sample with badsource and goodsink

Fig. 2: The source code of samples extracted from the test case 128562 in Juliet.



61
63
66
69
71
72
76
(a) CFG of sample in Fig. 2(c)
66
data
69
data
71
(b) DDG of sample in Fig. 2(c)

```
66 data = Byte.MAX_VALUE;
69 if (data <Byte.MAX_VALUE)
71 byte result = (byte)(data + 1);
```

(c) Arranged nodes of DDG

```
66 VAR0 = Byte.MAX_VALUE;
69 if (VAR0 <Byte.MAX_VALUE)
71 byte VAR1 = (byte)(VAR0 + 1);
```

(d) Normalized nodes of DDG

Fig. 3: Example of a workflow from Step I.3 to Step I.6.

source code, such as "/POTENTIAL FLAW:..." and "/FIX:..." shown in Fig. 2.

*3) Sub-step 3: CFG construction.* After labeling the sample $c_i$, we utilize *JavaParser* to parse the source code and create the abstract syntax tree (AST) for each method in sample $c_i$. Then, based on this AST, we build a corresponding control flow graph (CFG) with the variable information. Fig. 3 (a) illuminates the relevant CFG of the sample in Fig. 2 (c).

*4) Sub-step 4: Cross-method DDG construction.* Each CFG is transformed into a DDG according to the given algorithms in [21]. It then becomes a cross-method DDG depending on the calling relationship across the methods in $c_i$. This constructed cross-method DDG contains not only all nodes with variable information but also all nodes of CFG with calling information, which ensures the integrity of the cross-function vulnerable patterns. As is shown in Fig. 3 (b), the sample in Fig. 2 (c) turns into a DDG with three nodes. More details about how to build a DDG are available on our website https://github.com/Ng13oTy/Interpretability.

*5) Sub-step 5: Node arrangement.* After getting the cross-method DDG of $c_i$, we need to arrange the nodes in DDG as a sequence to satisfy the sequence-based models. Similar to [2], we sort the nodes in the same method according to the order of code lines. For the nodes among different methods, we arrange them based on the order of depth and breadth of the calling relations. The depth refers to the order in which methods are run in the calling method. The breath indicates what level the methods called by $f_{i,core}$ are at (e.g., the methods called by $f^{i}_{core}$ are at level 0 and the methods called by the methods of level 0 are at level 1). Fig. 3 (c) presents the nodes arranged from the cross-method DDG in Fig. 3 (b).

*6) Sub-step 6: Token normalization and embedding.* Because the corpus of source code is much larger than that of natural language [22], we flatten the source code of the arranged nodes and normalize it by using the method proposed in [1], [2]. Fig. 3 (d) shows the result of normalization for the arranged nodes in Fig. 3 (c). Since the DL models can not process the normalized source code directly, we have to embed it. We embed each of the normalized nodes by *Word2vec* [23] to a 50-dimension neural vector. In this way, we acquire the initial features $ll_i$ of the sample $c_i$.

## D. Model Training and Testing

As shown in Fig. 1, the model training step contains three sub-steps, i.e., graph node embedding, high-level features generation, and classification.

*1) Sub-step 1: Graph node embedding.* Actually, this sub-step aims to get a node vector through the node's semantic and type information. Given the $j$-th node of $ll_i$ with token vector $w_{i,j,t}$, $t \in [1, T]$, where $T$ is the number of tokens, its embedding vector $s_{i,j}$ equals $[se\_s_{i,j}, ty\_s_{i,j}]$ where $se\_s_{i,j}$ is the semantic information extracted from its code, $ty\_s_{i,j}$ denotes the type information in the form of one-hot, and $[\cdot]$ refers to the concatenating operation. We design two methods to get the $se\_s_{i,j}$ as follows.
① *M1.* This method produces the $se\_s_{i,j}$ via a convolutional layer, i.e.,

$$se\_s_{i,j} = Conv\left(\{w_{i,j,t}\}_{t=1}^{T}\right), \tag{2}$$

where $\{w_{i,j,t}\}_{t=1}^{T}$ is the matrix of token representation.
② *M2.* This method uses a word attention [12] to get the $se\_s_{i,j}$. First of all, we adopt a BGRU network to obtain the hidden state $h_{i,j,t}$ of $w_{i,j,t}$, i.e., $h_{i,j,t} = \left[\overrightarrow{h}_{i,j,t}, \overleftarrow{h}_{i,j,t}\right]$, where $\overrightarrow{h}_{i,j,t}$ and $\overleftarrow{h}_{i,j,t}$ mean the forward and backward hidden information of $w_{i,j,t}$ respectively.

Then, due to the different contributions of each word to $se\_s_{i,j}$, we carry out the following approach [12], i.e.,

$$\begin{aligned} u_{i,j,t} &= \tanh\left(W_w h_{i,j,t} + b_w\right), \\ score_{i,j,t} &= \frac{\exp\left(u_{i,j,t}^{\top} u_w\right)}{\sum_t \exp\left(u_{i,j,t}^{\top} u_w\right)}, \\ se\_s_{i,j} &= \sum_t score_{i,j,t}\, h_{i,j,t}. \end{aligned} \tag{3}$$

Specifically, we apply a one-layer MLP to transform $h_{i,j,t}$ into $u_{i,j,t}$, then count the attention score of words by calculating the similarity of $u_{i,j,t}$ with a word level context vector $u_w$, and normalize it through a softmax function. Finally, we compute the $se\_s_{i,j}$ as a weighted sum of the hidden representation of words.

*2) Sub-step 2: High-level features generation.* We adopt four attention-based DL models (i.e., HAN, SAN, BLSTM-att, and R-GCN-att) to generate the high-level features $hl_i$ of $c_i$, depending on the node embedding matrix $\{s_{i,j}\}_{j=1}^{L}$ where $L$ is the number of nodes in $ll_i$. Section II-E illustrates those four attention-based DL models in detail.

*3) Sub-step 3: Classification.* With the high-level features $hl_i$, we conduct the operation function for classification as,

$$\tilde{y}_i = Sigmoid\left(W_c\, hl_i + b_c\right), \tag{4}$$

where $W_c$ means the weight matrix, $b_c$ implies the bias and $\tilde{y}$ is the final predict result of $c_i$. We train the model by minimizing the loss function defined in Equation (1).

Additionally, based on the fine-tuned model after training, the step of model testing intends to evaluate the samples in the test dataset and generate the related prediction and attention results for analysis.

## *E. The Four Attention-based DL Models*

**HAN.** It is a model for document classification which confirms that different words and sentences result in diverse contributions to the semantic information of a sentence and the representation of the document respectively [12]. In this study, the DDG is regarded as a document and each node in this DDG denotes a sentence that consists of various tokens. We clarify that such tokens have different effects on the semantic representation of the correlate node. Meanwhile, different nodes in a DDG influence the result of vulnerability detection with different weights. Based on HAN, we get the semantic representation $se\_s_{i,j}$ of a node by the method $M2$ and then calculate the representation vector $hl_i$ of the sample $c_i$. In other words, we use another BGRU to create the hidden representation $h_{i,j}$ of the $j$-th node's embedding vector $s_{i,j}$, i.e., $h_{i,j} = \left[\overrightarrow{h}_{i,j}, \overleftarrow{h}_{i,j}\right]$, where $\overrightarrow{h}_{i,j}$ and $\overleftarrow{h}_{i,j}$ mean the forward and backward hidden information of $s_{i,j}$ respectively. Besides, we utilize a sentence attention to compute $hl_i$, i.e.,

$$\begin{aligned} u_{i,j} &= \tanh\left(W_s\, h_{i,j} + b_s\right), \\ score_{i,j} &= \frac{\exp\left(u_{i,j}^{\top} u_s\right)}{\sum_j \exp\left(u_{i,j}^{\top} u_s\right)}, \\ hl_i &= \sum_j score_{i,j}\, h_{i,j}, \end{aligned} \tag{5}$$

where $u_s$ is a sentence-level context vector which to measure the importance of sentences.

**SAN.** It takes the same way used by HAN to generate the high-level representation $hl_i$ of the $i$-th sample $c_i$, but it adopts the method $M1$ to acquire the semantic information $se\_s_{i,j}$ instead of the method $M2$ in HAN.

**BLSTM-att.** It aims to capture the most important semantic information in a sentence for relation classification [11]. We treat each node in a DDG as a word and such DDG as a sentence respectively. BLSTM-att firstly exploits $M1$ to get $se\_s_{i,j}$. Then, it uses a BLSTM rather than a BGRU to obtain the hidden state $h_{i,j}$ which equals to $\overrightarrow{h}_{i,j} \oplus \overleftarrow{h}_{i,j}$, where $\oplus$ refers to the sum of element-wise. Furthermore, there is also a node-level context vector $u_s$ in the BLSTM-att which is made to gain the attention score of each node. The process of getting the final $hl_i$ is described below,

$$\begin{aligned} u_{i,j} &= \tanh\left(h_{i,j}\right), \\ score_{i,j} &= \frac{\exp\left(u_{i,j}^{\top} u_s\right)}{\sum_j \exp\left(u_{i,j}^{\top} u_s\right)}, \\ temp_i &= \sum_j score_{i,j}\, h_{i,j}, \\ hl_i &= \tanh\left(temp_i\right). \end{aligned} \tag{6}$$

Note that there are certain differences between BLSTM-att and SAN, such as how to concatenate $\overrightarrow{h}_{i,j}$ and $\overleftarrow{h}_{i,j}$, and whether to adopt a MLP to deal with the $h_{i,j}$ or not.

**R-GCN-att.** The study in [6] propose an improved relation graph convolutional network (R-GCN) [24] with a triple attention mechanism to learn the vulnerability-related features. Inspired by it, we also adopt the R-GCN with a node-level attention to generate $hl_i$. Specifically, we apply $M1$ to compute $se\_s_{i,j}$ and use R-GCN to calculate $s_{i,j}$ to produce the hidden state $h_{i,j}$ by the formula,

$$h_{i,j}^{(l+1)} = \sigma\left(\sum_{r\in R}\sum_{d\in N_j^r} e_{d,j}\, W_r^{(l)}\, h_{i,d}^{(l)} + W_0^{(l)}\, h_{i,j}^{(l)}\right), \tag{7}$$

where $h_{i,j}^{(l)}$ is the hidden state of the $j$-th node of $ll_i$ at the $l$-th layer, $R$ means the set of edge types, and $d \in N_j^r$ implies that the $d$-th node is one of the members of $N_j^r$ (i.e., the neighbor set of node $j$ with respect to relation $r$). Both $W_r^{(l)}$ and $W_0^{(l)}$ denote weight matrices, where the former is related to the dependency $r$ and the latter is the self-loop weight. The $e_{d,j}$ refers to the normalizer and the $\sigma$ is defined as the activation function. In addition, the initial hidden state is set as $s_{i,j}$, i.e., $h_{i,j}^{(0)} = s_{i,j}$.

With the information aggregated by R-GCN, we employ the attention method given by [25] to calculate the attention score at the node-level [6]. It is formulated as,

$$\begin{aligned} \alpha_{i,j} &= \sigma\left(\mathrm{R-GCN}\left(\mathrm{X}, \mathrm{A}\right)\right), \\ score_{i,j} &= \frac{\exp\left(\alpha_{\mathrm{i,j}}\right)}{\sum_j \exp\left(\alpha_{\mathrm{i,j}}\right)}, \\ hl_i &= \sum_j score_{i,j}\, h_{i,j}^{(l\_num)}, \end{aligned} \tag{8}$$

where $X$ is the node feature matrix $\left\{h_{i,j}^{(l\_num)}\right\}_{j=1}^{L}$ that represents the feature information, $A$ denotes the adjacency matrix which describes the topology information, $h_{i,j}^{(l\_num)}$ is considered as the final hidden state of node $j$ and $l\_num$ means the number of layers. We add another R-GCN to obtain the attention weights by working on both the features of the nodes themselves and the topology of the graph. At last, we implement softmax function to normalize the attention weights and get the high-level feature $hl_i$ through summing the weighted over $X$.

## III. Experiments

### A. Evaluation Setup

*1) Juliet+:* Juliet contains 112 Common Weakness Enumeration (CWE) [26] entries. As only the part of CWEs are vulnerability-related and contain vulnerable data dependence flow, we choose 70 CWEs by checking manually at first. We also perform a deduplication operation for the normalized corpus to avoid the problem of data duplication [5]. Thus, we get 93,689 samples which include 27,813 bad samples and 65,876 good samples before normalization. After normalization, we preserve 29,990 samples of which 16,051 samples are vulnerable and 13,939 samples are non-vulnerable. Note that they are only non-repeating in sequence (see examples on our website). We randomly shuffle these deduplicated samples and select 75% of them as the training set and the rest as the test set while keeping the rate of vulnerable samples versus non-vulnerable samples in every CWE entry. The specific processes of data collection and labeling are presented in Section II-C.

*2) Evaluation configuration:* We use Pytorch and Deep Graph Library [27] to implement the four DL models. The dimension of embedding vector for each tokens is set as 50. Since there are 7 types of nodes and the size of semantic information $se\_se_{i,j}$ is 100, we adopt 107 as the size of the node embedding vector $s_{i,j}$. The length of the high-level features $hl_i$ for all models is 200, except the BLSTM-att with length of 100. Meanwhile, the values of $u_w$ and $u_s$ are defined as 50 and 100 respectively. We train all four models by using an Adam optimizer [28] with a learning rate of 0.001. Early stopping is also applied to prevent overfitting and decrease the training time. Additionally, for the sequence-based models, we choose 100 as the patience of epoch that is different from the 200 in R-GCN-att in order to save training time. All experiments run on a machine with an Intel Core 3.6 GHz CPU and an NVIDIA GeForce RTX-2060 GPU.

*3) Evaluation metrics:* First of all, the prediction performance represents whether a model can predict a sample is vulnerable or not accurately. Therefore, we employ six widely-used metrics [29] to evaluate the prediction performance of DL models, i.e., *accuracy* (**A**), *precision* (**P**), *recall* (**R**), *F1-score* (**F1**), *false positive rate* (**FPR**), *false negative rate* (**FNR**). Meanwhile, the attention effects represent the interpretability of DL models. Triggered by the metrics used for link prediction in knowledge graphs [30], we use **Hit@k**, **Hit@k%** to evaluate the attention effects. **Hit@k** denotes whether the attention score rankings of the vulnerability-related nodes in all nodes of $ll_i$ are in the top-k nodes. But it cannot adapt to samples with different numbers of nodes. To make up for this deficiency, we also adopt **Hit@k%**, which is rougher than **Hit@k**, to indicate the vulnerability-related nodes that are ranked in the top-k% of all nodes of $ll_i$.

TABLE I: The selected CWE entries with class distribution in the test set.

| CWE | #Samples | #Vul Samples | #Non-Vul Samples | #Vul : #Non-Vul |
|---|---|---|---|---|
| CWE22 | 166 | 138 | 28 | 5 |
| CWE79 | 263 | 227 | 36 | 6 |
| CWE89 | 629 | 315 | 314 | 1 |
| CWE190 | 1,364 | 606 | 758 | 0.8 |
| CWE191 | 1,088 | 483 | 605 | 0.8 |

*4) Hit type:* Corresponding to the strategies of labeling the vulnerability-related code lines, there are three types of vulnerability-related nodes, i.e., “bad_source” and “bad_sink” for the bad samples, and “fixed” for the good ones. Note that there is one vulnerability-related node in the average 8 nodes for bad samples while 16 nodes for good samples. Besides, for almost every sample in Juliet+, there are only one “bad_source” node and one “bad_sink” node for a bad sample, and only one “fixed” node for a good sample. We further design five hit types to represent the attention performances of DL models in different vulnerability-related node types. To be specific, “fixed”, “bad_source”, and “bad_sink” denote that the correlated vulnerability-related nodes are hit respectively. Moreover, the hit type “fixed” also represents the ability of a model to learn non-vulnerable patterns (distinguish vulnerable patterns from non-vulnerable patterns). Additionally, for a bad sample, “bad_either” means either “bad_source” nodes or “bad_sink” nodes are hit representing the ability to capture the vulnerable patterns, while “bad_avg” implies the average rank of them which reveals the ability of a model to obtain the context-dependency of software vulnerability.

### B. Experiments and Discussions

We conduct three research questions (RQs) to analyze the prediction results (RQ1) and interpretability (RQ2, RQ3) of different attention-based DL models. Moreover, RQ2 discusses the detailed attention performance and RQ3 investigates the factors that affect the attention effects of the GNN-based DL model R-GCN-att which achieves the best interpretability. Note that we train each DL model ten times and calculate the average results as a measure to reduce the deviation.

*1) RQ 1:* **How do the prediction performances of the proposed four DL models for software vulnerability?**

**Motivation.** Similar to the studies [1], [2], [6] that have achieved high prediction results on the synthetic vulnerability dataset of C or C++, we aim at exploring the prediction performance of DL models with the manually created synthetic vulnerability dataset Juliet+ (Java) in this study. We clarify that it is the first step to achieve high prediction results before we further evaluate the interpretability of such DL models.

**Result.** Table II presents the prediction results of the four DL models. All of them obtain a high performance reaching larger than 95% on the accuracy, precision, recall, and F1-score. The relevant values of false positive rate and false negative rate are low as well. We also notice that the differences of all prediction evaluation metrics are marginal (e.g., the changes of accuracy, precision, recall, and F1-score are about 2.5%).

TABLE II: The prediction results of the four models.

| Model | A(%) | P(%) | R(%) | F1(%) | FPR(%) | FNR(%) |
|---|---|---|---|---|---|---|
| HAN | **98.18** | **98.37** | **98.22** | **98.29** | **0.02** | **0.02** |
| SAN | 97.51 | 97.81 | 97.54 | 97.67 | 0.03 | **0.02** |
| BLSTM-att | 96.67 | 97.35 | 96.40 | 96.87 | 0.03 | 0.04 |
| R-GCN-att | 95.71 | 95.44 | 96.80 | 95.11 | 0.06 | 0.04 |

The reason is that the vulnerabilities in Juliet+ are far less complicated than the real-world vulnerabilities.

Among the three sequence-based models, HAN gets the best performance because it inspects the tokens (at the token level) and distinguishes the most critical ones related to the semantics of the nodes. The result of SAN outperforms that of BLSTM-att because they have different ways to generate high-level features. SAN uses BGRU and a MLP layer to obtain better high-level features than BLSTM-att to represent the vulnerability patterns.

The R-GCN-att, which adopts the graph structures of the source code as input containing more logical and structural information instead of the sequenced-based input, obtains a lower result (95.71% accuracy). This refers to that there are still a small number of duplicated samples in the graph structures and their labels with conflicts may mislead the R-GCN-att, although the dataset obtained by operation of deduplication is no longer repeated in the sequence based on the corpus after normalization. In this study, we ignore the effects of these conflicted samples because our critical target is to explore the interpretability of DL models rather than to compare the prediction results of such models. Meanwhile, we find that only a few samples conflict in the graph structures so that it does not fundamentally affect the conclusions.

**Answer to RQ1:** All the four attention-based DL models achieve high performance (more than 95% on accuracy, precision, recall, and F1-score) when predicting the software vulnerability of the samples in the test dataset of Juliet+.

*2) RQ 2:* **Can the DL models learn the vulnerable patterns? How do they perform in terms of attention effects?**

**Motivation.** As the main target in this study is to verify the interpretability of DL models for vulnerability detection based on the attention effects of the proposed four models, we investigate how the attention effects of the four DL models perform in terms of different hit types. Moreover, we want to know whether the GNN-based R-GCN-att outperforms the other sequence-based models in this RQ.

**Result.** Table III depicts the attention effects of the four DL models. HAN achieves the best performance among the three sequence-based models, while BLSTM-att is the worst. This is consistent with the prediction performance of the three sequence-based models in RQ1. Specifically, on the hit type of "fixed", BLSTM-att only acquires 27.2%, 9.47%, 40.51%, and 25.02% on Hit@3, Hit@1, Hit@50%, and Hit@30% respectively which are obviously lower than the relevant values of HAN (e.g., 32.80% lower on Hit@50%). This denotes that BLSTM-att cannot understand the differences between vulnerable and non-vulnerable patterns although it results in a high prediction result on Juliet+ (see in Table II). However, on the hit type "bad_either", the differences in the attention results of the three sequence-based models are relatively small (e.g., the biggest gap is 7.60% on Hit@50%). This denotes that all three of them have a similar ability to learn vulnerable patterns.

On the other hand, R-GCN-att obtains the best attention effects which are contrary to its prediction performance. Specifically, on the hit type of "fixed", R-GCN-att reaches 86.27% at Hit@50% and 76.73% at Hit@30%, which are 27.54% and 34.3% higher than the other three models on average. This illuminates that R-GCN-att discriminates the differences between vulnerable and non-vulnerable patterns more clearly. In addition, on the hit type of "bad_avg", R-GCN-att gives 53.65% at Hit@50% and 18.44% at Hit@30% respectively which are 15.38% and 13.15% higher than BLSTM-att. This also uncovers that R-GCN-att better captures the context-dependency of the vulnerable patterns.

Additionally, for the hit types of "bad_either", such four DL models of vulnerability detection yield 82.76% and 58.65% on average at the metric Hit@50% and Hit30%, which implies that, to some extent, those DL models can learn the vulnerable patterns of the dataset Juliet+ on the whole.

**Answer to RQ2:** The DL models can learn vulnerable patterns to a certain extent for interpretation on Juliet+. However, there are definitely differences across the four attention-based DL models. The GNN-based DL model R-GCN-att obtains the best attention effects while the DL model BLSTM-att cannot learn the differences between vulnerable and non-vulnerable patterns well, although they have similar results for predicting the software vulnerability.

*3) RQ3:* **How effective are the attention effects of R-GCN-att in different CWEs?**

**Motivation.** The answer to RQ2 tells us that the characteristics of the models can affect their interpretability. In RQ3, we want to validate how the DL model R-GCN-att which has the best interpretability performs on the attention effects of different CWEs. Moreover, we want to estimate what other facts affect the interpretability of R-GCN-att according to the characteristics of those CWEs. Only part of CWEs are picked out to be analyzed due to two reasons. First, according to the popularity of vulnerabilities existing in the Java program, we choose CWE89, the vulnerability of SQL Injection, along with CWE22 about Path Traversal (including CWE23 and CWE36) and CWE79 about Cross-Site Scripting (including CWE80, CWE81, and CWE83) [31]. Second, based on the distribution of the sample number, we select the two most numerous CWEs, i.e., CWE190 about Integer Overflow and CWE191 about Integer Underflow. Table I lists the statistics of such five CWEs in the test set after normalization.

TABLE III: The attention effects of the four models.

| Metric | Hit@3(%) | | Hit@1(%) | | Hit@50%(%) | | | Hit@30%(%) | | |
|---|---|---|---|---|---|---|---|---|---|---|
| **Hit_type** | **fixed** | **bad_either** | **fixed** | **bad_either** | **fixed** | **bad_avg** | **bad_either** | **fixed** | **bad_avg** | **bad_either** |
| HAN | 52.29 | 51.81 | 18.21 | 24.14 | 73.31 | 46.36 | **87.26** | 55.72 | 10.21 | 61.40 |
| SAN | 47.53 | 51.90 | 19.98 | 28.43 | 62.37 | 39.93 | 81.36 | 46.55 | 5.89 | 55.99 |
| BLSTM-att | 27.20 | 46.01 | 9.47 | 17.79 | 40.51 | 38.27 | 79.66 | 25.02 | 5.29 | 53.97 |
| R-GCN-att | **77.14** | **58.42** | **56.56** | **29.00** | **86.27** | **53.65** | 82.74 | **76.73** | **18.44** | **63.22** |

**Result.** Fig. 4 shows the attention effects of R-GCN-att on the five CWEs. For the attention effects on the type of "fixed", the model performs the best on CWE190 and CWE191 obviously and the worst on CWE89. Furthermore, Table I confirms that CWE190 and CWE191 contain the most sample, so R-GCN-att is well-trained to learn the non-vulnerable patterns. However, it is difficult for R-GCN-att to gain a high result on CWE89, although it has more samples than CWE22 and CWE79. We further observe that there is only one way to repair the software vulnerability for CWE22 and CWE79 by using "goodsource". However, CWE89 uses "good_source" or "good_sink" to repair itself which makes it harder to learn the repair patterns.

For the attention effects on "bad_avg" which represents the ability of one model to capture the context of vulnerable patterns, R-GCN-att reaches its best performance on CWE89 because it not only maintains a balanced class distribution but also contains a great number of vulnerable samples.

We also observe that the attention effects of R-GCN-att on "bad_source" are larger than those on "bad_sink" for all of the five CWEs. This illuminates that the model puts more emphasis on the "bad_source" nodes instead of the "bad_sink" nodes of the bad samples in Juliet+. Moreover, we discover that it needs to change more lines of codes when repairing "badsource" than that of repairing "badsink" which provides more contrast features.

Finally, for the attention effects on "bad_either", R-GCN-att does not perform well on CWE190 and CWE191 where the vulnerable samples are less than non-vulnerable samples.

**Answer to RQ3:** The attention effects of R-GCN-att on different CWEs are affected by various aspects such as the number of samples, the class distribution, and the differences of sample features. We propose to train the R-GCN-att model to fulfill the goal of better interpretability by a dataset that has balanced and sufficient samples with obvious differences between vulnerable and non-vulnerable patterns.

## IV. Threats to Validity

The presented framework in this paper is subject to some limitations that could potentially threaten our experimental results and relevant findings.

An external threat is the quality and representativeness of the dataset Juliet+ created based on Juliet. Juliet has also been used in existing studies about vulnerabilities analysis [4], [31]. We thus believe the Juliet+ should be representative and its quality should be good as well. Although there are limitations about Juliet+ such as the synthetic characteristics, we can still observe how the attention-based DL model focuses on the key vulnerable patterns and explain the classification result of software vulnerability prediction. Furthermore, we can further confirm our findings on a real-world software vulnerability dataset when it becomes available.

An internal threat is experimental settings that we compute the average results via multiple same experiments. We train each DL model ten times to generate the average result of every metric, which is sufficient to qualitatively observe the overall performance of all models and keep the result as stable as possible. In addition, we record the attention effect and use it to explain the interpretability of the DL models only when the models reach the highest F1-score. This setting may affect some experiment results because the attention effects may change when the prediction effects on other prediction metrics (e.g., accuracy) still increase. However, it should not fundamentally affect our conclusions, especially those for the interpretability of the DL models.

Another internal threat comes from the attention mechanism used to explain the result of software vulnerability detection. Actually, there are still many discussions on whether we should choose attention mechanism as an interpretation method to analyze and understand the behaviors (or results) of DL models [32]–[34]. However, Wiegreffe et al. [35] suggest that the attention mechanism is meaningful especially when it works in coordination with the entire DL model. Meanwhile, triggered by the effectiveness of attention for interpretation in Xmal [36], we also implement the attention into all of the four DL models in our experiments and clarify the interpretability of such attention-based DL models on the issue of software vulnerability detection.

## V. Related Work

**Vulnerability detection.** In existing studies on vulnerability detection, most efforts are devoted to designing rules [37], [38], machine learning [39] or deep learning [1]–[7] to learn the vulnerable patterns. Engler et al. [37] summarized six template checkers such as "<a> must be paired with <b>". Shin et al. [39] made an effort to use code complexity metrics when using conventional machine learning to detect software vulnerability. Deep learning methods like LSTM [40], GRU [41] and GGNN [42] are also adopted to detection because they automatically extract vulnerability features to reduce

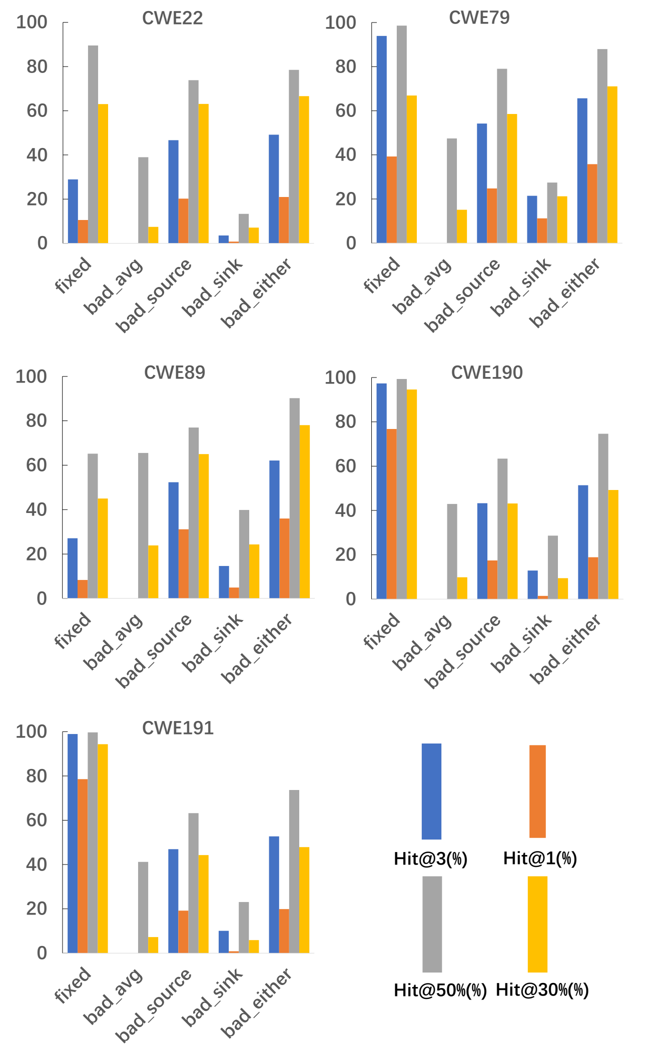


Fig. 4: The attention effects of R-GCN-att on CWE22, CWE79, CWE89, CWE190, and CWE191.

the efforts of security experts [1], [3]. However, most DL methods neglect the interpretability of predicted results which is important to understand vulnerable patterns of the source code [9].

**Model interpretation.** Although model interpretation has been spread to many security-related fields such as malware detection [36], [43] and security applications [44], [45], there are few studies [46]–[48] devoted to the interpretation of vulnerability detection. Zou et al. [46] adopt a custom heuristic search to get the important tokens that are used to generate some vulnerability rules and use fidelity to evaluate the interpretability. Ganz et al. [47] used nine common graph-agnostic and three graph-specific explanation methods to explain graph neural networks for vulnerability detection with some self-defined criteria. However, neither of them intuitively demonstrates interpretability by comparing human annotation results with model results. The study in [48] is the most similar one with us where it also uses attention to evaluate the interpretability by comparing the code lines labeled automatically and the most relevant code lines to the prediction given by the model. Different from such three jobs above, we not only manually mark the vulnerability-related lines of code for samples in Juliet+, but also divide them into more detailed types. Moreover, our evaluation is also more intuitive by observing the vulnerability-related codes being hit.

## VI. Conclusion

In this study, we propose a framework to investigate the interpretability of attention-based DL models for the problem of software vulnerability detection. Based on a manually created dataset Juliet+ under a great effort on labeling the vulnerability-related code lines, we design four models to explore their performance of detection and to understand which model better recognizes the critical code lines to represent vulnerable patterns of the source code. The empirical results show that all of the four models maintain a high detection performance that is more than 95% F1-score. However, only the GNN-based DL model R-GCN-att gains an acceptable result (86.27% on the hit type of "fixed" with metric Hit@50%) when interpreting the prediction of vulnerability. This validates that the R-GCN-att has learned the critical knowledge to distinguish the vulnerable patterns from the graph structure of source code. Additionally, we analyze the facts that may influence the attention effects of DL models. It is confirmed that the R-GCN-att model achieves the better attention effects (interpretability) of vulnerability detection when the dataset constructed consists of more balanced and sufficient samples with distinct differences between vulnerable and non-vulnerable patterns.

## Acknowledgments

This work has partially been sponsored by the National Natural Science Foundation of China (Grant No. 62102284 and No. 61872262).

## References

[1] Zhen Li, Deqing Zou, Shouhuai Xu, Xinyu Ou, Hai Jin, Sujuan Wang, Zhijun Deng, and Yuyi Zhong. Vuldeepecker: A deep learning-based system for vulnerability detection. *arXiv preprint arXiv:1801.01681*, 2018.

[2] Zhen Li, Deqing Zou, Shouhuai Xu, Hai Jin, Yawei Zhu, and Zhaoxuan Chen. Sysevr: A framework for using deep learning to detect software vulnerabilities. *IEEE Transactions on Dependable and Secure Computing (TDSC)*, 2021.

[3] Yaqin Zhou, Shangqing Liu, Jingkai Siow, Xiaoning Du, and Yang Liu. Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks. *arXiv preprint arXiv:1909.03496*, 2019.

[4] Huanting Wang, Guixin Ye, Zhanyong Tang, Shin Hwei Tan, Songfang Huang, Dingyi Fang, Yansong Feng, Lizhong Bian, and Zheng Wang. Combining graph-based learning with automated data collection for code vulnerability detection. *IEEE Transactions on Information Forensics and Security (TIFS)*, 16:1943–1958, 2020.

[5] Saikat Chakraborty, Rahul Krishna, Yangruibo Ding, and Baishakhi Ray. Deep learning based vulnerability detection: Are we there yet. *IEEE Transactions on Software Engineering (TSE)*, 2021.

[6] Weining Zheng, Yuan Jiang, and Xiaohong Su. Vulspg: Vulnerability detection based on slice property graph representation learning. *arXiv preprint arXiv:2109.02527*, 2021.

[7] Song Wang, Taiyue Liu, and Lin Tan. Automatically learning semantic features for defect prediction. In *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*, pages 297–308. IEEE, 2016.

[8] Geoffrey E Hinton, Simon Osindero, and Yee-Whye Teh. A fast learning algorithm for deep belief nets. *Neural computation*, 18(7):1527–1554, 2006.

[9] Guanjun Lin, Sheng Wen, Qing-Long Han, Jun Zhang, and Yang Xiang. Software vulnerability detection using deep neural networks: A survey. *Proceedings of the IEEE (Proc.IEEE)*, 108(10):1825–1848, 2020.

[10] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate. *arXiv preprint arXiv:1409.0473*, 2014.

[11] Peng Zhou, Wei Shi, Jun Tian, Zhenyu Qi, Bingchen Li, Hongwei Hao, and Bo Xu. Attention-based bidirectional long short-term memory networks for relation classification. In *Proceedings of the 54th annual meeting of the association for computational linguistics (volume 2: Short papers)*, pages 207–212, 2016.

[12] Zichao Yang, Diyi Yang, Chris Dyer, Xiaodong He, Alex Smola, and Eduard Hovy. Hierarchical attention networks for document classification. In *Proceedings of the 2016 conference of the North American chapter of the association for computational linguistics: human language technologies*, pages 1480–1489, 2016.

[13] Thomas Zenkel, Joern Wuebker, and John DeNero. Adding interpretable attention to neural translation models improves word alignment. *arXiv preprint arXiv:1901.11359*, 2019.

[14] Guisheng Fan, Xuyang Diao, Huiqun Yu, Kang Yang, and Liqiong Chen. Software defect prediction via attention-based recurrent neural network. *Scientific Programming (SP)*, 2019, 2019.

[15] Guanjun Lin, Wei Xiao, Jun Zhang, and Yang Xiang. Deep learning-based vulnerable function detection: A benchmark. In *International Conference on Information and Communications Security (ICICS)*, pages 219–232. Springer, 2019.

[16] René Just, Darioush Jalali, and Michael D Ernst. Defects4j: A database of existing faults to enable controlled testing studies for java programs. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, pages 437–440, 2014.

[17] Rebecca Russell, Louis Kim, Lei Hamilton, Tomo Lazovich, Jacob Harer, Onur Ozdemir, Paul Ellingwood, and Marc McConley. Automated vulnerability detection in source code using deep representation learning. In *2018 17th IEEE international conference on machine learning and applications (ICMLA)*, pages 757–762. IEEE, 2018.

[18] Paul E Black et al. Samate's contribution to information assurance. *NIST Special Publication*, 500(264):2, 2006.

[19] Juliet test suite v1.2 for java user guide. https://samate.nist.gov/SRD/resources/Juliet_Test_Suite_v1.2_for_Java_-_User_Guide.pdf, 2012.

[20] Nicholas Smith, Danny Van Bruggen, and Federico Tomassetti. Javaparser: visited. *Leanpub, oct. de*, 2017.

[21] Jeanne Ferrante, Karl J Ottenstein, and Joe D Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 9(3):319–349, 1987.

[22] Rafael-Michael Karampatsis, Hlib Babii, Romain Robbes, Charles Sutton, and Andrea Janes. Big code!= big vocabulary: Open-vocabulary models for source code. In *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*, pages 1073–1085. IEEE, 2020.

[23] word2vec. http://radimrehurek.com/gensim/models/word2vec.html, 2021.

[24] Michael Schlichtkrull, Thomas N Kipf, Peter Bloem, Rianne Van Den Berg, Ivan Titov, and Max Welling. Modeling relational data with graph convolutional networks. In *European semantic web conference*, pages 593–607. Springer, 2018.

[25] Junhyun Lee, Inyeop Lee, and Jaewoo Kang. Self-attention graph pooling. In *International Conference on Machine Learning (ICML)*, pages 3734–3743. PMLR, 2019.

[26] Common weakness enumeration. https://samate.nist.gov/SRD/index.php, 2021.

[27] Minjie Wang, Lingfan Yu, Da Zheng, Quan Gan, Yu Gai, Zihao Ye, Mufei Li, Jinjing Zhou, Qi Huang, Chao Ma, et al. Deep graph library: Towards efficient and scalable deep learning on graphs. 2019.

[28] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[29] Marcus Pendleton, Richard Garcia-Lebron, Jin-Hee Cho, and Shouhuai Xu. A survey on systems security metrics. *ACM Computing Surveys (CSUR)*, 49(4):1–35, 2016.

[30] Yongqi Zhang, Quanming Yao, Yingxia Shao, and Lei Chen. Nscaching: simple and efficient negative sampling for knowledge graph embedding. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*, pages 614–625. IEEE, 2019.

[31] Jiayi Hua and Haoyu Wang. On the effectiveness of deep vulnerability detectors to simple stupid bug detection. In *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*, pages 530–534. IEEE, 2021.

[32] Sarthak Jain and Byron C Wallace. Attention is not explanation. *arXiv preprint arXiv:1902.10186*, 2019.

[33] Sofia Serrano and Noah A Smith. Is attention interpretable? *arXiv preprint arXiv:1906.03731*, 2019.

[34] Jasmijn Bastings and Katja Filippova. The elephant in the interpretability room: Why use attention as explanation when we have saliency methods? *arXiv preprint arXiv:2010.05607*, 2020.

[35] Sarah Wiegreffe and Yuval Pinter. Attention is not not explanation. *arXiv preprint arXiv:1908.04626*, 2019.

[36] Bozhi Wu, Sen Chen, Cuiyun Gao, Lingling Fan, Yang Liu, Weiping Wen, and Michael Lyu. Why an android app is classified as malware? towards malware classification interpretation. In *CoRR*, volume abs/2004.11516, 2020.

[37] Dawson Engler, David Yu Chen, Seth Hallem, Andy Chou, and Benjamin Chelf. Bugs as deviant behavior: A general approach to inferring errors in systems code. *ACM SIGOPS Operating Systems Review*, 35(5):57–72, 2001.

[38] Flawfinder. https://dwheeler.com/flawfinder/, 2013.

[39] Yonghee Shin and Laurie Williams. An empirical model to predict security vulnerabilities using code complexity metrics. In *Proceedings of the Second ACM-IEEE international symposium on Empirical software engineering and measurement*, pages 315–317, 2008.

[40] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[41] Kyunghyun Cho, Bart Van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning phrase representations using rnn encoder-decoder for statistical machine translation. *arXiv preprint arXiv:1406.1078*, 2014.

[42] Yujia Li, Daniel Tarlow, Marc Brockschmidt, and Richard Zemel. Gated graph sequence neural networks. *arXiv preprint arXiv:1511.05493*, 2015.

[43] Ming Fan, Wenying Wei, Xiaofei Xie, Yang Liu, Xiaohong Guan, and Ting Liu. Can we trust your explanations? sanity checks for interpreters in android malware analysis. *IEEE Transactions on Information Forensics and Security (TIFS)*, 16:838–853, 2020.

[44] Marco Tulio Ribeiro, Sameer Singh, and Carlos Guestrin. "why should i trust you?" explaining the predictions of any classifier. In *Proceedings of the 22nd ACM SIGKDD international conference on knowledge discovery and data mining*, pages 1135–1144, 2016.

[45] Wenbo Guo, Dongliang Mu, Jun Xu, Purui Su, Gang Wang, and Xinyu Xing. Lemna: Explaining deep learning based security applications. In *proceedings of the 2018 ACM SIGSAC conference on computer and communications security*, pages 364–379, 2018.

[46] Deqing Zou, Yawei Zhu, Shouhuai Xu, Zhen Li, Hai Jin, and Hengkai Ye. Interpreting deep learning-based vulnerability detector predictions based on heuristic searching. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 30(2):1–31, 2021.

[47] Tom Ganz, Martin Härterich, Alexander Warnecke, and Konrad Rieck. Explaining graph neural networks for vulnerability discovery. In *Proceedings of the 14th ACM Workshop on Artificial Intelligence and Security*, pages 145–156, 2021.

[48] Zhen Li, Deqing Zou, Shouhuai Xu, Zhaoxuan Chen, Yawei Zhu, and Hai Jin. Vuldeelocator: a deep learning-based fine-grained vulnerability detector. *IEEE Transactions on Dependable and Secure Computing (TDSC)*, 2021.