DeepWukong: Statically Detecting Software Vulnerabilities using Deep Graph Neural Network

XIAO CHENG, HAOYU WANG*, JIAYI HUA, and GUOAI XU*, Beijing University of Posts and Telecommunications, China
YULEI SUI, University of Technology Sydney, Australia

Static bug detection has shown its effectiveness in detecting well-defined memory errors, e.g., memory leaks, buffer overflows and null dereference. However, modern software systems have a wide variety of vulnerabilities. These vulnerabilities are extremely complicated with sophisticated programming logic and these bugs are often caused by different bad programming practices, challenging existing bug detection solutions. It is hard and labour-intensive to develop precise and efficient static analysis solutions for different types of vulnerabilities, particularly for those that may not have a clear specification as the traditional well-defined vulnerabilities.

This paper presents DeepWukong, a new deep-learning-based embedding approach to static detection of software vulnerabilities for C/C++ programs. Our approach makes a new attempt by leveraging advanced recent graph neural networks to embed code fragments in a compact and low-dimensional representation, producing a new code representation that preserves high-level programming logic (in the form of control- and data-flows) together with the natural language information of a program. Our evaluation studies the top 10 most common C/C++ vulnerabilities during the past three years. We have conducted our experiments using 105,428 real-world programs by comparing our approach with four well-known traditional static vulnerability detectors and three state-of-the-art deep-learning-based approaches. The experimental results demonstrate the effectiveness of our research and have shed light on the promising direction of combining program analysis with deep learning techniques to address the general static code analysis challenges.

CCS Concepts: • **Security and privacy → Software security engineering**; • **Computing methodologies → Machine learning**; • **Software and its engineering → Automated static analysis**.

Additional Key Words and Phrases: Static analysis, graph embedding, vulnerabilities

ACM Reference Format:
Xiao Cheng, Haoyu Wang, Jiayi Hua, Guoai Xu, and Yulei Sui. 2020. DeepWukong: Statically Detecting Software Vulnerabilities using Deep Graph Neural Network. *ACM Trans. Softw. Eng. Methodol.* 1, 1, Article 1 (January 2020), 32 pages. https://doi.org/10.1145/3436877

1 INTRODUCTION

Modern software systems are often plagued with a wide variety of software vulnerabilities. Detecting and fixing these complicated, emerging and wide-ranging vulnerabilities are extremely hard. The number of vulnerabilities registered in the Common Vulnerabilities and Exposures (CVE) [1] has

*Haoyu Wang and Guoai Xu are co-corresponding authors.

Authors' addresses: Xiao Cheng, jackiecheng@bupt.edu.cn; Haoyu Wang, haoyuwang@bupt.edu.cn; Jiayi Hua, huajiayi@bupt.edu.cn; Guoai Xu, xga@bupt.edu.cn, Beijing University of Posts and Telecommunications, No 10, Xitucheng Road, Haidian District, Beijing, China, 100876; Yulei Sui, yulei.sui@uts.edu.au, University of Technology Sydney, 15 Broadway, Sydney, NSW, Australia, 2006.

1049-331X/2020/1-ART1 $15.00
https://doi.org/10.1145/3436877

Table 1. Top 10 most common C/C++ CVEs (in terms of their numbers appear in NVD [1]) from 2017.01.01-2019.07.20.

Vul Category	Description	CVE-num
CWE119	Improper Restriction of Operations within the Bounds of a Memory Buffer	4,524
CWE20	Improper Input Validation	2,769
CWE125	Out-of-bounds Read	1,245
CWE190	Integer Overflow or Wraparound	877
CWE22	Improper Limitation of a Pathname to a Restricted Directory	857
CWE399	Resource Management Errors	699
CWE787	Out-of-bounds Write	534
CWE254	Security Features	483
CWE400	Uncontrolled Resource Consumption	393
CWE78	Improper Neutralization of Special Elements used in an OS Command	258

increased significantly during the past three years. Statistics show that there are 34,473 newly registered CVEs and 34,093 of them are above medium-level security from 2017.01.01 to 2019.07.20 according to NVD [1]. Table 1 lists the number of CVEs for the top 10 most Common C/C++ Weakness Enumeration (CWE) vulnerabilities. It can be seen that these vulnerabilities behave differently with quite different specifications.

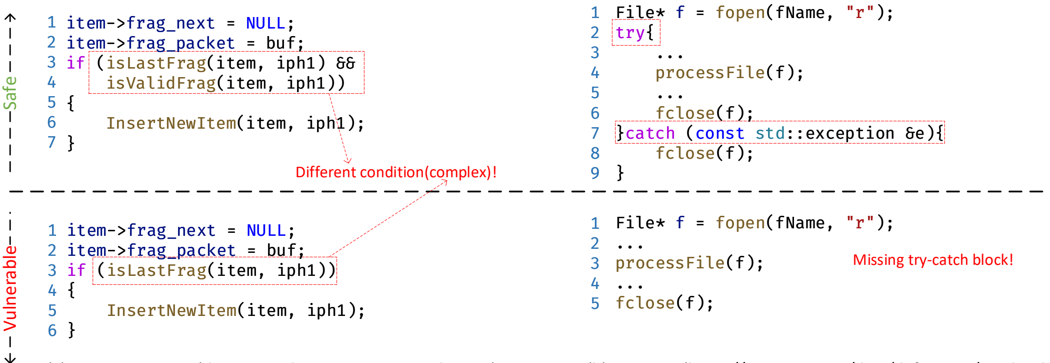
Existing efforts. Static bug detection, which approximates the runtime behavior of a program without running it, is the major way to pinpoint bugs at the early stage of software development cycle, thus reducing software maintenance cost. Traditional static analysis techniques (e.g., Clang Static Analyzer [2], Coverity [3], Fortify [4], Flawfinder [5], Infer [6], ITS4 [7], RATS [8], Checkmarx [9] and SVF [10]) have shown their success in detecting well-defined memory corruption bugs. However, adapting the existing solutions for detecting a wide variety of emerging vulnerabilities has two major limitations. First, they rely on static analysis experts to define specific detection strategies for different types of vulnerabilities, which is labour-intensive and time-consuming. Second, the effectiveness of the pre-developed detection systems highly relies on the expertise of the analysis developers and the knowledge of existing vulnerabilities. The emerging high-level vulnerabilities pose big challenges to existing bug detection approaches, making it hard to extend the existing bug detectors.

Challenges. We give two real-world vulnerabilities to demonstrate the challenges in identifying different high-level vulnerabilities. Fig. 1(a) shows a vulnerable code fragment from *IPsec-Tools 0.8.2*[1]. It contains a remotely exploitable attack when parsing and storing ISAKMP fragments when missing a conditional guard. The code below the black dotted line is vulnerable and the above one is safe (after it was fixed by the developer). Most of the two code fragments share the same logic (omitted here), but they are different in checking the last ISAKMP fragment. Unfortunately, the conditional check in the vulnerable code is incorrect and inconsistent with many other parts of the code in this project. This vulnerability took an experienced developer 23 days to eventually find and fix it[2]. Fig. 1(b) shows an example of improper resource shutdown or release[3]. The vulnerable code fragment does not close an opened file handle if an error occurs. If this is a long running process, it can run out of file handle resources; therefore, missing the "try/catch" exceptional handling may

[1] The whole fixing patch is available at http://cvsweb.netbsd.org/bsdweb.cgi/src/crypto/dist/ipsec-tools/src/racoon/isakmp_frag.c.diff?r1=1.5&r2=1.5.36.1

[2] https://nvd.nist.gov/vuln/detail/CVE-2016-10396

[3] The detailed description is available at https://cwe.mitre.org/data/definitions/404.html



```
item->frag_next = NULL;
item->frag_packet = buf;
if (isLastFrag(item, iph1) &&
    isValidFrag(item, iph1))
{
    InsertNewItem(item, iph1);
}
```

```
item->frag_next = NULL;
item->frag_packet = buf;
if (isLastFrag(item, iph1))
{
    InsertNewItem(item, iph1);
}
```

(a) CVE-2016-10396 (the racoon daemon in IPsec-Tools 0.8.2)

```
File* f = fopen(fName, "r");
try{
    ...
    processFile(f);
    ...
    fclose(f);
}catch (const std::exception &e){
    fclose(f);
}
```

```
File* f = fopen(fName, "r");
...
processFile(f);
...
fclose(f);
```

(b) CWE-404 (https://cwe.mitre.org/data/definitions/404.html)

Fig. 1. Real-world vulnerabilities that are hard to be automatically identified by traditional static vulnerability detection approaches. For each example, we list the safe code fragment and the corresponding identified vulnerable code.

cause program crashes or hang. These high-level bugs are caused by inconsistent business logic and bad programming practices; therefore, it is challenging for traditional static detectors (e.g., memory error detectors) which rely on well-defined specifications to capture these vulnerabilities.

Insights. In reality, the unexpected behaviors of a vulnerable program often manifest in different aspects of the code features, including code tokens, APIs, and control- and data-flow of a program. The code pattern can be quite different between vulnerable and safe code because it can reflect the interprocedural execution order, the logic of a program, and also good/bad programming practices. There are a handful of efforts in pinpointing vulnerabilities at different levels of granularity (e.g. program [11], package [12], component [13], file [14], method [15–17] and slice [18]) by combining machine learning with static bug detection. The general idea is to generate a prediction model that captures the correlation between vulnerable programs and their (extracted) program features through sample training. Later, a new program can be predicted as safe or vulnerable based on the trained prediction model.

However, almost all of them focus on detecting low-level memory errors, e.g., buffer overflows and use-after-frees. The real-world vulnerabilities are much more complicated (cf. Figure 1), posing challenges for existing approaches. Moreover, current approaches extract shallow code features (e.g., code tokens and abstract syntax trees) by directly applying the embedding techniques, such as Word2Vec [19] and Doc2Vec [20], while the comprehensive code features (e.g., control- and data-dependence) are not precisely preserved in the embedding space. Allamanis et al. [21] have proposed to use gated graph neural network to represent the syntactic and semantic structure of a program, aiming at solving two software engineering tasks, i.e., variable renaming and misuses, rather than detecting general software vulnerabilities. In addition, they did not perform interprocedural program dependence analysis (by considering pointer aliases), which is essential for real-world programs since function calls are quite common in modern software projects. Finally, existing deep-learning-based approaches predict a bug at method or file level in a coarse grained manner. It is hard to provide a more fine-grained way (e.g., identifying the program slices that may trigger a bug) for developers to precisely pinpoint a vulnerability.

Our solution. In this paper, we propose DeepWukong, a new deep-learning-based code embedding approach, to detect 10 different types of popular vulnerabilities. Our approach makes a new attempt by leveraging recent graph neural networks to embed code fragments in a compact low-dimensional code representation that preserves a program's high-level control- and data-flow

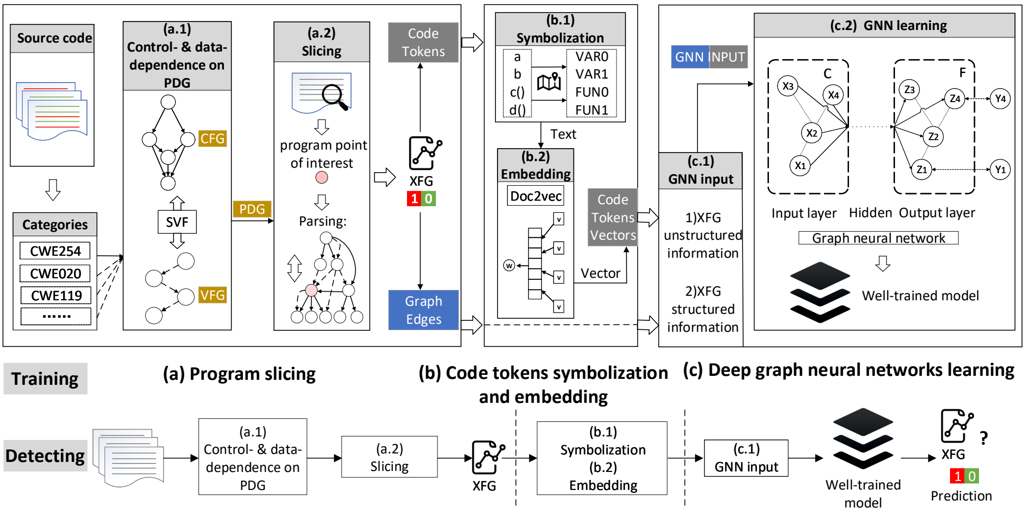


Fig. 2. Overview of DeepWukong. Models encoded with vulnerability patterns are saved after training, which then are loaded to flag whether the target gray code is vulnerable or not.

information, without the need of manually defining rules. Program slices (or XFGs) are first extracted from code fragments. A slice (or an XFG) is labelled as vulnerable if it contains a vulnerable statement (vulnerable program statements are all annotated in our ground truth samples) and safe otherwise. A neural network model is then trained using both these safe and vulnerable program slices. Both the unstructured and structured code information of a program are embedded when building our neural networks. The unstructured information is code tokens, while the structured information is manifested by the connections of nodes on XFG containing both control- and data-dependence of a program. Both information is fed into the graph neural networks to produce compact code representation in the latent feature space. By learning the vulnerable and safe program slices using recent advances in GNN, DeepWukong supports more precise bug prediction to localize vulnerabilities at the finer-grained program slice level rather than at the file or method level.

Overall framework. Fig. 2 shows the overall framework of DeepWukong consisting of two phases: a training phase and a detecting phase. For the training phase, in **(a.1)**, DeepWukong first computes the control- and data-dependence over the interprocedural CFG and VFG, with pointer aliases information being considered, and constructs the Program Dependence Graph (PDG) based on the control- and data-flow information. In **(a.2)**, DeepWukong generates each slice or XFG (a subgraph of the PDG) for the program by conducting forward and backward traversal along the PDG starting from a program point of interest (i.e., slicing criteria) until a fixed-point is reached, thereby maintaining both the data- and control-flow of a program.

In order to precisely preserve the semantic information of the source code for training a neural network, DeepWukong first conducts the variable name normalization by mapping user-defined variables and functions to their canonical symbolic names in **(b.1)**. DeepWukong then uses Doc2Vec [20] to transform the tokens for each statement (i.e., each node on XFG) of the source code into vector representations as shown in **(b.2)**.

Next, we obtain both structured (XFG edges) and unstructured information (code tokens in the form of embedding vectors for each node on XFG) as the inputs of our neural networks as depicted in **(c.1)**. Three kinds of graph neural networks are then used to evaluate the performance of our deep-learning-based approach as illustrated in **(c.2)**. Finally, a trained model is produced for bug prediction.

For the detecting phase, the control- and data-dependence of a target program is first extracted **(a.1)** to produce a set of slices (XFGs) **(a.2)**. For each XFG, after symbolization **(b.1)** and embedding **(b.2)**, both its edges and the code tokens of its nodes are used as the features **(c.1)** to feed into the previously trained model to predict whether each slice (XFG) of the target program is vulnerable or not.

We have evaluated DeepWukong using a comprehensive benchmark that contains a list of 105,428 vulnerable program with 104,104 from SARD [22] and 1,324 from two real-world project (i.e., *redis* and *lua*), which are related to the top 10 most common C/C++ vulnerabilities. We have conducted the experiments by comparing our approach with conventional static detectors, including Flawfinder [5], RATS [8], Clang Static Analyzer [2], Infer [6], and three deep-learning-based approaches [15, 16, 18]. We also compare the performance of three different graph neural networks for our code embedding, including GCN [23], GAT [24] and k-GNNs [25].

The key research contributions of this paper are:

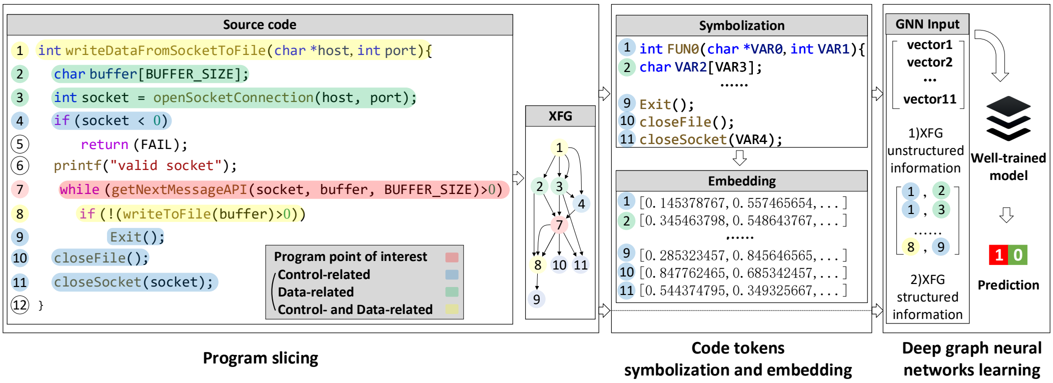
- We propose DeepWukong, a new deep-learning-based approach that embeds both textual and structured information of code into an comprehensive code representation by leveraging graph neural networks to support precise static bug detection of 10 types of popular C/C++ vulnerabilities. To enable precise code embedding, we propose a new program slicing approach to extract complicated high-level semantic features including data- and control-flows of a program.
- We have conducted our experiments by comparing DeepWukong with the traditional static bug detectors and three recent learning-based bug detection approaches. Experimental results show that our deep-learning-based approach outperforms the existing approaches in terms of informedness, markedness and F1 Score.
- We have contributed a comprehensive reference dataset to our community from real-world programs, along with all the experiment results in this paper. The data is available at: https://github.com/DeepWukong/DeepWukong.

2 MOTIVATING EXAMPLE

Fig. 3 gives an example to show our key idea by going through the three steps in Figure 2. It is a simplified pseudo code fragment from a real-world web server[4]. Note that the node number represents the statement number (line number) of the program. The code tries to establish a server socket connection. It accepts a request (Nodes 2-6) to store data on the local file system by reading the message repeatedly via a while loop (Nodes 7-9).

(a) Program slicing. DeepWukong starts with a program point of interest for its slicing. The program point is usually from an API call or a program point specified by users. We use the API call "getNextMessageAPI" at Node 7 as an example (highlighted in red), we first obtain its control- and data-flow related nodes 1, 2, 3, 4, 7, 8, 9, 10 and 11 based on the transitive closure of the union of control- and data-dependence extracted by SVF [10]. Note that Nodes 4, 9, 10 and 11 are marked blue because they are control-flow related, Nodes 2 and 3 are marked green because they are data-flow related, and Nodes 1 and 8 are marked yellow since they are both control- and data-related. The detailed program slicing algorithm is illustrated in Fig. 4 and Section 3.1.3. The corresponding statements from slicing (i.e., Nodes 1, 2, 3, 4, 7, 8, 9, 10 and 11) form the nodes of the XFG, while the control- and data-dependence between different statements contributes to the edges of XFG. For

[4] https://github.com/mongrel2/mongrel2



```
1  int writeDataFromSocketToFile(char *host, int port){
2    char buffer[BUFFER_SIZE];
3    int socket = openSocketConnection(host, port);
4    if (socket < 0)
5       return (FAIL);
6    printf("valid socket");
7     while (getNextMessageAPI(socket, buffer, BUFFER_SIZE)>0)
8       if (!(writeToFile(buffer)>0))
9          Exit();
10   closeFile();
11   closeSocket(socket);
12 }
```

```
1 int FUN0(char *VAR0, int VAR1){
2 char VAR2[VAR3];
      ......
9 Exit();
10 closeFile();
11 closeSocket(VAR4);
```

```
1 [0. 145378767, 0. 557465654, . . . ]
2 [0. 345463798, 0. 548643767, . . . ]
      ......
9 [0. 285323457, 0. 845646565, . . . ]
10 [0. 847762465, 0. 685342457, . . . ]
11 [0. 544374795, 0. 349325667, . . . ]
```

Code tokens
symbolization and embedding
GNN Input
vector1
vector2
...
vector11
1)XFG unstructured information
1 , 2
1 , 3
......
8 , 9
2)XFG structured information
Well-trained model
1 0
Prediction
Deep graph neural networks learning

Fig. 3. A real-world program extracted from *mongrel2* (a web server). It shows the key idea of DEEPWUKONG by going through the three steps in Figure 2.

instance, Node 4 and Node 7 are connected with an Edge 4->7 because Node 7 is control dependent on Node 4. Node 3 is linked with Node 7 because Node 7 is data dependent on Node 3 (the variable "socket" defined at Node 3 is used at Node 7).

(b) Code tokens symbolization and embedding. After we obtain the unstructured and structured code information, i.e., XFG, we then process each node on the XFG by symbolizing their variable and function names to make a canonical form of the textual representation. We map user-defined variables and functions to their symbolic names (e.g., variables are named as "VAR0" and "VAR1", functions are named as "FUN0" and "FUN1"). For example, we use "FUN0" to represent function name "WriteDataFromSocketToFile" and "VAR0" to represent variable name "host". DOC2VEC [20] is then used to transform the textual information, i.e., code tokens, of each node into a vector representation. For example, "**vector1**" is the vector representation of the code tokens at Node 1 (i.e., "int FUN0 (char * VAR0 , int VAR1)").

(c) Deep graph neural networks learning. The structured information of the XFG can be obtained from its edges (i.e., 1->2, 1->3, ..., 8->9). The unstructured code information is the vector representations in the previous step (i.e., the embedding vectors of nodes on the XFG (i.e., **vector1**, **vector2**, ..., **vector11**). We then feed the structured and unstructured information to the first layer of our graph neural networks. A model reflecting the code patterns is learned in the training phase, which is then used to predict whether a target XFG is vulnerable or not in the detecting phase.

3 DEEPWUKONG

This section further details each component of our approach, including program slicing (Section 3.1), code tokens symbolization and embedding (Section 3.2) and bug prediction based on deep graph neural networks (Section 3.3).

3.1 Program Slicing

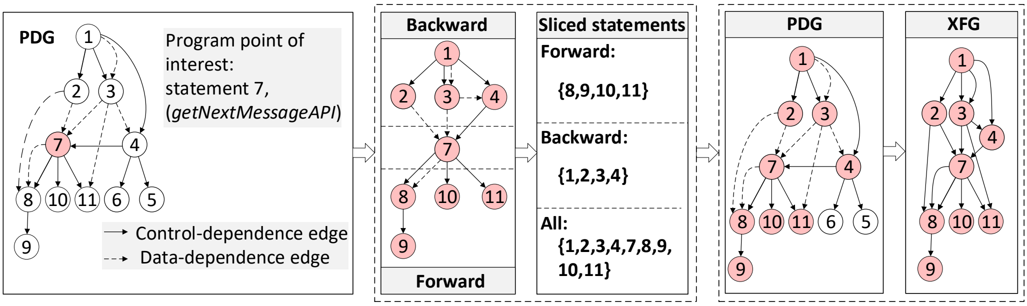
In order to conduct program slicing, we first generate the Program Dependence Graph (PDG) by considering both the control-dependence and data-dependence, which are computed over the interprocedural CFG and VFG (Section 3.1.1). Then we perform code slicing based on a slicing criteria (i.e., starting traversing from a program point of interest (Section 3.1.2)) to construct each slice or XFG (a subgraph of the PDG) via forward and backward analysis on the PDG (Section 3.1.3).

3.1.1 Control- and data-dependence on PDG . A Program Dependence Graph (PDG) is a directed graph where each node represents an instruction (program statement) and each edge represents a data- or control-dependence relation between two statements. A control-dependence edge $N_i \rightarrow N_j$ means the execution of N_j is determined by N_i, while a data-dependence edge $N_i' \rightarrow N_j'$ represents that the definition at N_i' is used at N_j'.

The control-dependence between two statements is computed over the control-flow graph (CFG) of a program where each node represents an instruction (program statement) and each edge connects two nodes, signifying the control-flow or execution order between two instructions. CFG is commonly used in static analysis and compiler optimisations, since it contains the basic execution path information such as branch conditions. Control-dependence is usually defined in terms of post-dominance. Given nodes X and Y ($X! = Y$) on a CFG, we say X post-dominates Y if all paths from Y to the end of the program traverse through X. Y is control-dependent on X if (1) there exists a directed path P from X to Y with any Z in P (excluding X and Y) is post-dominated by Y and (2) X is not post-dominated by Y. We use augmented postdominator tree (APT) [26], which is constructed in space and time proportional to the size of a program, to calculate the control-dependence of a program.

Data-dependence is obtained by the def-use relations on the interprocedural value-flow graph (VFG) [10, 27] of a program. In this graph, each node represents an instruction (i.e., program statement), and each edge represents a def-use relation of a variable between two statements. Given nodes X' and Y' ($X'! = Y'$) on a VFG. Y' is data-dependent on X' if a variable used at Y' is defined at X'. Consider two nodes X' (a definition of a variable v') and Y' (a def or use of v') in VFG, if there is a path from X' to Y' and there is no redefinition of v', Y' is data dependent on X'.

3.1.2 Program points of interest. Previous studies [18, 28, 29] show that *system API calls* are widely used by application programs and the misuse of them is one of the major causes of vulnerabilities. We use *system API calls* as the main program point of interest for our slicing. For some vulnerabilities (e.g., integer overflows, CWE190) which occur when applying arithmetic operators, such as addition and multiplication operations, we further choose *code statements containing arithmetic operators* as the program point of interest to complement the system API calls. We refer to SVF [10] to automatically identify system API calls (1,449 in total) and we use antlr [30] to identify arithmetic operators (i.e., +, -, *, /, %), bit-wise operators (&, |, ˆ , ˜ , «, »), compound assignment expressions (i.e.,+=, -=, *=, /=, %=, »=, «=, &=, ˆ =, |=), and increment/decrement expressions (++, –). For these identified statements of code, we regard them as the points of interest for our code slicing.



(a) Control- and data-dependence on PDG **(b) Slicing** **(c) XFG**

Fig. 4. An example to demonstrate our slicing by revisiting the code in Figure 3. For the PDG shown in Fig. 4(a), each node represents a code statement and each edge represents control- or data-dependence relation. Node 7 is the program point of interest. The algorithm iteratively performs backward and forward slicing until a fixed-point is reached.

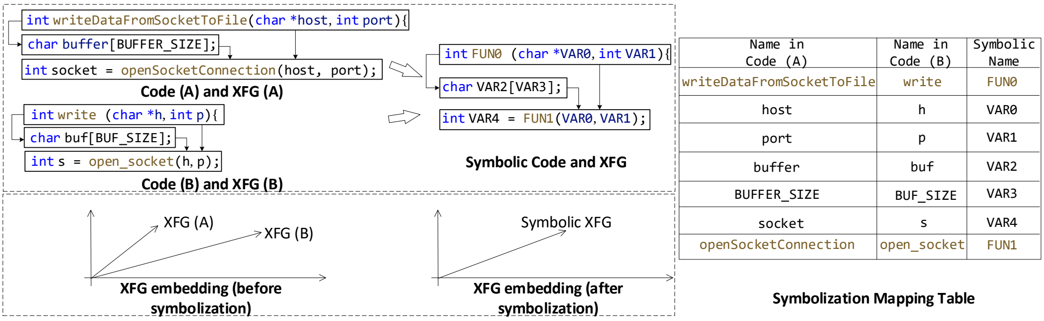
3.1.3 Slicing. After building the PDG of a program, we perform slicing [31] based on each program point of interest to produce its corresponding XFG, a subgraph of PDG. The nodes of XFG are obtained by conducting forward and backward slicing starting from a program point of interest p_i. For the forward slicing, we conduct forward traversal along the PDG starting from p_i and get forward sliced statements set S_f. For the backward slicing, we traverse the PDG starting from p_i to include all the visited statements in S_b. The statements in both S_f and S_b are the final set of statements (i.e., $S_f \cup S_b$), which preserve both control- and data-flow information of the source code and form the nodes of the resulting XFG. Note that all the reachable statements from forward or backward traversal are included in the final set of statements. We then connect those nodes based on the edges of PDG to produce the final XFG, thereby capturing both control- and data-dependence of the program.

Example 3.1. Figure 4 shows an example (originated from the code in Figure 3) to illustrate our slicing approach. We first generate PDG by constructing control-dependence (solid edge) and data-dependence (dotted edge) (as shown in Fig. 4(a)). After that, we perform forward and backward traversals along PDG starting from the API call "getNextMessageAPI" at Line 7 to obtain the forward statements set (Lines 8, 9, 10 and 11) and backward statements set (Lines 1, 2, 3 and 4) to generate the nodes of XFG (as shown in Fig. 4(b)). Finally, we connect the relevant nodes by following the edges of PDG, which represent the control- or data-dependence between two nodes. For example, Node 1 and Node 2 are connected with an edge 1->2 since Node 2 is control dependent on Node 1 (as shown in Fig. 4(c)).

3.2 Code Tokens Symbolization and Embedding

This section will introduce how we perform code token normalization and how to embed code tokens of each XFG's node into a vector representation.

3.2.1 Code tokens symbolization. For each constructed XFG, we first perform the embedding of unstructured code information (i.e., the code tokens of each node on XFG) using Doc2Vec [20]. Before the embedding, we normalize the code tokens into a canonical symbolic form in order to reduce the noise introduced by personalized naming conventions for program variables to better preserve the original code semantics.



```
int writeDataFromSocketToFile(char *host, int port){
char buffer[BUFFER_SIZE];
int socket = openSocketConnection(host, port);
```

```
int write (char *h, int p){
char buf[BUF_SIZE];
int s = open_socket(h, p);
```

```
int FUN0 (char *VAR0, int VAR1){
char VAR2[VAR3];
int VAR4 = FUN1(VAR0, VAR1);
```

Name in Code (A)	Name in Code (B)	Symbolic Name
writeDataFromSocketToFile	write	FUN0
host	h	VAR0
port	p	VAR1
buffer	buf	VAR2
BUFFER_SIZE	BUF_SIZE	VAR3
socket	s	VAR4
openSocketConnection	open_socket	FUN1

Code (A) and XFG (A)
Code (B) and XFG (B)
Symbolic Code and XFG
XFG (A)
XFG (B)
XFG embedding (before symbolization)
Symbolic XFG
XFG embedding (after symbolization)
Symbolization Mapping Table

Fig. 5. The distraction caused by personalized naming. XFG (B) and XFG (A) have the same semantic but different user-defined method and variable names.

Example 3.2. Figure 5 gives an example to show that Doc2Vec may imprecisely produce different vector representations of two cloned code fragments in the latent embedding space due to different variable naming conventions. XFG (B) is a clone code of XFG (A), which shares the same code semantic information but uses distinct user-defined variables and functions. With our symbolization, "writeDataFromSocketToFile", "host", "port", "buffer", "BUFFER_SIZE", "socket" and "openSocketConnection" in XFG (A), and "write", "h", "p", "buf", "BUF_SIZE", "s" and "open_socket" in XFG (B) are mapped to "FUN0", "VAR0", "VAR1", "VAR2", "VAR3", "VAR4" and "FUN1", respectively. In a 2-dimensional embedding space, the two embedding vectors are different (with a distance between vectors A and B) without our symbolization, while normalizing the code tokens will help produce exactly the same vector.

Following [32], we have replaced the user-defined variables and functions with their symbolic names, that is, mapping each one with a common identifier together with a separate index (i.e., FUN0, FUN1, VAR0, VAR1). Note that variables in two different code fragments (XFGs) can be mapped to the same symbolic name. For example, "buffer" in Code (A) and "buf" in Code (B) are mapped to the same symbolic name "VAR2". It is worth mentioning that our symbolization, which only changes the names of variables, does not affect the semantic of a program.

3.2.2 Code tokens embedding.

After symbolization, we will transform the code tokens of each XFG's node (i.e., program statement) into a vector using Doc2Vec [20], a widely used technique to represent documents as low-dimensional vectors. Doc2Vec is an unsupervised model which can encode the entire code statement instead of an individual code token into a fixed-length vector. The key algorithm of Doc2Vec is called Distributed Memory version of Paragraph Vector, where a unique statement-topic token is used to represent the semantic meaning of a statement.

Given a corpus comprising a sequence of tokens $w_1, w_2, ..., w_N$ including a number of code tokens and statement-topic tokens, the task is to predict a token w_i given its surrounding tokens $w_{i-k}, ..., w_{i+k}$ within a certain window of size $2k + 1$ (or so called a fixed-length context). Tokens with a similar meaning are close to each other in the latent vector space when maximising the average log probability of tokens:

$$\frac{1}{N}\sum_{i=k}^{N-k} \log p\left(w_i | w_{i-k}, ..., w_{i+k}\right) \tag{1}$$

The calculation of the probability of w_i is typically achieved using a multi-classification classifier, such as softmax:

$$p\left(w_i | w_{i-k}, ..., w_{i+k}\right) = \frac{\exp(y_{w_i})}{\sum_j \exp(y_{w_j})} \tag{2}$$

Here, y_{w_j} stands for the j^{th} element in the unnormalized log-probability distribution vector $\vec{y}$ for output tokens:

$$\vec{y} = \mathbf{U}h\left(w_{i-k}, ..., w_{i+k}; \mathbf{V}\right) + b \tag{3}$$

where each column of $\mathbf{V}$ is a unique vector representing each token (i.e., a sequence of code tokens and statement-topic tokens) and is initialized randomly. $\mathbf{U}, b$ are the softmax parameters and $h(\cdot)$ is a concatenation of token vectors extracted from $\mathbf{V}$.

The representation vectors of the code tokens of a program statement are then trained and updated automatically using stochastic gradient descent (SGD) [33]. After training, each code token vector is used to represent the token's feature while the statement-topic token vector holds the overall feature of a code statement.

Example 3.3. Figure 6 illustrates the key steps for embedding a code statement into a vector representation. The inputs are a statement-topic token vector (v_s) which represents the topic of a function declaration statement ("int FUN0 (char * VAR0 , int VAR1)") and eight token vectors ($v_1, v_2, ..., v_8$) which represent code tokens ("int", "FUN0", "(","char", "VAR0", ",", "VAR1", ")"). Those

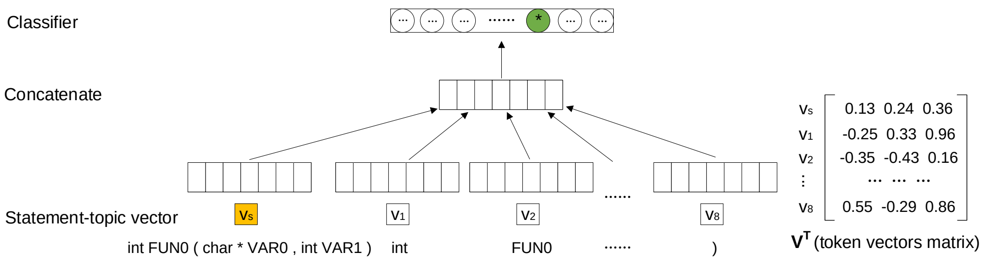


Fig. 6. An example to explain the usage of Doc2Vec for learning a statement vector. A statement-topic token represents the topic of a function declaration statement ("int FUN0 (char * VAR0 , int VAR1)") and a code token corpus consisting of eight tokens of the statement including "int", "FUN0", "(","char", "VAR0", ",", "VAR1", ")", which are then used to predict the token ("*") by concatenating token vectors and applying a softmax function. A fixed-length statement-topic vector (v_s) is obtained after training.

vectors are initialized randomly and are concatenated to produce a fixed-length vector, which is then used to predict the token "*" (shown in green) by applying a softmax function. The vectors are updated automatically in the training process. v_s can eventually represent the overall feature of the entire function declaration statement ("int FUN0 (char * VAR0 , int VAR1)"), while v_1, v_2, ..., v_8 capture the semantics of "int", "FUN0", ..., "VAR1", respectively.

Doc2Vec can provide a more precise embedding of the code statement compared to simply assembling vectors produced by individual token embedding, because important information may be lost during the token padding process, which will probably discard essential semantic-related tokens to produce a fixed-length sequence. Each node in XFG represents a code statement, which is independently considered as a short document (i.e., sequential words). After the training process of Doc2Vec, each statement-topic vector is obtained to represent each node of the XFG.

3.3 Deep graph neural networks learning

We leverage a recent proposed graph neural architecture [34] comprising a graph convolutional layer, a graph pooling layer and a graph readout layer, to perform the graph classification task for XFGs. To achieve an ideal performance and demonstrate the generality of our approach, we instantiate our graph convolutional layer by feeding the XFGs into three state-of-the-art deep graph neural networks to train different prediction models and choose the best one. Note that these graph neural networks use the same graph pooling and readout layers, while having different graph convolutional layers.

We first introduce our overall graph neural network architecture to show the full pipeline of our model. Then, we describe in detail how we adopt these networks in our convolutional layer and conduct the neural network training process. A detailed exposition of the graph convolutional layers is presented in appendix A.

3.3.1 Overall graph neural network architecture. Fig. 7 shows a general structure of our neural network, consisting of several blocks of interleaved graph convolutional/pooling layers (conv-pool block), a graph readout layer (JK-net-style [35, 36] summary) which integrates features at different graph scales, and a multilayer perceptron (MLP) [37] for the final prediction. Note that the number of conv-pool blocks is set according to the size of our dataset.

Graph convolutional layer (GCL). The general propagation rule which guides the feature transformation for the graph convolutional layer is shown here:

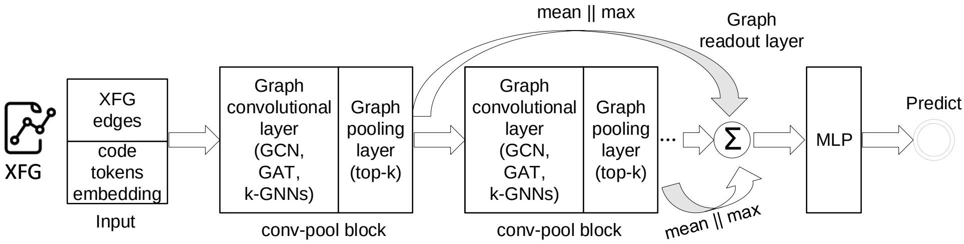

mean || max
Graph readout layer
XFG
XFG edges
code tokens embedding
Input
Graph convolutional layer (GCN, GAT, k-GNNs)
Graph pooling layer (top-k)
conv-pool block
Graph convolutional layer (GCN, GAT, k-GNNs)
Graph pooling layer (top-k)
conv-pool block
...
Σ
mean || max
MLP
Predict

Fig. 7. A general architecture of our graph neural network.

$$\vec{f}^{(l)}(v) = \sigma\left(\vec{f}^{(l-1)}(v) \cdot \mathbf{W}_1^{(l)} + \sum_{\omega \in N(v)} \vec{f}^{(l-1)}(\omega) \cdot \mathbf{W}_2^{(l)}\right), \tag{4}$$

where $\vec{f}^{(l)}(v) \in \mathbb{R}^{1\times d^{(l)}}$ means the output feature of node v in l^{th} layer, $d^{(l)}$ represents the dimension and $N(v)$ means the adjacent nodes of node v. $\mathbf{W}^{(l)} \in \mathbb{R}^{d^{(l-1)}\times d^{(l)}}$ means the weight matrices of l^{th} layer. Note that the weight matrices guiding the feature transformation of a node itself ($\mathbf{W}_1$) and its neighbours ($\mathbf{W}_2$) can be different according to the principle proposed by some graph neural networks (e.g. k-GNNs [25]). $\sigma(\cdot)$ is the activation function.

Graph pooling layer (GPL). The objective of the graph pooling layer is to reduce the size of the original graph by applying a pooling ratio, $k \in (0, 1]$, meaning that a graph with N nodes will have $\lceil kN \rceil$ nodes after such a pooling layer, thereby reducing the amount of subsequent computation in the network. It simply drops $N - \lceil kN \rceil$ nodes after each pooling operation based on a projection score against a learnable vector, $\vec{p}$. The projection scores are also used as gating values to enable gradients to flow into $\vec{p}$; therefore, for the remaining nodes, the extent of feature retention will be proportional to their scores.

$\mathbf{F} = \left\{\vec{f}(0), \vec{f}(1), ..., \vec{f}(N)\right\} \in \mathbb{R}^{N\times d}$ means the matrix of node features. We use $\mathbf{A}$ to denote the adjacency matrix representing a graph. The transformation principle of the graph pooling layer which computes a pooled graph, $(\mathbf{F}', \mathbf{A}')$, from an input graph, $(\mathbf{F}, \mathbf{A})$, can be formally expressed as:

$$\vec{y} = \frac{\mathbf{F}\vec{p}}{\|\vec{p}\|} \qquad \vec{i} = top{-}k(\vec{y}, k) \qquad \mathbf{F}' = (\mathbf{F} \odot \tanh \vec{y})_{\vec{i}} \qquad \mathbf{A}' = \mathbf{A}_{\vec{i},\vec{i}} \tag{5}$$

$\|\cdot\|$ stands for the $L2$ norm, $top{-}k$ selects the $top{-}k$ indices from a given input vector, $\odot$ means (broadcasted) elementwise multiplication, and $\cdot_{\vec{i}}$ is an indexing operation which takes nodes at indices specified by $\vec{i}$.

Graph readout layer (GRL). In order to produce a fixed-length representation for the entire graph so as to conduct the graph classification task, a graph readout layer is introduced to flatten feature of all the nodes. Following traditional convolutional neural networks (CNNs), we perform global average pooling and global max pooling to strengthen the performance of our representation. Moreover, inspired by the JK-net architecture [35, 36], this summarization is performed after each block of interleaved graph convolutional/pooling layers (conv-pool block) and finally all of the summarizations are aggregated together. The output graph vector $\vec{s}^{(l)} \in \mathbb{R}^{1\times d^{(l)}}$ of the l^{th} conv-pool block, $(\mathbf{F}^{(l)}, \mathbf{A}^{(l)})$ can be formally expressed as:

$$\vec{s}^{(l)} = \frac{1}{N^{(l)}} \sum_{i=1}^{N^{(l)}} \vec{f}^{(l)}(i) || max_{j=1}^{N^{(l)}} \vec{f}^{(l)}(j) \tag{6}$$

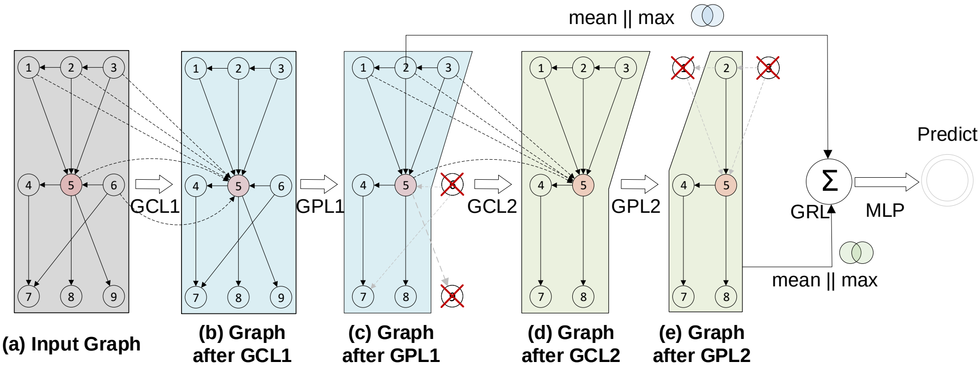


Fig. 8. An example to illustrate the full pipeline of our general graph neural network architecture.

Here, $\vec{f}^{(l)}(i)$ stands for the feature vector of node i, $N^{(l)}$ means the number of nodes on the graph, and || denotes the concatenation operation. Finally, the summary vector of the graph is computed as the sum of all those summaries (i.e. $\vec{s} = \sum_{l=1}^{L} \vec{s}^{(l)}$). Note that it is significant to aggregate across conv-pool blocks to preserve information at different scales of processing.

The vector representing the entire graph is then fed into a multilayer perceptron (MLP) [37] for the final prediction. The predicted distribution of the model $q(lb_i)$ is computed using a softmax function, i.e., the dot product between the graph vector $\vec{s}$ and the vector representation lb_i of each label $lb_i \in lb$.

$$q(lb_i) = \frac{\exp(\vec{s} \cdot lb_i)}{\sum_{lb_j \in Y} \exp(\vec{s} \cdot lb_j)}, \tag{7}$$

Example 3.4. Figure 8 exemplifies the full pipeline of our graph neural network, consisting of two interleaved graph convolutional/pooling layers (GCL1/GPL1, GCL2/GPL2) for graph feature abstraction, a graph readout layer (GRL) for summarization and flattening, and a multilayer perceptron for prediction.

Starting from the input graph in **(a)**, which consists of nine nodes (Nodes 1-9) with their links (1->5, 2->1, 2->5, 3->2, 3->5, 4->7, 5->4, 5->8, 5->9, 6->5, 6->7), the graph is then fed into the convolutional layer, which transforms the feature of each node by considering its predecessors and itself. For example, the vector representation of Node 5 (highlighted in red) after GCL1 in **(b)** is calculated over its predecessors (Node 1, 2, 3, 6) and itself based on Equation 4. Afterward, the graph passes through GPL1, reducing its size from 9 to 7, with Node 6 and Node 9 together with their links with other nodes (Edges 5->9, 6->5, 6->7) removed according to Equation 5, to produce the graph in **(c)**. Note that the graph pooling ratio k here is set to 0.7. Similarly, the size of the graph is further abstracted and reduced through GCL2 and GPL2, respectively. A JK-net-style summary [35, 36] is subsequently applied to the graph after GPL1 in **(c)** and that after GPL2 in **(e)** by summing the concatenation of the mean and max pooling of all the nodes of each graph, respectively (Equation 6), to generate the vector representation of the entire graph, which is finally used to predict the category of the input graph.

3.3.2 Graph neural networks training. Fig. 9 shows a general structure for training our neural networks. A batch of XFG with its edges and code tokens (node) embedding is input into the graph neural network, consisting of several blocks of interleaved graph convolutional/pooling layers (conv-pool block), a graph readout layer and a MLP.

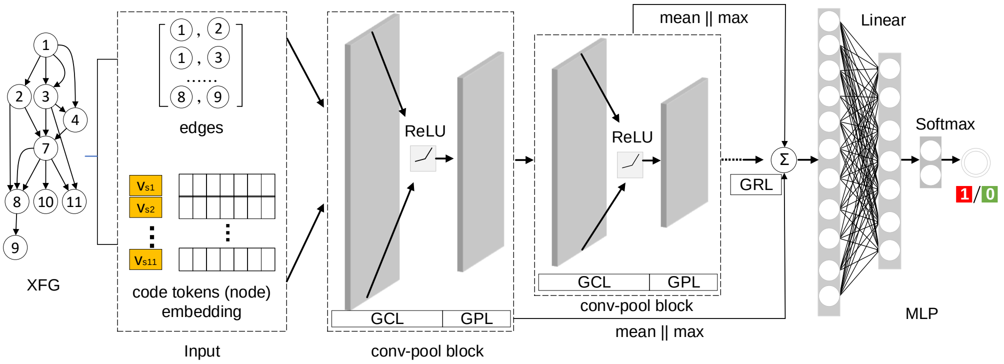


Fig. 9. The structure of the graph neural network in DeepWukong.

GNN input. The inputs of the neural network are a batch of XFG obtained from the previous steps, which can be divided into two parts: (1) the edges (i.e., structured information) and (2) the code tokens (node) embedding (i.e., unstructured information).

GNN learning. The inputs are fed into the above-introduced graph neural network. In the graph convolutional layer, features are extracted from the node embedding and the edges of the input XFG. After each graph convolutional layer, we use $top-k$ pooling [34] to reduce the size of the XFG. In our detection system, there can be multiple conv-pool block. A graph readout layer realised by the concatenate of mean pooling and max pooling is applied after each conv-pool block to flatten multiple nodes vectors of the graph into one graph vector. We then use a hidden fully connected layer after that, which allows high-order relationships between the features to be detected. Finally, a softmax layer is used to output the probabilities of class labels. It is worth noting that, most of the parameters in the neural network are updated automatically by back propagation during training. At last, we get a well-trained graph neural network model encoded with vulnerability patterns after GNN learning. We then use the model for further vulnerability detection.

4 EXPERIMENTAL EVALUATION

We seek to evaluate the effectiveness of DeepWukong on detecting the top 10 most common C/C++ vulnerabilities, comparing with four traditional static vulnerability detectors and three state-of-the-art deep-learning-based approaches. To this end, we first introduce our dataset extracted from real-world vulnerabilities, and then depict how they are labelled (Section 4.1). Next, we detail our experimental setup and model training (Section 4.2). Finally, we present our experimental results and observations (Section 5).

4.1 Dataset

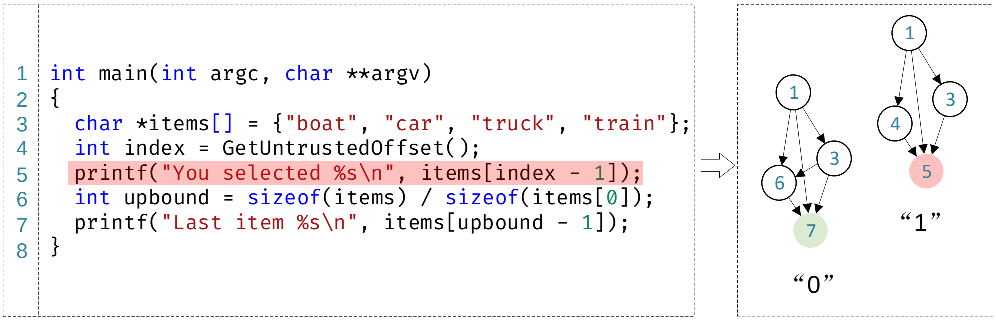
4.1.1 Target vulnerabilities. We have established our dataset based on two sources (1) SARD [22], a widely-used vulnerability database, and (2) two real-world open-source projects (*lua* and *redis*). The statistics of the vulnerable programs in our dataset are shown in Table 2.

SARD. We have harvested a comprehensive vulnerability benchmark dataset from Software Assurance Reference Dataset (SARD) [22], which hosts a large number of known real-world security flaws. It is widely used to evaluate the performance of vulnerability detection approaches in our community [18, 38, 39]. In the SARD dataset, each program (i.e., test case) corresponds to one or more CWE IDs, as multiple types of vulnerabilities could be identified in a program. We seek to study the top 10 most common C/C++ vulnerabilities as aforementioned in Section 1. Thus, we have implemented a crawler to harvest all the available programs related to the following vulnerabilities:

Table 2. The statistics of our datasets. #LOI denotes the number of lines of LLVM instructions. #Pointer, #Object, #Call represent the numbers of pointers, objects and method calls, |V| (ICFG), |E| (ICFG), |V| (VFG) and |E| (VFG) are the numbers of ICFG nodes, ICFG edges, VFG nodes and VFG edges, respectively.

Vulnerability Category	#LOI	#Pointer	#Object	#Call	|V| (ICFG)	|E| (ICFG)	|V| (VFG)	|E| (VFG)
CWE119	7,811,996	2,992,105	597,759	84,857	2,657,488	2,962,382	2,333,565	2,526,699
CWE20	7,728,477	3,205,748	601,656	65,394	2,828,956	3,133,667	2,420,491	2,471,687
CWE125	1,528,454	595,807	118,084	16,704	532,091	595,730	466,155	479,682
CWE190	1,611,932	653,629	108,248	18,250	612,164	690,579	496,601	512,800
CWE22	7,728,477	3,205,748	601,656	65,394	2,828,956	3,133,667	2,420,491	2,471,687
CWE399	5,552,545	1,895,244	359,892	63,500	1,780,518	2,013,559	1,419,467	1,566,615
CWE787	5,525,537	2,129,251	425,057	60,567	1,884,266	2,098,410	1,663,824	1,813,045
CWE254	14,576,245	5,079,282	965,625	151,987	4,535,930	5,096,619	3,898,307	4,102,872
CWE400	1,210,852	458,445	71,088	19,654	428,373	495,526	331,910	337,427
CWE78	703,660	290,148	50,478	5,383	242,148	263,665	217,122	200,428
lua	68,222	52,585	1,869	4,764	59,055	72,829	100,701	106,830
redis	735,275	401,520	14,461	119,216	45,737	56,505	654,495	778,854
Total	50,067,406	18,337,496	3,424,166	625,006	16,118,260	18,066,536	14,443,657	15,342,334

(1) **CWE119: Improper Restriction of Operations within the Bounds of a Memory Buffer.** The program reads from or writes to a memory location that is outside of the intended boundary of the memory buffer.
(2) **CWE20: Improper Input Validation.** The program does not validate or incorrectly validates input that can affect the control-flow or data-flow of a program.
(3) **CWE125: Out-of-bounds Read.** The program reads data past the end, or before the beginning, of the intended buffer.
(4) **CWE190: Integer Overflow or Wraparound.** The program performs a calculation that can produce an integer overflow or wraparound, when the logic assumes that the resulting value will always be larger than the original value.
(5) **CWE22: Improper Limitation of a Pathname to a Restricted Directory.** The program uses external input to construct a pathname that is intended to identify a file or directory that is located underneath a restricted parent directory, but the software does not properly neutralize special elements within the pathname that can cause the pathname to resolve to a location that is outside of the restricted directory.
(6) **CWE399: Resource Management Errors.** It is related to improper management of system resources.
(7) **CWE787: Out-of-bounds Write.** The program writes data past the end, or before the beginning, of the intended buffer.
(8) **CWE254: Security Features.** It is related to security related operations, e.g., authentication, access control, confidentiality, cryptography, and privilege management, etc.
(9) **CWE400: Uncontrolled Resource Consumption.** The program does not properly control the allocation and maintenance of a limited resource thereby enabling an actor to influence the amount of resources consumed, eventually leading to the exhaustion of available resources.
(10) **CWE78: Improper Neutralization of Special Elements.** The vulnerable program constructs all or part of an OS command using externally-influenced input from an upstream component, but it does not neutralize or incorrectly neutralizes special elements that could modify the intended OS command when it is sent to a downstream component.



```
1 int main(int argc, char **argv)
2 {
3   char *items[] = {"boat", "car", "truck", "train"};
4   int index = GetUntrustedOffset();
5   printf("You selected %s\n", items[index - 1]);
6   int upbound = sizeof(items) / sizeof(items[0]);
7   printf("Last item %s\n", items[upbound - 1]);
8 }
```

Source code with label

XFG with label

Fig. 10. An example to show the labelling process of XFG. The source code shown on the left side is an example of CWE119, which shows the location (i.e., statement 5) of the weaknesses included. The XFGs generated from the source code are presented on the right side including one "0" (i.e., safe) XFG (without statement 5) and one "1" (i.e., vulnerable) XFG (with statement 5).

Real-world open-source projects. To evaluate our approach on complex real-world open-source projects, we have further collected recent security-related commits of two open-source projects: *redis-5.0.8*[5] (a well-known database system server) and *lua-5.3.4*[6] (a widely-used script language).

4.1.2 Labelling the benchmark. As the vulnerabilities collected from SARD and real-world projects have different formats, we first present the labelling methods of them, respectively, and then show how we label XFG and method for further training and detecting.

Labelling SARD. The programs we collect from SARD have already been labelled at the statement level as "good" (i.e., containing no security defect), "bad" (i.e., containing one or more specific security defects), or "mixed" (i.e., containing security defects and their fixed safe patches) using corresponding CWE IDs.

Labelling real-world open-source projects. To label the benchmark in open-source projects, we adopt the methodology proposed by Zhou et al. [40]. To ensure the quality of our labelling process for the open-source projects, **(1)** we first extract bug-fixing commits by excluding the commits whose messages do not contain any bug-related keywords such as "bug", "crash" and "memory error". **(2)** Then we manually check the modifications of each bug-fixing commit. Based on how the bug is fixed, we label safe statements as "0" from the fixed commit and vulnerable statements as "1" from its corresponding vulnerable commit. It took 720 hours for three experienced software engineers to carefully examine all the bug-related commits of our collected projects.

Labelling XFG and method. Our vulnerability detection is performed at the XFG level rather than method or file level, i.e., reporting whether an XFG (a program slice) is vulnerable or not. As described in Section 3.1, we first conduct program slicing for each program to generate a number of slices (i.e., XFG) provided that one program may have multiple points of interest, which determine the number of XFGs. Then, we further utilize the above-mentioned statement-level labelling to label the vulnerable/safe XFG. Based on the labelling method of SARD, if a sample is extracted from a "good" program we label it as "0" (i.e., safe), and if it is from a "bad" or "mixed" program we label it as "1" (i.e., vulnerable) as long as it contains one vulnerable statement labelled by SARD, and "0" (i.e., safe) otherwise. Similarly, the XFG extracted from real-world open-source projects is

[5] https://redis.io/

[6] https://www.lua.org/

Table 3. Distribution of labelled samples from SARD.

Vulnerability Category	granularity	# vulnerable samples	# safe samples	# total
CWE119	testcase	18,928	360	19,288
	XFG	**21,508**	**50,052**	**71,560**
	method	18,311	101,803	120,114
	slice	8,853	11,700	20,553
CWE20	testcase	19,617	404	20,021
	XFG	**35,746**	**106,624**	**142,370**
	method	26,397	16,8245	194,642
	slice	8,112	24,806	32,918
CWE125	testcase	3,645	0	3,645
	XFG	**3,677**	**8,187**	**11,864**
	method	3,536	20,818	24,354
	slice	1,310	2,061	3,371
CWE190	testcase	1,052	149	1,201
	XFG	**2,555**	**6,076**	**8,631**
	method	3,759	24,994	28,753
	slice	188	1,280	1,468
CWE22	testcase	2,764	5	2,769
	XFG	**5,165**	**6,857**	**12,022**
	method	4,692	22,141	26,833
	slice	1,426	1,786	3,212
CWE399	testcase	12,367	24	12,391
	XFG	**14,060**	**38,001**	**52,061**
	method	12,669	86,043	98,712
	slice	2,521	5,937	8,458
CWE787	testcase	12,946	48	12,994
	XFG	**14,419**	**31,535**	**45,954**
	method	12,477	62,670	75,147
	slice	7,417	8,451	15,868
CWE254	testcase	27,776	409	28,185
	XFG	**49,569**	**134,232**	**183,801**
	method	38,603	229,942	268,545
	slice	12,791	31,179	43,970
CWE400	testcase	1,872	0	1,872
	XFG	**6,677**	**22,671**	**29,348**
	method	3,875	31,629	35,504
	slice	485	2,337	2,822
CWE78	testcase	1,729	9	1,738
	XFG	**2,819**	**5,027**	**7,846**
	method	4,719	26,429	31,148
	slice	305	5,285	5,590
TOTAL	testcase	102,696	1,408	104,104
	XFG	**156,195**	**409,262**	**565,457**
	method	129,038	774,714	903,752
	slice	43,408	94,822	138,230

considered to be vulnerable as long as it covers at least one statement of vulnerable code. Note that, some existing learning-based approaches [15, 16] perform vulnerability detection at the method level, we further process the sample code and flag the vulnerabilities also at the method level (i.e., the method is marked as vulnerable if there is a vulnerable statement in the method) in order to compare with these existing approaches (cf. Section 5.3).

Example 4.1. Take the code fragment[7] in Figure 10 as an example, it is an example to demonstrate the principle of CWE119. The figure depicts the content of source code and its label (i.e.,"bad" at Statement 5) with the corresponding CWE ID (i.e., CWE119). Two XFGs extracted from the code fragment are shown on the right side. The program points of interest of them are Statements 5 and 7, respectively. Accordingly, we label the XFG on the right as "1" (i.e., vulnerable) as it contains a vulnerable statement at Statement 5 and the XFG on the left as "0" (i.e.,safe), because all the statements (Statements 1, 3, 6, 7) are not vulnerable.

[7] https://cwe.mitre.org/data/definitions/119.html

Table 4. Distribution of labelled samples from open-source project.

project	granularity	# vulnerable samples	# safe samples	# total
redis	**XFG**	**737**	**1,270**	**2,007**
	method	1,877	2,606	4,483
	slice	512	340	852
lua	**XFG**	**586**	**1,578**	**2,164**
	method	898	1,072	1,970
	slice	601	987	1,588
TOTAL	**XFG**	**1,323**	**2,848**	**4,171**
	method	2,775	3,678	6,453
	slice	1,113	1,327	2,440

4.1.3 Handling Duplicate/Conflict Samples. It is possible that there exists duplicate XFGs because different programs may have the same code logic leading to the same sliced statements and structure. The duplicate samples could lead to the *overfitting problem* and artificially inflate performance metrics because training data can leak into validation. We remove duplicate XFGs in our dataset by comparing their nodes, which are sorted by line number, and edges in space and time proportional to the size of an XFG. It is also possible that a few XFGs are labelled with both “1” and “0” because of the mislabelling from SARD, as mentioned by previous work [18]. In this situation, we simply remove all the conflict samples.

4.1.4 Distribution of the benchmark. From the SARD dataset, we collect 102,696 and 1,408 vulnerable and safe programs, respectively (cf. Table 3). After program slicing, **we have labelled 156,195 vulnerable XFGs and 409,262 safe XFGs**. Each XFG contains 11 nodes (i.e., statements) on average, with respect to an average of 50 nodes and 217 statements for each ICFG and file of a program, respectively, which further indicates that our XFG is fine-grained. For the two real-world open-source projects *redis* and *lua*, we have collected 885 and 311 bug-fixing commits, respectively, from which we have produced and labelled 1,323 vulnerable XFGs and 2,848 safe XFGs (cf. Table 4). Each XFG contains 210 nodes (i.e., statements) on average with an average of 423 nodes and 1004 statements for each ICFG and file of a program, respectively. Comparing with existing studies, the benchmarks we collected are large and comprehensive to perform evaluation.

4.2 Experimental Setup

Experimental environment and neural network configuration. The experiments are performed on a machine with NVIDIA GeForce GTX 1080 GPU and Intel Xeon E5-1620 CPU operating at 3.50GHz. The graph neural networks are implemented using PyTorch Geometric [41]. We perform experiments separately on each type of vulnerabilities, i.e., training a model for each of the 10 types of the vulnerabilities. We randomly choose 80% of the programs for training and the remaining 20% for detecting. The neural networks are trained in a batch-wise fashion and the batch size is set to 64. We adopt a 10-fold cross validation to train the neural network. The dimension of the vector representation of each node is set to 64. The dropout is set to 0.5 and the number of epochs is set to 50. The minibatch stochastic gradient descent together with ADAM [42] is used for training with the learning rate of 0.001. We use grid search to perform hyper parameter (e.g. batch size, dropout, learning rate) tuning in order to determine the optimal values for a given model. To prevent under- and over-fitting problem, we use different number of convolutional layers on the dataset according to the size of vulnerability category (5 for CWE20 and CWE254, 4 for CWE119, 3 for CWE399, CWE787 and CWE400 and 2 for CWE125, CWE190, CWE22, CWE78 and open-source projects). The parameter k in k-GNNs is set to 3 and the graph pooling ratio is set to 0.8. The other parameters of our neural network are tuned in a standard method. All network weights and biases are randomly initialized using the default Torch initialization.

Evaluation metrics. we apply six widely used metrics [43], including accuracy (ACC), false positive rate (FPR), false negative rate (FNR), true positive rate (TPR), Precision (P), F1 Score (F1) and AUC, to evaluate the performance of DeepWukong and the other competitors. We also adopt informedness (IFN) and markedness (MKN) [44], which are unbiased variants of recall and precision, respectively, to conduct our evaluation.

4.3 Research Questions

Our evaluation aims to answer the following four research questions:

RQ1 **Can DeepWukong accurately detect vulnerabilities?** In particular, we would like to further investigate (RQ1.1) can DeepWukong achieve consistently good performance across various types of vulnerabilities? and, (RQ1.2) are there any performance differences when using three types of deep graph neural networks, and to what extent?

RQ2 **Can DeepWukong outperform traditional bug detection tools?**

RQ3 **Can DeepWukong outperform existing deep-learning-based vulnerability detection approaches?**

RQ4 **Can DeepWukong be applied to real-world open-source applications effectively?** SARD is a security benchmark whose overall complexity may not be comparable to real-world applications; therefore, we further compare DeepWukong with the other state-of-the-art approaches mentioned based on the dataset built on real-world open-source applications.

5 RESULTS AND ANALYSIS

5.1 RQ1: The performance of DeepWukong

Overall result. Table 5 shows the overall results on the benchmark programs in terms of the aforementioned evaluation metrics. In general, DeepWukong achieves very promising results. On average, the accuracy is 97.4% (ranging from 96.6% to 98.9%) and the F1 Score is 95.6% (ranging from 94.0% to 98.8%), across all the top 10 vulnerabilities. Though it is generally time-consuming for training, bug prediction and detection is quite fast. For example, the training time of the largest dataset (CWE254) is 83,248.5s while it only takes 321.1s for detection.

5.1.1 RQ1.1: Detection Result across Vulnerabilities. As shown in Table 5, the performance differences when detecting different types of vulnerabilities are marginal. Our tool performs the best on CWE22 vulnerability, with the F1 Score of 98.8%. DeepWukong achieves the worst result on CWE254 and CWE190, with a good enough F1 Score of 94.0% and 94.1%, respectively. We further manually examine a number of exceptional cases to explore the reasons leading to the difference (false positives and false negatives).

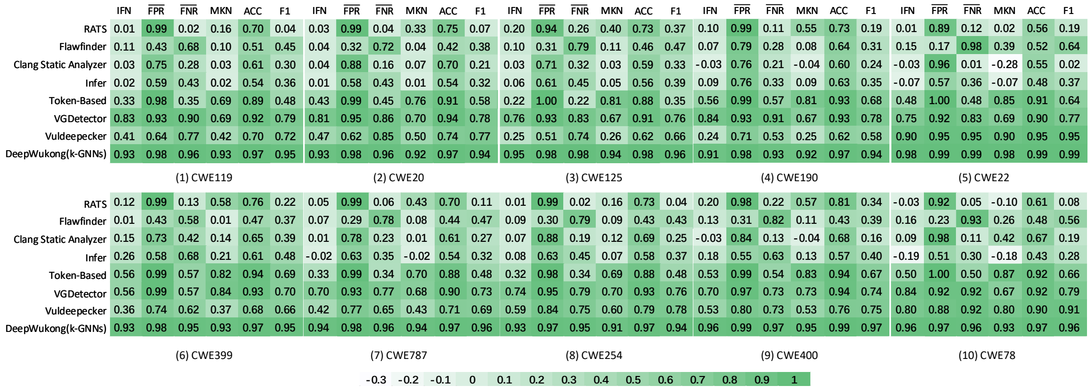
We have identified the following two main reasons leading to the inaccurate prediction. First, the selected program points of interest are not perfect. We take advantage of program slicing to generate the XFG, seeking to cover as much vulnerable code as possible, and preserve the semantics of the vulnerabilities. However, it is quite possible that we may lose some vulnerabilities that are not associated with the program points of interest we select (cf. Section 3.1.2). Nevertheless, we have extended the existing work on selecting program points of interest and have a broader coverage than state-of-the-art work [18]. Second, some kinds of vulnerabilities may exhibit quite different behaviors (i.e., manifesting as different patterns on control- and data-flow), which makes it hard for us to learn their patterns, especially when considering the limited number of training samples. Nevertheless, the performance of DeepWukong is already good enough across all kinds of vulnerabilities that we considered in our evaluation.

Table 5. A comparison of GNNs (best results shown in bold).

Category	GNN	IFN	FPR	FNR	MKN	ACC	F1	AUC	training time(s)	testing time(s)
CWE119	GCN	0.921	0.027	0.052	0.916	0.966	0.943	0.996	**26,125.6**	**99.1**
	GAT	0.920	0.031	0.049	0.909	0.964	0.941	0.995	26,129.4	100.5
	k-GNNs	**0.932**	**0.023**	**0.045**	**0.926**	**0.970**	**0.950**	**0.996**	26,140.7	103.9
CWE20	GCN	0.921	0.026	0.053	0.905	0.967	0.935	0.996	64,522.9	246.2
	GAT	0.924	0.028	0.048	0.902	0.967	0.935	0.995	**64,481.4**	**238.2**
	k-GNNs	**0.933**	**0.023**	**0.044**	**0.918**	**0.972**	**0.944**	**0.997**	64,566.4	257.7
CWE125	GCN	0.926	0.029	0.046	0.917	0.966	0.946	0.995	2,443.9	10.1
	GAT	0.894	0.039	0.067	0.885	0.953	0.924	0.989	**2,443.1**	**10.1**
	k-GNNs	**0.954**	**0.022**	**0.025**	**0.942**	**0.977**	**0.964**	**0.997**	2,446.2	10.6
CWE190	GCN	0.869	0.036	0.095	0.874	0.946	0.909	0.988	1,955.2	7.8
	GAT	0.900	0.033	**0.066**	0.895	0.957	0.928	0.991	**1,953.6**	7.6
	k-GNNs	**0.913**	**0.021**	0.067	**0.923**	**0.966**	**0.941**	**0.995**	1,954.1	**7.6**
CWE22	GCN	0.958	0.021	0.021	0.958	0.979	0.976	0.998	2,198.9	7.9
	GAT	0.969	0.016	0.015	0.967	0.984	0.982	0.999	**2,186.3**	**6.5**
	k-GNNs	**0.979**	**0.011**	**0.011**	**0.978**	**0.989**	**0.988**	**0.999**	2,203.5	8.6
CWE399	GCN	0.907	0.026	0.067	0.905	0.963	0.931	0.994	15,614.1	59.3
	GAT	0.848	0.036	0.116	0.859	0.943	0.892	0.987	**15,602.6**	**57.9**
	k-GNNs	**0.929**	**0.020**	**0.052**	**0.928**	**0.972**	**0.947**	**0.996**	15,611.8	58.9
CWE787	GCN	0.913	0.025	0.063	0.917	0.963	0.941	0.995	13,245.1	52.9
	GAT	0.893	0.028	0.079	0.903	0.956	0.930	0.992	**13,242.8**	**52.7**
	k-GNNs	**0.938**	**0.018**	**0.044**	**0.941**	**0.974**	**0.958**	**0.997**	13,262.7	56.1
CWE254	GCN	0.905	0.027	0.069	0.903	0.962	0.930	0.995	83,234.9	319.7
	GAT	0.903	**0.025**	0.056	**0.922**	0.964	**0.947**	0.994	**83,150.6**	**317.9**
	k-GNNs	**0.925**	0.027	**0.047**	0.910	**0.967**	0.940	**0.997**	83,248.5	321.1
CWE400	GCN	0.936	0.015	0.049	0.933	0.977	0.949	0.967	11,627.5	44.4
	GAT	0.933	0.016	0.052	0.934	0.976	0.949	0.996	**11,620.1**	**41.7**
	k-GNNs	**0.963**	**0.011**	**0.025**	**0.951**	**0.985**	**0.967**	**0.998**	11,631.7	47.5
CWE78	GCN	0.918	0.043	**0.039**	0.905	0.959	0.943	0.995	1,458.9	**5.9**
	GAT	0.907	0.043	0.050	0.899	0.954	0.937	0.994	**1,458.6**	6.1
	k-GNNs	**0.932**	**0.026**	0.042	**0.930**	**0.968**	**0.956**	**0.996**	1,461.7	6.4
AVERAGE	GCN	0.917	0.027	0.055	0.913	0.965	0.940	0.995	22242.7	85.3
	GAT	0.909	0.029	0.061	0.905	0.962	0.935	0.993	**22226.9**	**83.9**
	k-GNNs	**0.940**	**0.020**	**0.040**	**0.935**	**0.974**	**0.956**	**0.997**	22252.7	87.8

ANSWER: DeepWukong is effective in automatically learning high-level semantic features from graph embedding, which shows very good performance in detecting all the top 10 vulnerabilities. A few exceptional cases are introduced by the program points of interest selection process and the limited number of training samples.

5.1.2 RQ1.2: A Comparison of Graph Neural Networks. Table 5 compares the results of different GNNs on the XFG classification. k-GNNs performs the best among the three networks, although the other two networks have close results. There are mainly two reasons behind. First, we perform vulnerability detection at the XFG level. An XFG (slice) contains several program statements, which is more *fine-grained* than the approach working at the method level. Thus, XFG is able to precisely preserve code features (control- and data-flows) only for bug-relevant code statements. Theoretically, k-GNNs [25] works better with fine-grained structures of a given graph than the other two networks, which was shown by previous studies [25, 45]. In addition, GCN and GAT consider each node separately at the same level, while k-GNNs acts better by capturing the subgraph structured information (e.g., "if-else" code block) that is not visible at the node level. As implied by the similar time complexity of these models, the training and testing time costs are close among these neural networks considering each type of vulnerabilities.



	IFN	$\overline{\text{FPR}}$	$\overline{\text{FNR}}$	MKN	ACC	F1
RATS	0.01	0.99	0.02	0.16	0.70	0.04
Flawfinder	0.11	0.43	0.68	0.10	0.51	0.45
Clang Static Analyzer	0.03	0.75	0.28	0.03	0.61	0.30
Infer	0.02	0.59	0.43	0.02	0.54	0.36
Token-Based	0.33	0.98	0.35	0.69	0.89	0.48
VGDetector	0.83	0.93	0.90	0.69	0.92	0.79
Vuldeepecker	0.41	0.64	0.77	0.42	0.70	0.72
DeepWukong(k-GNNs)	0.93	0.98	0.96	0.93	0.97	0.95

(1) CWE119

	IFN	$\overline{\text{FPR}}$	$\overline{\text{FNR}}$	MKN	ACC	F1
RATS	0.03	0.99	0.04	0.33	0.75	0.07
Flawfinder	0.04	0.32	0.72	0.04	0.42	0.38
Clang Static Analyzer	0.04	0.88	0.16	0.07	0.70	0.21
Infer	0.01	0.58	0.43	0.01	0.54	0.32
Token-Based	0.43	0.99	0.45	0.76	0.91	0.58
VGDetector	0.81	0.95	0.86	0.70	0.94	0.78
Vuldeepecker	0.47	0.62	0.85	0.50	0.74	0.77
DeepWukong(k-GNNs)	0.93	0.98	0.96	0.92	0.97	0.94

(2) CWE20

	IFN	$\overline{\text{FPR}}$	$\overline{\text{FNR}}$	MKN	ACC	F1
RATS	0.20	0.94	0.26	0.40	0.73	0.37
Flawfinder	0.10	0.31	0.79	0.11	0.46	0.47
Clang Static Analyzer	0.03	0.71	0.32	0.03	0.59	0.33
Infer	0.06	0.61	0.45	0.05	0.56	0.39
Token-Based	0.22	1.00	0.22	0.81	0.88	0.35
VGDetector	0.76	0.93	0.83	0.67	0.91	0.76
Vuldeepecker	0.25	0.51	0.74	0.26	0.62	0.66
DeepWukong(k-GNNs)	0.95	0.98	0.98	0.94	0.98	0.96

(3) CWE125

	IFN	$\overline{\text{FPR}}$	$\overline{\text{FNR}}$	MKN	ACC	F1
RATS	0.10	0.99	0.11	0.55	0.73	0.19
Flawfinder	0.07	0.79	0.28	0.08	0.64	0.31
Clang Static Analyzer	-0.03	0.76	0.21	-0.04	0.60	0.24
Infer	0.09	0.76	0.33	0.09	0.63	0.35
Token-Based	0.56	0.99	0.57	0.81	0.93	0.68
VGDetector	0.84	0.93	0.91	0.67	0.93	0.78
Vuldeepecker	0.24	0.71	0.53	0.25	0.62	0.58
DeepWukong(k-GNNs)	0.91	0.98	0.93	0.92	0.97	0.94

(4) CWE190

	IFN	$\overline{\text{FPR}}$	$\overline{\text{FNR}}$	MKN	ACC	F1
RATS	0.01	0.89	0.12	0.02	0.56	0.19
Flawfinder	0.15	0.17	0.98	0.39	0.52	0.64
Clang Static Analyzer	-0.03	0.96	0.01	-0.28	0.55	0.02
Infer	-0.07	0.57	0.36	-0.07	0.48	0.37
Token-Based	0.48	1.00	0.48	0.85	0.91	0.64
VGDetector	0.75	0.92	0.83	0.69	0.90	0.77
Vuldeepecker	0.90	0.95	0.95	0.90	0.95	0.95
DeepWukong(k-GNNs)	0.98	0.99	0.99	0.98	0.99	0.99

(5) CWE22

	IFN	$\overline{\text{FPR}}$	$\overline{\text{FNR}}$	MKN	ACC	F1
RATS	0.12	0.99	0.13	0.58	0.76	0.22
Flawfinder	0.01	0.43	0.58	0.01	0.47	0.37
Clang Static Analyzer	0.15	0.73	0.42	0.14	0.65	0.39
Infer	0.26	0.58	0.68	0.21	0.61	0.48
Token-Based	0.56	0.99	0.57	0.82	0.94	0.69
VGDetector	0.56	0.99	0.57	0.84	0.93	0.70
Vuldeepecker	0.36	0.74	0.62	0.37	0.68	0.66
DeepWukong(k-GNNs)	0.93	0.98	0.95	0.93	0.97	0.95

(6) CWE399

	IFN	$\overline{\text{FPR}}$	$\overline{\text{FNR}}$	MKN	ACC	F1
RATS	0.05	0.99	0.06	0.43	0.70	0.11
Flawfinder	0.07	0.29	0.78	0.08	0.44	0.47
Clang Static Analyzer	0.01	0.78	0.23	0.01	0.61	0.27
Infer	-0.02	0.63	0.35	-0.02	0.54	0.32
Token-Based	0.33	0.99	0.34	0.70	0.88	0.48
VGDetector	0.70	0.93	0.77	0.68	0.90	0.73
Vuldeepecker	0.42	0.77	0.65	0.43	0.71	0.69
DeepWukong(k-GNNs)	0.94	0.98	0.96	0.94	0.97	0.96

(7) CWE787

	IFN	$\overline{\text{FPR}}$	$\overline{\text{FNR}}$	MKN	ACC	F1
RATS	0.01	0.99	0.02	0.16	0.73	0.04
Flawfinder	0.09	0.30	0.79	0.09	0.43	0.43
Clang Static Analyzer	0.07	0.88	0.19	0.12	0.69	0.25
Infer	0.08	0.63	0.45	0.07	0.58	0.37
Token-Based	0.32	0.98	0.34	0.69	0.88	0.48
VGDetector	0.74	0.95	0.79	0.70	0.93	0.76
Vuldeepecker	0.59	0.84	0.75	0.60	0.79	0.78
DeepWukong(k-GNNs)	0.93	0.97	0.95	0.91	0.97	0.94

(8) CWE254

	IFN	$\overline{\text{FPR}}$	$\overline{\text{FNR}}$	MKN	ACC	F1
RATS	0.20	0.98	0.22	0.57	0.81	0.34
Flawfinder	0.13	0.31	0.82	0.11	0.43	0.39
Clang Static Analyzer	-0.03	0.84	0.13	-0.04	0.68	0.16
Infer	0.18	0.55	0.63	0.13	0.57	0.40
Token-Based	0.53	0.99	0.54	0.83	0.94	0.67
VGDetector	0.70	0.97	0.73	0.73	0.94	0.74
Vuldeepecker	0.53	0.80	0.73	0.53	0.76	0.75
DeepWukong(k-GNNs)	0.96	0.99	0.97	0.95	0.99	0.97

(9) CWE400

	IFN	$\overline{\text{FPR}}$	$\overline{\text{FNR}}$	MKN	ACC	F1
RATS	-0.03	0.92	0.05	-0.10	0.61	0.08
Flawfinder	0.16	0.23	0.93	0.26	0.48	0.56
Clang Static Analyzer	0.09	0.98	0.11	0.42	0.67	0.19
Infer	-0.19	0.51	0.30	-0.18	0.43	0.28
Token-Based	0.50	1.00	0.50	0.87	0.92	0.66
VGDetector	0.84	0.92	0.92	0.67	0.92	0.79
Vuldeepecker	0.80	0.88	0.92	0.80	0.90	0.91
DeepWukong(k-GNNs)	0.96	0.97	0.96	0.93	0.97	0.96

(10) CWE78

Fig. 11. Comparing DeepWukong with the state-of-the-art vulnerability detection approaches. The 10 subfigures represent the 10 types of vulnerabilities. For a given vulnerability, each cell represents the performance of the approach (y-axis) under the corresponding metrics (x-axis). The darker the cell (the higher the value), the better the performance. Note that, for the FPR and FNR, we present their additive inverse here, which represents 1-FPR and 1-FNR separately. Note that, the minimum negative value we observed is larger than -0.3 so the range is set to [-0.3,1].

Table 6. Runtime costs comparing DeepWukong with state-of-the-art vulnerability detection approaches on CWE254. N/A indicates the phase is not applicable to the approach.

Time	RATS	Flawfinder	Clang Static Analyzer	Infer	Token-based	VGDetector	Vuldeepecker	DeepWukong
training time(s)	N/A	N/A	N/A	N/A	135,973.1	96,388.1	40,072.2	83,248.5
testing time(s)	488.7	451.4	25,584.8	28,537.7	540.6	383.1	172.1	321.1

ANSWER: The performance of DeepWukong is not tied to a particular graph neural network, as all the three networks have shown promising results. k-GNNs outperforms the other two graph neural networks slightly, as it can better capture both the structured and unstructured information of XFG.

5.2 RQ2: DeepWukong VS. Traditional Vulnerability Detection Tools

To answer the second research question, we select four most popular open-source tools, i.e., Flawfinder [5], RATS [8], Clang Static Analyzer [2] and Infer [6], as the baselines for our comparison. These tools are widely used in the software engineering community for static bug detection [46, 47]. They claim that they can cover a wide range of bugs. By looking into the source code of these tools, we find that these frameworks can detect vulnerabilities based on pre-defined detecting rules. In general, they are reported to perform well on low-level well-defined bugs, e.g., buffer overflow, null pointer dereference and format string problems, etc.

Comparison Method. To conduct fair comparison, we have applied all these frameworks to detect vulnerabilities in our labelled benchmark at the XFG level, i.e., predicting an XFG is vulnerable if it contains at least one vulnerable statement flagged by these tools.

Result. As shown in Figure 11, DeepWukong outperforms all these four tools with regard to the evaluation metrics. It is interesting to see that, these static analysis tools have either high false positives or high false negatives. Taking CWE22 vulnerability as an example, over 83% of the samples reported by Flawfinder are false positives. For the CWE119 vulnerability, over 98%

of the samples reported by RATS are false negatives. Among the four traditional tools, RATS and Flawfinder perform worst. RATS has a high false negative rate of 90% on average, while Flawfinder has a high false positive rate of 64% on average. Clang Static Analyzer also poses a high false negative rate of 79% on average, as the detecting rules are not complete and cannot detect high-level bugs. Infer, which relies on Separation logic [48] and Bi-abduction [49], performs relatively better, with a false negative rate at 56% and a false positive rate at 40%. As for the runtime performance (see Table 6), the runtime cost by RATS on CWE254 is 488.7s, similar to that of Flawfinder with a relatively lower figure standing at 451.4s, while Clang Static Analyzer and Infer spend significantly longer time, at 25,584.8s and 28,537.7s, respectively. DeepWukong uses 83,248.51s on training and 321.1s on testing.

Analysis. This result suggests that these tools are hard to be adopted to detect various high-level software vulnerabilities, despite the comparatively low runtime cost. The main reason leading to the result is that they highly rely on well-defined bug patterns or specifications. By manually checking their existing detecting rules, we find that most of the rules are quite simple and the number of rules is also limited. The real-world vulnerabilities, especially for emerging vulnerabilities, are far more complicated than the simple rules defined by these detection tools, which will greatly limit the usage scenarios of the traditional rule-based detection tools, as they are purely relying on experienced human experts and developers to craft the sophisticated analyzers.

> ANSWER: Our experiment results suggest that the traditional rule-based detection approaches are not applicable to detecting real-world complicate vulnerabilities, while DeepWukong, which embeds control- and data-flow information via deep-learning, is effective in pinpointing vulnerabilities in a general manner without the knowledge of any pre-defined anti-patterns.

5.3 RQ3: DeepWukong VS. Existing deep-learning-based approaches

5.3.1 Existing deep-learning-based approaches. We first summarize the following three representative deep-learning-based vulnerability detection approaches.

Token-based embedding [16] was proposed to detect vulnerabilities by representing the source code as sequential tokens. It first generates a token sequence for each method and then embeds raw-text information via deep-learning. As this framework is not open-source, we have re-implemented this detection framework strictly following the approach described in White et al. [16] for comparison. It is worth mentioning that, there exists duplicate samples in the dataset primarily due to shared dependencies of different testcases, so we perform a comparative experiment with or without duplicate methods by applying a LSTM neural network, as shown in Table 7. Taking CWE20, which contains the number of samples above average, as an example, we observed a significant decline after removing all duplicates, with the F1 Score dropping from 77.3% to 58.4%. Regarding accuracy, both figures are relatively high (above 90%), because of the imbalanced distribution of the dataset: the number of safe methods, at 774,714, is more than six times as many as that of vulnerable methods, at 129,038 (a random guess can achieve an accuracy of above 85%).

As this framework performs detection at the method level, after removing all duplicates, we have applied it to 903K methods that we labelled (cf. Table 3) to train the classifier and perform bug prediction.

VGDetector [15] extracts control flow graph of source code and uses graph convolutional network [23] to embed it in a compact and low-dimensional representation. It claims to be well performed on detecting control-flow-related bugs. Note that this framework only works at the method level, thus we have applied it to all the 903K CFGs, we extract and label (cf. Table 3) to perform the model training and prediction.

Table 7. A comparative experiment on dataset with and without the duplicate samples using Token-based embedding.

Category	Duplicate Samples	INF	FPR	FNR	MKN	ACC	F1
CWE119	Without	0.332	**0.017**	0.651	0.690	0.887	0.481
	With	**0.605**	0.032	**0.312**	**0.752**	**0.915**	**0.728**
CWE20	Without	0.434	**0.012**	0.554	**0.769**	0.912	0.584
	With	**0.727**	0.024	**0.249**	0.765	**0.945**	**0.773**
CWE125	Without	0.217	**0.003**	0.780	0.812	0.883	0.351
	With	**0.757**	0.012	**0.231**	**0.826**	**0.925**	**0.806**
CWE190	Without	0.557	**0.014**	0.429	**0.808**	0.932	0.682
	With	**0.770**	0.019	**0.211**	0.805	**0.948**	**0.809**
CWE22	Without	0.479	**0.004**	0.517	**0.851**	0.906	0.643
	With	**0.733**	0.015	**0.252**	0.836	**0.912**	**0.804**
CWE399	Without	0.561	**0.012**	0.427	0.820	0.935	0.693
	With	**0.834**	0.031	**0.135**	**0.837**	**0.942**	**0.867**
CWE787	Without	0.325	**0.015**	0.660	**0.705**	0.879	0.480
	With	**0.544**	0.035	**0.421**	0.689	**0.913**	**0.688**
CWE254	Without	0.323	**0.017**	0.660	0.690	0.878	0.480
	With	**0.569**	0.019	**0.412**	**0.726**	**0.902**	**0.666**
CWE400	Without	0.534	**0.009**	0.457	**0.831**	0.942	0.669
	With	**0.773**	0.015	**0.212**	0.818	**0.943**	**0.813**
CWE78	Without	0.499	**0.004**	0.497	**0.873**	0.921	0.659
	With	**0.756**	0.012	**0.232**	0.845	**0.933**	**0.815**

Vuldeepecker [18] is a recent approach that detects vulnerabilities in source code, which relies on data-dependence information to represent the source code. It first locates bug-related APIs and then extracts the data-flow related program slices based on the parameters of these APIs, thus generating code slices by assembling API-related program slices [18] to enforce accurate bug detection. It uses code slice to pinpoint bugs and the number of generated code slices depends on the number of APIs existed in the source code. Unfortunately, this framework is not open-source. We have made our efforts to reproduce their results by re-implementing their approach by strictly following their algorithm [18]. Note that we have also identified that the dataset [8] released by the author contains lots of *duplicate samples (roughly 50% of the samples are duplicates), which inflate performance metrics, i.e., overfitting problem.* We evaluate Vuldeepecker's performance on the dataset with and without removing the duplicate samples.

Table 8 shows the results of Vuldeepecker with and without duplicate samples. When including the duplicate samples, our implementation achieves similar results as that reported in their paper, which suggests that our reproduced framework correctly follows their algorithm. However, it is surprising to see that, after removing duplicate samples, the performance of Vuldeepecker decreases significantly, e.g., the F1 Score decreases from 92% to 72% for CWE119, and decreases from 95% to 66% for CWE399. More surprisingly, the F1 Score decreases by over 34% to 58% for CWE190. To conduct a fair comparison, we have removed all the duplicate samples, to use the same experimental setup as DeepWukong. We have extracted 138,230 slices (cf. Table 3) to evaluate the performance of Vuldeepecker.

5.3.2 *Result.*

As shown in Figure 11, DeepWukong outperforms all the three referred deep-learning-based approaches when evaluating the 10 types of vulnerabilities against the evaluation metrics.

Taking the CWE254 vulnerability as an example, the false negative rate of the Token-based approach is roughly 66%, while the false negative rate of our DeepWukong is only 5%. Regarding the result of the F1 Score, DeepWukong is roughly 46% higher than the Token-based approach. As

[8] https://github.com/CGCL-codes/VulDeePecker

Table 8. A comparative experiments on datasets with and without the duplicate samples using VULDEEPECKER.

Category	Duplicate Samples	TPR	FPR	FNR	P	ACC	F1
CWE119	Without	0.766	0.359	0.234	0.681	0.704	0.721
	With	**0.956**	**0.124**	**0.044**	**0.885**	**0.916**	**0.919**
CWE20	Without	0.850	0.380	0.150	0.691	0.740	0.772
	With	**0.968**	**0.134**	**0.032**	**0.911**	**0.946**	**0.939**
CWE125	Without	0.741	0.492	0.259	0.595	0.629	0.662
	With	**0.938**	**0.156**	**0.062**	**0.921**	**0.921**	**0.929**
CWE190	Without	0.529	0.292	0.471	0.652	0.623	0.582
	With	**0.958**	**0.196**	**0.042**	**0.891**	**0.912**	**0.923**
CWE22	Without	0.952	0.052	0.048	0.954	0.953	0.951
	With	**0.978**	**0.036**	**0.022**	**0.971**	**0.982**	**0.974**
CWE399	Without	0.615	0.259	0.385	**0.950**	0.704	0.656
	With	**0.957**	**0.059**	**0.043**	0.942	**0.948**	**0.949**
CWE787	Without	0.652	0.231	0.348	0.740	0.714	0.692
	With	**0.948**	**0.136**	**0.052**	**0.921**	**0.898**	**0.934**
CWE254	Without	0.752	0.161	0.248	0.822	0.794	0.781
	With	**0.936**	**0.146**	**0.064**	**0.932**	**0.917**	**0.934**
CWE400	Without	0.732	0.202	0.268	0.782	0.764	0.751
	With	**0.918**	**0.176**	**0.082**	**0.952**	**0.942**	**0.935**
CWE78	Without	0.923	**0.122**	0.077	0.892	0.902	0.913
	With	**0.965**	0.135	**0.035**	**0.922**	**0.931**	**0.943**

for VGDETECTOR, taking CWE399 as an example, the F1 Score of VGDETECTOR is only 70%, roughly 25% lower than DEEPWUKONG. The false negative rate of VGDETECTOR reaches 43%, which is 38% higher than our approach, showing that DEEPWUKONG can precisely detect more bugs. Comparing with VULDEEPECKER, taking CWE190 as an example, VULDEEPECKER nearly misses half of all the vulnerabilities with the false negative rate of 47% and the F1 Score is only 58%. DEEPWUKONG is roughly 36% higher than VULDEEPECKER for the F1 Score and 40% lower for the false negative rate.

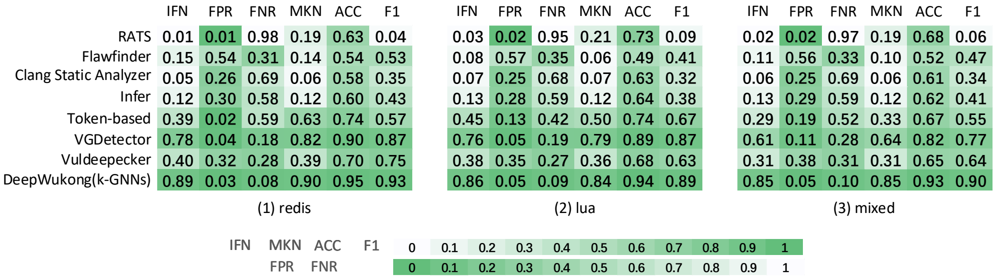
For the running time shown in Table 6, all these approaches generally have a high runtime cost for training but a low runtime cost for prediction. In particular, DEEPWUKONG and VULDEEPECKER take less time than the other two tools.

5.3.3 Analysis. The experimental results suggest that representing the source code using a single code feature, such as raw-text (e.g., TOKEN-BASED embedding), control-flow (e.g., VGDETECTOR) or data-flow (e.g., VULDEEPECKER) is not enough to detect a wide variety of vulnerabilities.

DEEPWUKONG VS. TOKEN-BASED Approach. TOKEN-BASED embedding considers each method of a program as a single big code block (i.e., sequential tokens), while DEEPWUKONG embeds both the textual and structured information of code. The real-world programs are complex and only considering the code as a plain text will miss its data- and control-flow information and can not precisely capture code semantics.

DEEPWUKONG VS. VGDETECTOR. Using control-flow alone to represent source code is also not enough to detect all vulnerabilities. Compared with TOKEN-BASED embedding, VGDETECTOR splits source code into basic blocks and adds control-flow edges between basic blocks, but VGDETECTOR does not conduct data-flow analysis, making the representation less comprehensive. It mainly focuses on dealing with control-flow-related vulnerabilities but can not precisely capture data-dependence which is essential for vulnerabilities such as buffer overflows.

DEEPWUKONG VS. VULDEEPECKER. VULDEEPECKER [18] only considers data-flows while ignoring control-flow information. It claims to perform well on buffer errors and resource management errors because those vulnerabilities are highly related to data-flow. However, the approach is shown ineffective in detecting vulnerabilities related to control-flows such as Insufficient Control



	IFN	FPR	FNR	MKN	ACC	F1
RATS	0.01	0.01	0.98	0.19	0.63	0.04
Flawfinder	0.15	0.54	0.31	0.14	0.54	0.53
Clang Static Analyzer	0.05	0.26	0.69	0.06	0.58	0.35
Infer	0.12	0.30	0.58	0.12	0.60	0.43
Token-based	0.39	0.02	0.59	0.63	0.74	0.57
VGDetector	0.78	0.04	0.18	0.82	0.90	0.87
Vuldeepecker	0.40	0.32	0.28	0.39	0.70	0.75
DeepWukong(k-GNNs)	0.89	0.03	0.08	0.90	0.95	0.93

(1) redis

	IFN	FPR	FNR	MKN	ACC	F1
RATS	0.03	0.02	0.95	0.21	0.73	0.09
Flawfinder	0.08	0.57	0.35	0.06	0.49	0.41
Clang Static Analyzer	0.07	0.25	0.68	0.07	0.63	0.32
Infer	0.13	0.28	0.59	0.12	0.64	0.38
Token-based	0.45	0.13	0.42	0.50	0.74	0.67
VGDetector	0.76	0.05	0.19	0.79	0.89	0.87
Vuldeepecker	0.38	0.35	0.27	0.36	0.68	0.63
DeepWukong(k-GNNs)	0.86	0.05	0.09	0.84	0.94	0.89

(2) lua

	IFN	FPR	FNR	MKN	ACC	F1
RATS	0.02	0.02	0.97	0.19	0.68	0.06
Flawfinder	0.11	0.56	0.33	0.10	0.52	0.47
Clang Static Analyzer	0.06	0.25	0.69	0.06	0.61	0.34
Infer	0.13	0.29	0.59	0.12	0.62	0.41
Token-based	0.29	0.19	0.52	0.33	0.67	0.55
VGDetector	0.61	0.11	0.28	0.64	0.82	0.77
Vuldeepecker	0.31	0.38	0.31	0.31	0.65	0.64
DeepWukong(k-GNNs)	0.85	0.05	0.10	0.85	0.93	0.90

(3) mixed

Fig. 12. Comparing DeepWukong with our baselines on real-world complex applications. Subfigure (1) represents *redis* dataset, subfigure (2) represents *lua* dataset and subfigure (3) represents the mixed dataset of *redis* and *lua*. For a given vulnerability, each cell represents the performance of the approach (y-axis) under the corresponding metrics (x-axis). The darker the cell, the better the performance. Note that, we did not observe negative values for IFN and FNR so the range is set to [0,1].

Flow Management vulnerability. Vuldeepecker also failed to capture the structured information which can lead to imprecise code embedding.

XFG designed in our approach could capture both interprocedural control- and data-flow information of a program. These comprehensive semantic features are highly related to vulnerability patterns, which can be used to detect both low-level vulnerabilities like memory errors and high-level ones such as business logic errors.

ANSWER: DeepWukong preserves the comprehensive code features (i.e., interprocedural control- and data-flow) in the graph embedding, which is able to capture the common characteristics of vulnerabilities, and greatly improve the performance of existing deep-learning-based approaches.

5.4 RQ4: DeepWukong on real-world open-source applications

Comparison Method. To evaluate whether DeepWukong can be applied to real-world complex projects, we evaluate DeepWukong using two real-world open-source applications (Table 4). We compare DeepWukong with our baselines by randomly choosing 80% of the XFGs generated from these two open-source projects for training and the remaining XFGs for detection. We used the same comparison method described in Section 5.2 to compare DeepWukong with traditional bug detectors. Similarly, we also compared DeepWukong with the existing deep-learning-based approaches introduced in Section 5.3. To further understand whether DeepWukong can successfully discover real-world emerging vulnerabilities that are not in the SARD dataset, we evaluate DeepWukong by choosing the latest 5 security-related commits of each open-source project, from which we randomly select 50 vulnerable XFGs, and then input those samples into the model trained from SARD.

Result and analysis. Figure 12 compares DeepWukong with all the seven vulnerability detection approaches by using our vulnerable programs extracted from the two real-world projects. Obviously, DeepWukong outperforms all the other approaches with an average improvement of 61%, 57% and 42% in terms of informedness, markedness and F1 Score, respectively.

Traditional vulnerability detection tools, not surprisingly, have high false positive and false negative rates, showing that they cannot be used to tackle a wide variety of vulnerabilities in the wild, particularly those without well-defined specifications/patterns. The performance of the existing deep-learning-based approaches is also poorer than DeepWukong. The false negative rate of the Token-based approach on *redis* is 59%, which is about seven times higher than that of DeepWukong (8%). VGDetector reports a markedness of 64% on the mixed dataset, while the score for DeepWukong is 21% higher at 85%. As for Vuldeepecker, the F1 Score on the mixed dataset is merely 64% compared to 90% by DeepWukong. When detecting vulnerabilities in the two open-source projects using the model trained from SARD, we achieve an accuracy of 86%, demonstrating that DeepWukong is able to precisely capture real-world emerging vulnerabilities.

> ANSWER: DeepWukong can effectively detect vulnerabilities in real-world open-source applications which require comprehensive code features, and significantly boost the performance of both traditional and deep-learning-based vulnerability detecting approaches.

6 THREATS TO VALIDITY

First, the vulnerability labelling in this work might not be perfect. It is possible that some samples are mislabelled. Here, we trust the labelling results of SARD since they are labelled by domain experts. We also try our best to conduct the dataset labelling for the vulnerable patches from two real-world projects, which takes 720 hours by three experienced engineers. Second, our framework performs program slicing based on function calls and operator statements as the program points of interest. As aforementioned, it might not be perfect, as we may miss some corner cases. One way to improve our approach is to identify other types of program points of interest, and filter irrelevant program points of interest to further eliminate possible noises in the training phase. Third, our experiments are limited to the top 10 vulnerabilities in C/C++ programs. However, it is easy to extend our methodology to support more vulnerabilities and other programming languages. Last, our approach only considers three state-of-the-art graph neural networks for code embedding, it is interesting to further explore more types of neural networks.

7 RELATED WORK

Static vulnerability detection. There are quite a few traditional static program analysis frameworks developed to process source code and report potential vulnerabilities in all kinds of software engineering systems (e.g. Clang Static Analyzer [2], Infer [6], SVF [10], Coverity [3], Fortify [4], Flawfinder [5], ITS4 [7], RATS [8], Checkmarx [9]). Besides, a number of academic researches [50–52] seek to detect specific vulnerabilities (mainly of memory errors and information leaks). These traditional approaches heavily rely on conventional static analysis theories (e.g., data-flow, abstract interpretation and taint analysis) and need human expert to define effective detecting rules. The number of rules is limited and cannot cover all of the vulnerability patterns. As a result, they often suffer from high false positives and false negatives when detecting complex programs, as indicated in our evaluation (cf. Section 5.2).

Similarity-based vulnerability detection. There is a line of research for detecting vulnerabilities by applying similarity analysis (e.g., code clone bugs) [53–55]. They normally represent the code fragment into an abstract representation and compute the similarity between pairs of abstractions. Then they set a similarity threshold and consider the target code as vulnerable if the similarity between the target code fragment and vulnerable ones is above the threshold. This method still requires human to select appropriate code and extract features from it for comparison.

Machine learning based vulnerability detection. There is another line of research recently for automatically detecting vulnerabilities using machine learning. DeepBugs [56] represents code via text vector for detecting name-based bugs. Grieco et al. [11] uses lightweight static and dynamic features to detect memory corruption. Neuhaus et al. [12] use support vector machines (SVM) to analyze code from Red hat packages. Yan et al. [57] perform machine-learning-guided type state analysis for detecting use-after-frees. VGDetector [15] uses control flow graph and graph convolutional network to detect control-flow-related vulnerabilities. Vuldeepecker [18] applies code embedding using data-flow information of a program for detecting resource management errors and buffer overflows. All these solutions can only detect specific well-defined vulnerabilities. Compared with these approaches that mainly focus on detecting limited types of bugs, our approach focuses on detecting multiple vulnerabilities (both low-level and high-level) with low false positive and negative rates.

Machine-learning for program analysis. A number of studies have been proposed to combine machine learning with program analysis to perform better code analysis. The existing approaches have defined sets of features that prove to be useful representing code. They can be mainly divided into four categories. There are approaches that extract features based on texts [58–61], tokens [62–65], ASTs [21, 66–68] and graphs [69–74]. For example, Wang et al. [67] proposed learning from token vectors extracted from Abstract Syntax Trees (ASTs) and use it to realize software defect prediction. White et al. [75] used a stream of identifiers and literal types to automatically encode code for code clone detection. Sui et.al [74] utilized a bag of paths from interprocedural context-sensitive alias-aware value-flow-graph for code classification and summarization. These studies may have a correlation with part of our work, but with different goals. Most of them are focused on code clone detection and defect prediction, which is different from vulnerability detection and in general. Comparing with those code representations, XFG is more fine-grained (code statements VS. program level) and preserves as much high level semantic information as possible. Nevertheless, the different code representation approaches might be a complement to our work. Devign [17] is a very recent work on deep-learning-based vulnerability detection. There are two major differences between Devign and DeepWukong. Their approach aims at pinpointing bugs at the method level, while DeepWukong has a finer granularity (i.e., at program slice level). Their analysis is intra-procedural which does not support inter-procedural analysis, while DeepWukong has conducted a more precise inter-procedural analysis, which is essential for real-world programs since function calls are quite common in modern software projects.

8 CONCLUSION

In this paper, we present DeepWukong, a new deep learning based approach that embeds both textual and code structured features into an effective representation to support detection of a wide range of vulnerabilities. DeepWukong first performs program slicing to extract fine-grained but complicated semantic features, and then combines with graph neural networks to produce compact and low-dimensional representation. We have applied DeepWukong to over 100K vulnerable programs for 10 most popular C/C++ vulnerabilities and 2 real-world open-source projects, and demonstrate that DeepWukong outperforms several state-of-the-arts, including traditional vulnerability detectors and deep-learning-based approaches.

ACKNOWLEDGMENT

This work was supported by the National Key Research and Development Program of China (No.2018YFB0803605), the National Natural Science Foundation of China (grants No.61702045 and No.62072046), and Australian Research Grants (DP210101348).

REFERENCES

[1] American Information Technology Laboratory. NATIONAL VULNERABILITY DATABASE, 2020. https://nvd.nist.gov/.

[2] Clang static analyzer, 2020. https://clang-analyzer.llvm.org/scan-build.html.

[3] Coverity, 2020. https://scan.coverity.com/.

[4] HP Fortify, 2020. https://www.hpfod.com/.

[5] Flawfinder, 2020. https://dwheeler.com/flawfinder/.

[6] Infer, 2020. https://fbinfer.com/.

[7] J. Viega, J. T. Bloch, Y. Kohno, and G. McGraw. Its4: a static vulnerability scanner for c and c++ code. In *Proceedings 16th Annual Computer Security Applications Conference (ACSAC'00)*, pages 257–267, Dec 2000.

[8] RATS, 2014. https://code.google.com/archive/p/rough-auditing-tool-for-security/.

[9] Checkmarx, 2020. https://www.checkmarx.com/.

[10] Yulei Sui and Jingling Xue. SVF: Interprocedural static value-flow analysis in LLVM. In *CC '16*, pages 265–266, 2016.

[11] Gustavo Grieco, Guillermo Luis Grinblat, Lucas Uzal, Sanjay Rawat, Josselin Feist, and Laurent Mounier. Toward large-scale vulnerability discovery using machine learning. In *Proceedings of the Sixth ACM Conference on Data and Application Security and Privacy*, CODASPY '16, pages 85–96, New York, NY, USA, 2016. ACM.

[12] Stephan Neuhaus and Thomas Zimmermann. The beauty and the beast: Vulnerabilities in red hat's packages. In *In Proceedings of the 2009 USENIX Annual Technical Conference (USENIX ATC*, 2009.

[13] Stephan Neuhaus, Thomas Zimmermann, Christian Holler, and Andreas Zeller. Predicting vulnerable software components. In *Proceedings of the 14th ACM Conference on Computer and Communications Security*, CCS '07, page 529–540, New York, NY, USA, 2007. Association for Computing Machinery.

[14] Yonghee Shin, Andrew Meneely, Laurie Williams, and Jason A. Osborne. Evaluating complexity, code churn, and developer activity metrics as indicators of software vulnerabilities. *IEEE Trans. Software Eng.*, 37:772–787, 11 2011.

[15] X. Cheng, H. Wang, J. Hua, M. Zhang, G. Xu, L. Yi, and Y. Sui. Static detection of control-flow-related vulnerabilities using graph embedding. In *2019 24th International Conference on Engineering of Complex Computer Systems (ICECCS)*, pages 41–50, Nov 2019.

[16] Martin White, Christopher Vendome, Mario Linares-Vásquez, and Denys Poshyvanyk. Toward deep learning software repositories. In *Proceedings of the 12th Working Conference on Mining Software Repositories*, MSR '15, pages 334–345, Piscataway, NJ, USA, 2015. IEEE Press.

[17] Yaqin Zhou, Shangqing Liu, Jing Kai Siow, Xiaoning Du, and Yang Liu. Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks. In Hanna M. Wallach, Hugo Larochelle, Alina Beygelzimer, Florence d'Alché-Buc, Emily B. Fox, and Roman Garnett, editors, *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada*, pages 10197–10207, 2019.

[18] 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. *The Network and Distributed System Security Symposium (NDSS)*, 2018.

[19] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg Corrado, and Jeffrey Dean. Distributed representations of words and phrases and their compositionality. In *Proceedings of the 26th International Conference on Neural Information Processing Systems - Volume 2*, NIPS'13, pages 3111–3119, USA, 2013. Curran Associates Inc.

[20] Quoc V. Le and Tomas Mikolov. Distributed representations of sentences and documents. In *ICML*, volume 32 of *JMLR Workshop and Conference Proceedings*, pages 1188–1196. JMLR.org, 2014.

[21] Miltiadis Allamanis, Marc Brockschmidt, and Mahmoud Khademi. Learning to represent programs with graphs. *CoRR*, abs/1711.00740, 2017.

[22] Software Assurance Reference Dataset, 2017. https://samate.nist.gov/SARD/index.php.

[23] Thomas N. Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. *CoRR*, abs/1609.02907, 2016.

[24] Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Liò, and Yoshua Bengio. Graph Attention Networks. *International Conference on Learning Representations*, 2018. accepted as poster.

[25] Christopher Morris, Martin Ritzert, Matthias Fey, William L. Hamilton, Jan Eric Lenssen, Gaurav Rattan, and Martin Grohe. Weisfeiler and leman go neural: Higher-order graph neural networks. *CoRR*, abs/1810.02244, 2018.

[26] Keshav Pingali and Gianfranco Bilardi. Apt: A data structure for optimal control dependence computation. In *Proceedings of the ACM SIGPLAN 1995 Conference on Programming Language Design and Implementation*, PLDI '95, pages 32–46, New York, NY, USA, 1995. ACM.

[27] Ben Hardekopf and Calvin Lin. Flow-sensitive pointer analysis for millions of lines of code. In *Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, pages 289–298. IEEE Computer Society, 2011.

[28] Crispin Cowan, Calton Pu, Dave Maier, Heather Hintony, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, and Qian Zhang. Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *Proceedings of the 7th Conference on USENIX Security Symposium - Volume 7*, SSYM'98, pages 5–5, Berkeley, CA, USA, 1998. USENIX Association.

[29] C. Cowan, F. Wagle, Calton Pu, S. Beattie, and J. Walpole. Buffer overflows: attacks and defenses for the vulnerability of the decade. In *Proceedings DARPA Information Survivability Conference and Exposition. DISCEX'00*, volume 2, pages

119–129 vol.2, Jan 2000.

[30] Antlr, 2014. https://www.antlr.org/.

[31] Mark Weiser. Program slicing. In *Proceedings of the 5th International Conference on Software Engineering*, ICSE '81, page 439–449. IEEE Press, 1981.

[32] Rahul Gupta, Soham Pal, Aditya Kanade, and Shirish K. Shevade. Deepfix: Fixing common C language errors by deep learning. In *Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence, February 4-9, 2017, San Francisco, California, USA.*, pages 1345–1351, 2017.

[33] H. Robbins. A stochastic approximation method. *Annals of Mathematical Statistics*, 22:400–407, 2007.

[34] Cătălina Cangea, Petar Veličković, Nikola Jovanović, Thomas Kipf, and Pietro Liò. Towards sparse hierarchical graph classifiers, 2018.

[35] Keyulu Xu, Chengtao Li, Yonglong Tian, Tomohiro Sonobe, Ken-ichi Kawarabayashi, and Stefanie Jegelka. Representation learning on graphs with jumping knowledge networks. *CoRR*, abs/1806.03536, 2018.

[36] Keyulu Xu, Weihua Hu, Jure Leskovec, and Stefanie Jegelka. How powerful are graph neural networks? *CoRR*, abs/1810.00826, 2018.

[37] H. Ramchoun, M. A. Janati Idrissi, Y. Ghanou, and M. Ettaouil. Multilayer perceptron: Architecture optimization and training with mixed activation functions. In *Proceedings of the 2nd International Conference on Big Data, Cloud and Applications*, BDCA'17, New York, NY, USA, 2017. Association for Computing Machinery.

[38] Shadi A. Aljawarneh, Ali Alawneh, and Reem Jaradat. Cloud security engineering: Early stages of sdlc. *Future Generation Computer Systems*, 74:385 – 392, 2017.

[39] H. H. AlBreiki and Q. H. Mahmoud. Evaluation of static analysis tools for software security. In *2014 10th International Conference on Innovations in Information Technology (IIT)*, pages 93–98, Nov 2014.

[40] Yaqin Zhou and Asankhaya Sharma. Automated identification of security issues from commit messages and bug reports. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2017, page 914–919, New York, NY, USA, 2017. Association for Computing Machinery.

[41] Matthias Fey and Jan E. Lenssen. Fast graph representation learning with PyTorch Geometric. In *ICLR Workshop on Representation Learning on Graphs and Manifolds*, 2019.

[42] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. In Yoshua Bengio and Yann LeCun, editors, *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, 2015.

[43] Marcus Pendleton, Richard Garcia-Lebron, Jin-Hee Cho, and Shouhuai Xu. A survey on systems security metrics. *ACM Comput. Surv.*, 49(4):62:1–62:35, December 2016.

[44] D. M. W. Powers. Evaluation: From precision, recall and f-measure to roc., informedness, markedness & correlation. *Journal of Machine Learning Technologies*, 2(1):37–63, 2011.

[45] Petra Mutzel. Algorithmic Data Science (Invited Talk). In Rolf Niedermeier and Christophe Paul, editors, *36th International Symposium on Theoretical Aspects of Computer Science (STACS 2019)*, volume 126 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 3:1–3:15, Dagstuhl, Germany, 2019. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[46] Yves Younan, Wouter Joosen, and Frank Piessens. Code injection in c and c++ : A survey of vulnerabilities and countermeasures. Technical report, DEPARTEMENT COMPUTERWETENSCHAPPEN, KATHOLIEKE UNIVERSITEIT LEUVEN, 2004.

[47] the UC Davis Reducing Software Security Risk through an Integrated Approach project, 2000. http://seclab.cs.ucdavis.edu/projects/testing/.

[48] Josh Berdine, Cristiano Calcagno, and Peter W. O'Hearn. Smallfoot: Modular automatic assertion checking with separation logic. In Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf, and Willem-Paul de Roever, editors, *Formal Methods for Components and Objects*, pages 115–137, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[49] Cristiano Calcagno, Dino Distefano, Peter O'Hearn, and Hongseok Yang. Compositional shape analysis by means of bi-abduction. In *Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '09, pages 289–300, New York, NY, USA, 2009. ACM.

[50] Fabian Yamaguchi, Christian Wressnegger, Hugo Gascon, and Konrad Rieck. Chucky: Exposing missing checks in source code for vulnerability discovery. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer &#38; Communications Security*, CCS '13, pages 499–510, New York, NY, USA, 2013. ACM.

[51] F. Yamaguchi, A. Maier, H. Gascon, and K. Rieck. Automatic inference of search patterns for taint-style vulnerabilities. In *2015 IEEE Symposium on Security and Privacy*, pages 797–812, May 2015.

[52] M. Backes, B. Köpf, and A. Rybalchenko. Automatic discovery and quantification of information leaks. In *2009 30th IEEE Symposium on Security and Privacy*, pages 141–153, May 2009.

[53] J. Jang, A. Agrawal, and D. Brumley. Redebug: Finding unpatched code clones in entire os distributions. In *2012 IEEE Symposium on Security and Privacy*, pages 48–62, May 2012.

[54] S. Kim, S. Woo, H. Lee, and H. Oh. Vuddy: A scalable approach for vulnerable code clone discovery. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 595–614, May 2017.
[55] Nam H. Pham, Tung Thanh Nguyen, Hoan Anh Nguyen, and Tien N. Nguyen. Detection of recurring software vulnerabilities. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*, ASE '10, pages 447–456, New York, NY, USA, 2010. ACM.
[56] Michael Pradel and Koushik Sen. Deepbugs: A learning approach to name-based bug detection. *Proc. ACM Program. Lang.*, 2(OOPSLA):147:1–147:25, October 2018.
[57] Hua Yan, Yulei Sui, Shiping Chen, and Jingling Xue. Machine-learning-guided typestate analysis for static use-after-free detection. In *Proceedings of the 33rd Annual Computer Security Applications Conference*, ACSAC 2017, page 42–54, New York, NY, USA, 2017. Association for Computing Machinery.
[58] J. Howard Johnson. Identifying redundancy in source code using fingerprints. In *Proceedings of the 1993 Conference of the Centre for Advanced Studies on Collaborative Research: Software Engineering - Volume 1*, CASCON '93, pages 171–183. IBM Press, 1993.
[59] J. Howard Johnson. Substring matching for clone detection and change tracking. In *Proceedings of the International Conference on Software Maintenance*, ICSM '94, page 120–126, USA, 1994. IEEE Computer Society.
[60] J. Howard Johnson. Visualizing textual redundancy in legacy source. In *Proceedings of the 1994 Conference of the Centre for Advanced Studies on Collaborative Research*, CASCON '94, pages 32–. IBM Press, 1994.
[61] C. K. Roy and J. R. Cordy. Nicad: Accurate detection of near-miss intentional clones using flexible pretty-printing and code normalization. In *2008 16th IEEE International Conference on Program Comprehension*, pages 172–181, June 2008.
[62] T. Kamiya, S. Kusumoto, and K. Inoue. Ccfinder: a multilinguistic token-based code clone detection system for large scale source code. *IEEE Transactions on Software Engineering*, 28(7):654–670, 2002.
[63] Z. Li, S. Lu, S. Myagmar, and Y. Zhou. Cp-miner: finding copy-paste and related bugs in large-scale software code. *IEEE Transactions on Software Engineering*, 32(3):176–192, March 2006.
[64] H. Sajnani and C. Lopes. A parallel and efficient approach to large scale clone detection. In *2013 7th International Workshop on Software Clones (IWSC)*, pages 46–52, May 2013.
[65] Hitesh Sajnani, Vaibhav Saini, Jeffrey Svajlenko, Chanchal K. Roy, and Cristina V. Lopes. Sourcerercc: Scaling code clone detection to big-code. In *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, pages 1157–1168, New York, NY, USA, 2016. ACM.
[66] Jian Zhang, Xu Wang, Hongyu Zhang, Hailong Sun, Kaixuan Wang, and Xudong Liu. A novel neural source code representation based on abstract syntax tree. In *Proceedings of the 41st International Conference on Software Engineering*, ICSE '19, pages 783–794, Piscataway, NJ, USA, 2019. IEEE Press.
[67] Song Wang, Taiyue Liu, and Lin Tan. Automatically learning semantic features for defect prediction. In *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, pages 297–308, New York, NY, USA, 2016. ACM.
[68] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. Code2vec: Learning distributed representations of code. *Proc. ACM Program. Lang.*, 3(POPL):40:1–40:29, January 2019.
[69] Kai Chen, Peng Liu, and Yingjun Zhang. Achieving accuracy and scalability simultaneously in detecting application clones on android markets. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 175–186, New York, NY, USA, 2014. ACM.
[70] Mark Gabel, Lingxiao Jiang, and Zhendong Su. Scalable detection of semantic clones. In *Proceedings of the 30th International Conference on Software Engineering*, ICSE '08, pages 321–330, New York, NY, USA, 2008. ACM.
[71] Raghavan Komondoor and Susan Horwitz. Using slicing to identify duplication in source code. In Patrick Cousot, editor, *Static Analysis*, pages 40–56, Berlin, Heidelberg, 2001. Springer Berlin Heidelberg.
[72] Jens Krinke. Identifying similar code with program dependence graphs. In *Proceedings of the Eighth Working Conference on Reverse Engineering (WCRE'01)*, WCRE '01, pages 301–, Washington, DC, USA, 2001. IEEE Computer Society.
[73] Chao Liu, Fen Chen, Jiawei Han, and Philip Yu. Gplag: Detection of software plagiarism by program dependence graph analysis. In *Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, volume 2006, pages 872–881, 01 2006.
[74] Yulei Sui, Xiao Cheng, Guanqin Zhang, and Haoyu Wang. Flow2vec: Value-flow-based precise code embedding. *Proc. ACM Program. Lang.*, 4(OOPSLA), November 2020.
[75] Martin White, Michele Tufano, Christopher Vendome, and Denys Poshyvanyk. Deep learning code fragments for code clone detection. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, ASE 2016, pages 87–98, New York, NY, USA, 2016. ACM.

A GRAPH CONVOLUTIONAL LAYER.

This section introduces the three Graph Convolutional Networks we used in our Graph Convolutional layer.

A.1 Graph Convolutional Network (GCN).

GCN [23] scales linearly with respect to the number of graph edges. It learns hidden layer representations that encode both local graph structure and features of nodes. It introduces a simple and well-behaved layer-wise propagation rule for neural network models which operate directly on graphs. *During our implementation, we consider a multi-layer GCN with the following layer-wise propagation rule* (illustrated by Figure 13):

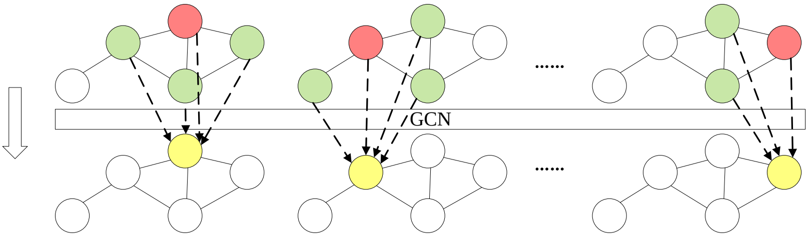


Fig. 13. The illustration of the layer-wise propagation rule of GCN.

$$\mathbf{F}^{(l)} = \sigma\left(\widetilde{\mathbf{D}}^{-\frac{1}{2}}\widetilde{\mathbf{A}}\widetilde{\mathbf{D}}^{-\frac{1}{2}}\mathbf{F}^{(l-1)}\mathbf{W}^{(l)}\right) \tag{8}$$

$\mathbf{F}^{(l)} = \left\{\vec{f}^{(l)}(0), \vec{f}^{(l)}(1), ..., \vec{f}^{(l)}(N)\right\} \in \mathbb{R}^{N \times d^{(l)}}$ means the matrix of activations in the l^{th} layer of the neural network. For the first layer, $\mathbf{F}^{(0)} = \left\{\vec{f}^{(0)}(0), \vec{f}^{(0)}(1), ..., \vec{f}^{(0)}(N)\right\}$ is the nodes feature matrix of the graph. $\widetilde{\mathbf{A}} = \mathbf{A} + \mathbf{I}_N$ represents the adjacency matrix of the graph with added self-connections where $\mathbf{I}_N$ is the identity matrix. $\widetilde{\mathbf{D}}_{ii} = \sum_j \widetilde{\mathbf{A}}_{ij}$ and $\mathbf{W}^{(l)}$ is the trainable weight matrix for the l^{th} layer. We choose Rectified Linear activation function $\sigma(\cdot) = ReLU(\cdot) = max(0, \cdot)$ as our activation function.

A.2 Graph Attention Network (GAT).

GAT [24] is a graph neural network that adopts self-attentional layers, and thus could encode the graph structured by specifying different weights to different neighbors without additional costs. GAT has better performance on inductive and transductive problems. Our approach is branch-aware and the neighboring relationship when meeting a branch condition different from the linearly executing circumstances, requiring (implicitly) different processing. GAT is good at recognizing the different structures of XFG by attending over the features of nodes neighborhoods.

For the l^{th} layer, we first do a shared linear transformation parametrized by a weight matrix, $\mathbf{W}^{(l)}$ for every node. Then the attention coefficients is computed by a shared attentional mechanism $a^{(l)} : \mathbb{R}^{d^{(l-1)} \times d^{(l)}} \rightarrow \mathbb{R}$. We only compute attention coefficients $e_{ij}^{(l)}$ for nodes $j \in N(i)$, where $N(i)$ is the neighbor nodes set of node i, indicating the importance of node j's features to node i.

$$\begin{aligned} e_{ij}^{(l)} &= a^{(l)}\left(\vec{f}^{(l-1)}(i) \cdot \mathbf{W}^{(l)}, \vec{f}^{(l-1)}(j) \cdot \mathbf{W}^{(l)}\right) \\ &= LeakyReLU(\left[\vec{f}^{(l-1)}(i) \cdot \mathbf{W}^{(l)} \parallel \vec{f}^{(l-1)}(j) \cdot \mathbf{W}^{(l)}\right]\vec{a}^{(l)}) \end{aligned} \tag{9}$$

Here $\parallel$ is the concatenation operation. The attention mechanism $a^{(l)}$ is a single-layer feedforward network, parametrized by a weight vector $\vec{a}^{(l)} \in \mathbb{R}^{2d^{(l)}}$, and then applying the LeakyReLU nonlinearity function.

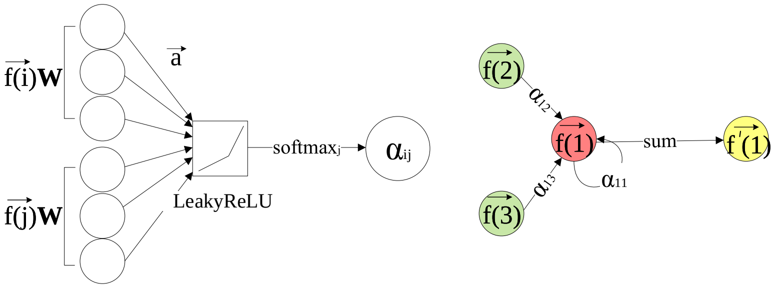


Fig. 14. The illustration of the attention mechanism (left) and propagation rule (right) of GAT.

In order to make coefficients easily comparable across different nodes, we normalize them across all choices of node j by applying softmax function (illustrated by Figure 14 (left)):

$$\alpha_{ij}^{(l)} = \frac{exp(e_{ij}^{(l)})}{\sum_{k\in N(i)} exp(e_{ik}^{(l)})} = \frac{exp(LeakyReLU(\left[\vec{f}^{(l-1)}(i)\mathbf{W}^{(l)} \parallel \vec{f}^{(l-1)}(j)\mathbf{W}^{(l)}\right]\vec{a}^{(l)}))}{\sum_{k\in N(i)} exp(LeakyReLU(\left[\vec{f}^{(l-1)}(i)\mathbf{W}^{(l)} \parallel \vec{f}^{(l-1)}(k)\mathbf{W}^{(l)}\right]\vec{a}^{(l)}))}, \tag{10}$$

Then, we consider a multi-layer GAT with the following layer-wise propagation rule where $\sigma(\cdot)$ denotes an activation function (illustrated by Figure 14 (right)):

$$\vec{f}^{(l)}(i) = \sigma\left(\sum_{j\in N(i)} \alpha_{ij}^{(l)} \vec{f}^{(l-1)}(j) \cdot \mathbf{W}^{(l)}\right) \tag{11}$$

A.3 k-dimensional GNNs (k-GNNs).

k-GNNs [25] is a recent neural network improving handling of higher-order graph structure by taking it at multiple scales into account.

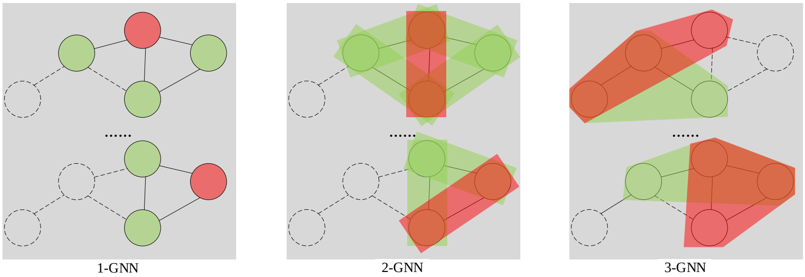


Fig. 15. The illustration of the neighbouring of 1-GNNs, 2-GNNs and 3-GNNs.

The node of XFG is a single statement of the program, making XFG a molecule graph because the size of each node is tiny. In addition, a program can be divided into multiple code blocks (e.g. if-else block, loop block) and each block can be separately considered as a logic unit. The program performs message passing between those code blocks through control- or data-dependence, making the graph representation of the program (i.e., XFG) has higher-order graphic structure. k-GNNs [25] works well with the fine- and coarse-grained structures of a given graph and can capture the code block structured information that is not visible at the single statement-level.

k-GNNs have the same expressiveness as k-dimensional Weisfeiler-Leman graph isomorphism heuristic. We denote the set of nodes and the set of edges of G by $V(G)$ and $E(G)$, respectively. The number of nodes is $N = |V(G)|$. For a given k, $[V(G)]^k = \{U \subseteq V(G) | |U| = k\}$ over $V(G)$ represents all k-element subset. Let $s = \{s_1, ..., s_k\}$ be a k-set in $[V(G)]^k$. The neighborhood of s is defined as (illustrated by Figure 15):

$$N(s) = \left\{ t \in [V(G)]^k \mid |s \cap t| = k - 1 \right\} \tag{12}$$

Further, the local neighborhood $N_L(s)$ consists of all $t \in N(s)$ such that $(v, w) \in E(G)$ for the unique $v \in s \setminus t$ and the unique $w \in t \setminus s$. The global neighborhood $N_G(s)$ then is defined as $N(s) \setminus N_L(s)$.

In each k-GNN layer $l \geq 0$, we compute a feature vector $\vec{f}_k^{(l)}(s)$ for each k-set s in $[V(G)]^k$. For $l = 0$, we set $\vec{f}_k^{(0)}(s)$ by pooling the feature vectors of each node in the k-set. In each layer $l > 0$, we compute new features by

$$\vec{f}_k^{(l)}(s) = \sigma\left(\vec{f}_k^{(l-1)}(s) \cdot \mathbf{W}_1^{(l)} + \sum_{u \in N_L(s) \cup N_G(s)} \vec{f}_k^{(l-1)}(u) \cdot \mathbf{W}_2^{(l)} \right) \tag{13}$$

Moreover, to scale k-GNNs to larger datasets and to prevent overfitting, we omit the global neighborhood of s, i.e., the final propagation rule is,

$$\vec{f}_{k,L}^{(l)}(s) = \sigma\left(\vec{f}_{k,L}^{(l-1)}(s) \cdot \mathbf{W}_1^{(l)} + \sum_{u \in N_L(s)} \vec{f}_{k,L}^{(l-1)}(u) \cdot \mathbf{W}_2^{(l)} \right) \tag{14}$$