Detecting Condition-Related Bugs with Control Flow Graph Neural Network

Jian Zhang
SKLSDE Lab, Beihang University
China
zhangj@act.buaa.edu.cn

Xu Wang*†
SKLSDE Lab, Beihang University
China
xuwang@buaa.edu.cn

Hongyu Zhang
Chongqing University
China
hyzhang@cqu.edu.cn

Hailong Sun
SKLSDE Lab, Beihang University
China
sunhl@act.buaa.edu.cn

Xudong Liu*
SKLSDE Lab, Beihang University
China
liuxd@act.buaa.edu.cn

Chunming Hu*
SKLSDE Lab, Beihang University
China
hucm@buaa.edu.cn

Yang Liu
Nanyang Technological University
Singapore
yangliu@ntu.edu.sg

ABSTRACT

Automated bug detection is essential for high-quality software development and has attracted much attention over the years. Among the various bugs, previous studies show that the condition expressions are quite error-prone and the condition-related bugs are commonly found in practice. Traditional approaches to automated bug detection are usually limited to compilable code and require tedious manual effort. Recent deep learning-based work tends to learn general syntactic features based on Abstract Syntax Tree (AST) or apply the existing Graph Neural Networks over program graphs. However, AST-based neural models may miss important control flow information of source code, and existing Graph Neural Networks for bug detection tend to learn local neighbourhood structure information. Generally, the condition-related bugs are highly influenced by control flow knowledge, therefore we propose a novel CFG-based Graph Neural Network (CFGNN) to automatically detect condition-related bugs, which includes a graph-structured LSTM unit to efficiently learn the control flow knowledge and long-distance context information. We also adopt the API-usage attention mechanism to leverage the API knowledge. To evaluate the proposed approach, we collect real-world bugs in popular GitHub repositories and build a large-scale condition-related bug dataset. The experimental results show that our proposed approach significantly outperforms the state-of-the-art methods for detecting condition-related bugs.

*Also with Zhongguancun Laboratory, Beijing, P.R.China.
†Corresponding author: Xu Wang, xuwang@buaa.edu.cn.

ISSTA '23, July 17–21, 2023, Seattle, WA, USA

ACM ISBN 979-8-4007-0221-1/23/07…$15.00
https://doi.org/10.1145/3597926.3598142

CCS CONCEPTS

• **Software and its engineering → Software maintenance tools**.

KEYWORDS

Bug detection, graph neural network, deep learning

ACM Reference Format:
Jian Zhang, Xu Wang, Hongyu Zhang, Hailong Sun, Xudong Liu, Chunming Hu, and Yang Liu. 2023. Detecting Condition-Related Bugs with Control Flow Graph Neural Network. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '23), July 17–21, 2023, Seattle, WA, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3597926.3598142

1 INTRODUCTION

Bug detection plays a critical role in software development and maintenance [16, 19, 21]. Many automated techniques have been proposed to help developers detect various types of bugs in source code [8, 11, 12, 27, 28, 32, 37]. Conditional expressions are shown to be one of the most error-prone code elements and may contain condition-related bugs [34, 50]. A manual study on 369 real bugs from five open-source projects shows that 45% of these bugs occur in *if* conditional statements [10]. For example in Figure 1(a), the conditional expression "*theads.isEmpty()*" in line 3 is buggy as it leads to the non-stop deadlocked threads and wrong monitoring of thread status. Previous research on condition-related bugs mainly focuses on how to fix them with automatic program repair techniques [49, 50]. Due to the popularity and possible severe consequences of condition-related bugs, it is also important to design an effective detector for them.

Detecting condition-related bugs is challenging. Manually designing test cases for high multiple condition coverage is time consuming and error prone. Existing tool support for automatic test case generation is also limited. Traditional static analysis based approaches such as FindBugs [20] heavily depend on manual effort to define common bug patterns or rules [20, 24, 35, 48]. Some approaches use data mining or NLP techniques to automatically extract features of different bugs [8, 28, 31, 42, 43]. However, these

approaches learn rules based on token occurrences and can only capture shallow semantics. Therefore, it is necessary to characterize the condition-related bugs and explore how to automatically detect them more effectively.

Recently, deep learning (DL) based approaches have shown promising results. These approaches apply end-to-end neural networks on the Abstract Syntax Tree (AST), Program Dependence Graph (PDG) or their hybrid graphs (AST as the backbone with edges introduced by control flow, data flow, define-use dependencies, etc.), and detect bugs by incorporating syntactic or dependency information [6, 9, 27, 37, 44, 51, 60]. Although effective, the DL-based bug detection models have two main limitations for the condition-related bugs. First, AST-based neural models [27, 37] may miss important control flow information of source code and have a poor performance on detecting condition-related bugs, since the conditional expressions can directly influence the execution of control flow statements. Second, the existing PDG and hybrid graph-based neural models [6, 9, 44, 51, 60] usually adopt general Graph Neural Networks (GNNs) [25] for program representation and may fail to capture the long-distance dependency information, because such GNNs only learn and propagate local neighbourhood structure information within a limited iteration number [13, 52].

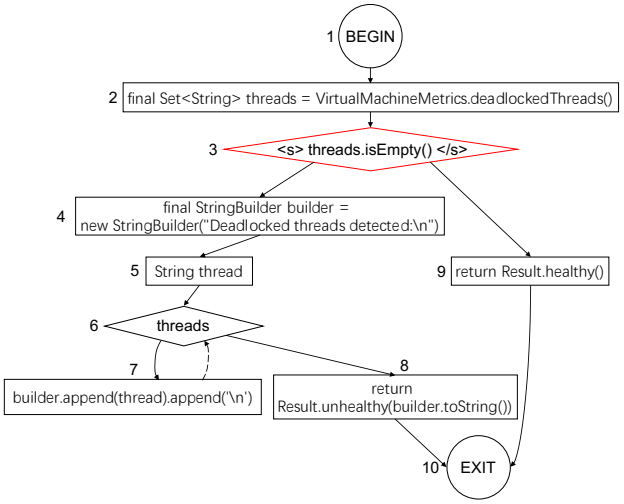
In this work, we propose a novel CFG-based Graph Neural Network (CFGNN) model, which leverages control flow paths of CFGs and API knowledge to automatically detect the condition-related bugs. Here the control flow path refers to a finite sequence of non-repeating edges from the BEGIN node to the EXIT node in a CFG (as depicted in Figure 1(c)), each of which covers at least one new edge. As conditional statements can influence the nodes and control flow structures of CFGs, our model captures the node and the control flow knowledge of CFGs as the context information through graph embedding. Different from existing GNNs that learn structural information with random node order and local neighbourhood propagation within a fixed iteration number, we train our model along all possible control flow paths through graph-structured LSTM units. Note that since these control flow paths usually contain many common nodes (such as node 1, 2, and 3 in Figure1(c)), we neither parse CFG into parallel paths nor train multiple LSTM models over all paths separately, but share the model parameters of these nodes by one single LSTM unit when traversing the CFG. In this way, the control flow structure and the long-distance dependency information can be efficiently captured. Moreover, our work is also motivated by the fact that conditional expressions often contain API calls, and the incorrect API usages in the control flow paths can make buggy conditions more noticeable. For example, the API calls "*threads.isEmpty()*" of node 3 and "*Result.healthy()*" of node 9 are obviously wrong, since we should not report the healthy status if deadlocked threads exist (that is, threads are not empty). Therefore, we incorporate API usage information through the attention mechanism [7] to further improve the performance. Based on the embeddings of our CFGNN model, we check whether there exists a condition-related bug or not.

More specifically, as shown in Figure 1, given a Java method and the buggy conditional expression (Figure 1(a) line 3), we first parse it into a CFG (Figure 1(b)) and add symbols $< s >$ and $< /s >$ to indicate the position of the conditional expression (node 3, marked by the red line). Then we encode the statements as node embeddings using Bidirectional Long Short-Term Memory (BiLSTM) [18] to learn the sequential information of tokens. On top of the node embeddings, we capture the structural information along the control flow paths of CFGs with our single graph-structured LSTM unit, which updates LSTM hidden states based on the forward and backward edge directions of CFGs. In addition, we design an API-usage attention module by combining the attention module with API annotations of nodes (i.e. statements).

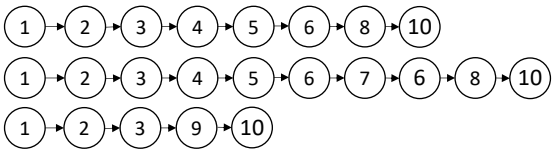
To evaluate the proposed approach, we construct our dataset based on the Java methods collected from high-quality Github projects. The dataset contains 573,332 buggy and non-buggy conditional expressions. We then conduct extensive experiments and the results demonstrate the effectiveness of our CFGNN model. We achieve an F1-score of 46.2% for detecting the condition-related

```
1  public Result check() throws Exception {
2          final Set<String> threads = VirtualMachineMetrics.deadlockedThreads();
3  -       if (threads.isEmpty()) {
   +       if ( ! threads.isEmpty()) {
4              final StringBuilder builder =
5                      new StringBuilder("Deadlocked threads detected:\n");
6              for (String thread : threads) {
7                  builder.append(thread).append('\n');
8              }
9              return Result.unhealthy(builder.toString());
10         }
11         return Result.healthy();
12 }
```

(a) A buggy conditional expression (line 3)



(b) The CFG of the Java method



(c) Control flow Paths in the CFG

Figure 1: An example of condition-related bug, where 1, 2, ..., 10 are the indexes of CFG nodes.

bugs, which significantly outperforms the existing state-of-the-art models such as the AST-based, hybrid graph-based, and other CFG-based deep learning models [6, 27, 44].

In summary, this paper makes the following contributions:

- We propose a novel neural model, CFG-based Graph Neural Network (CFGNN), which can capture the global control flow information and leverage the API knowledge in CFGs;
- We provide a large-scale dataset and apply the proposed CFGNN to detect the condition-related bugs;
- We conduct extensive experiments to evaluate our approach, and the results show that the proposed approach is effective and outperforms all baselines.

The remainder of this paper is organized as follows. Section 2 presents the problem definition. Section 3 describes the details of our approach. Section 4 describes our dataset, evaluation procedure, and evaluation results. We also discuss the practicality and limitations in this section. We describe threats to the validity of this work and related work in Section 5 and Section 6, respectively. Finally, we conclude our paper in Section 7.

2 PROBLEM DEFINITION

In this section, we formulate the problem of condition-related bug detection and discuss how to choose the context representation of source code for this task.

2.1 Condition-Related Bug

Conditional expressions are widely-used in the headers of conditional statements such as *if*, *while* and *for* statements, which determine whether the following statements in the bodies will be executed or not. If conditional expressions are incorrectly written, the functionality of source code will significantly change. Thus it is important to detect such condition-related bugs. We define the condition-related bugs as one type of errors that relate to at least one buggy condition in the source code.

Generally, in order to fix a condition-related bug, the corresponding condition expression and other related code lines should be modified, deleted, or added. All these source code lines and the condition expression may be in one same method, multiple methods, or even multiple files. However, the inter-procedural condition-related bugs are involved with more contextual information and too complex to be effectively detected than the intra-procedural ones. Therefore, in this work we focus on the condition-related bug detection where the condition expression and all its related code are **within the same method**.

As Figure 1(a) illustrates, suppose there is a suspicious conditional expression (i.e. "*theads.isEmpty()*") in line 3 within the context of a Java method, our goal is to detect whether it is buggy. More formally, let $M = \{t_1, ..., t_L\}$ denotes the method, where $t_i, i \in [1, L]$ is one token in it and L is the total length. Given the position (b, e) of an conditional expression $C = \{t_b, ..., t_e\}, b, e \in [1, L]$ from M, the target is to give a result $Y \in \{0, 1\}$ based on the context of C and M (e.g., the conditional probability $P_{buggy}(C|M) \in [0, 1]$ for probabilistic approaches), where $Y = 1$ means there is such a bug, otherwise $Y = 0$. Since different positions of C can yield different values of Y, we add symbols $< s >$ and $< /s >$ to the start and end of the conditional expression to mark its position. Note that the conditional expression C may consist of multiple sub-conditions as defined in program language specification [14]. Furthermore, if there are multiple conditional statements in one same method, we can detect them one by one for multiple times.

In this way, we can propose automated approaches to help developers find buggy conditions, which further provides clues for the logic errors in the code, especially at the development stage.

2.2 Design Rationale

In order to detect the condition-related bugs with specified condition expressions and their contextual methods, we should choose a good context representation of source code as the raw input of our approach. In this section, we discuss the rationale of choosing CFG as the code representation and the rationale of proposing a new GNN model.

Compared with the original token sequences of source code, **AST** is a kind of tree aimed at representing the abstract syntactic structure of the source code [56]. Recently many deep learning-based bug detection approaches utilize the syntactical information of ASTs to learn the semantics of buggy methods [27, 37]. Although it has been shown effective in detecting general or other different types of bugs, AST may lose more important features of condition-related bugs since neither the local nor global control flow is included, whereas such information is a key characteristic for the accuracy of bug detection [45, 46]. Our intuition for the problem is that the conditional statements highly dominate the execution of other statements, and thus it is more useful to consider the control flow information.

The control flow and data flow dependencies of source code are widely represented by **PDGs**. Precise construction of PDGs usually rely on the compiled code and limits the application of many scenarios involving incomplete source code [6, 27, 40]. If needed, the simplified and approximate version of PDGs can be obtained by ignoring the unknown code elements such as variable, method invocation, and so on [57]. In addition, the hybrid graphs consisting of ASTs and edge dependencies such as control flow, data flow and define-use, are also used to represent more code semantics [6, 9, 44, 51, 60]. However, most of these PDG and hybrid graph-based methods do not highlight the importance of control flow information. For example, code property graphs [51, 60] take AST as the backbone and treat the control flow dependencies as only one kind of edges, which may work well for other bug types but not for condition-related bugs. Furthermore, related work such as [60] adopts the Gated Graph Neural Network (GGNN) [25], which may not well capture the long-distance dependency information, since GGNN encodes all the local nodes in one iteration and the global dependence is limited by the iteration number [13, 52].

As described above, the condition-related bugs are highly influenced by the control flow information, which can be captured by a **CFG**. Also, CFGs can be very easily extracted from source code even when the code fragments are incomplete (i.e. single methods). Thus we prefer CFGs as the raw input of our approach. To avoid the weakness in long-distance dependency of existing GGNNs on PDG and hybrid graphs, we design a novel Graph Neural Network for CFGs, aiming to better model the control flow paths of CFGs. We also incorporate API knowledge to augment the capability of extracting API-related features from the contexts.

3 APPROACH

3.1 Overview

In this section, we introduce our approach to condition-related bug detection. Our model is based on the neural network that can automatically detect the buggy condition expressions at the statement level. As described previously, to detect the potential condition-related bugs of a method, our model requires one specified condition expression and its context code in terms of the method. A wider application scenario is that we can check all the condition expressions in a given method one by one if multiple conditional expressions exist in the method.

Figure 2 shows the overview of our approach. We design an end-to-end neural architecture named CFG-based Graph Neural Network (CFGNN), including preprocessing, node embedding, control flow encoding and API-usage attention mechanism. In the preprocessing phase, we parse the given method into CFG and tokenize the statements of the nodes by existing tools (the details of these tools will be described in Section 4). We mark the position of one specified condition expression by adding some special symbols. For instance, we add the tokens $< s >$ and $< /s >$ to indicate the condition node under inspection. Then the preprocessed CFG is taken as input by CFGNN.

In order to facilitate the description, we define the notations as follows. Let $G = (V, E)$ denotes the CFG of the given method, where $V = \{n_1, ..., n_K\}$ is the set of nodes $n_i, i \in [1, K]$, $E = \{f_1, ...f_{T_1}, b_1, ..., b_{T_2}\}$ is the union set of forward edges $E_F = \{f_i, i \in [1, T_1]\}$ and backward edges $E_B = \{b_i, i \in [1, T_2\}]$. The E_F usually contains the edges for sequential and branch statements, and E_B is usually composed of the loop-back edges. We describe the components of our model below.

3.2 Node Embedding

Apart from those who treat a basic block as a sequence of statements, many of existing approaches also treat it as one statement, for example, using Spoon [44] or Joern [51, 57]. Therefore, in our work, a node of the CFG built from one Java method is a statement (see Figure 1). The semantics of nodes is fundamental for checking the functionality of the whole graph. Existing work either treats them as independent IDs [40] or bag of words [6], which is not sufficient for representing the meaning of a node since the sequential naturalness matters [17]. Also, the buggy nodes can disrupt the correct control flow by a large margin especially for the change of conditional expressions. Hence, we adopt the LSTM [18] to model the sequences of tokens in CFG nodes for better capturing the semantics of statements. At the same time, in order to distinguish whether the node after the branch statement is inside or outside its body (that is, the condition is true or false), we add a "T" or "F" before the node accordingly.

Given a node $n_i = \{c_{i_1}, ...c_{i_L}\}$ with the length i_L (i.e., the number of tokens in it), each of which is first embedded into a vector via a randomly initialized embedding matrix W_e, that is, $x_{i_t} = W_e c_{i_t}$. The token embeddings will be updated during the training procedure to yield the best W_e. Afterwards, we use the LSTM network to encode n_i based on the token embeddings. To further enhance the capability of capturing the context information within long token sequences, we adopt the Bidirectional LSTM (BiLSTM) [39] to obtain the hidden states:

$$h_{i_t} = BiLSTM(x_{i_t}, h_{i_{t-1}}). \quad (1)$$

We compute the average value of the hidden states to get the vectors of individual nodes. Next, we encode the control flow of the CFG to learn the structural information among the nodes.

3.3 Control Flow Encoding

The intra-procedure control flow of a single method is represented by the paths from the BEGIN node to the EXIT node without repeated edges in the CFG. Obviously, the edge directions, node orders and the long-term dependencies between two nodes in the control flow paths are important for capturing the global control flow structure since the control dependencies are embedded in them. However, existing GGNNs [6, 44] destroy the node order by treating the nodes separately. Moreover, take the CFG in Figure 1(b) as an example, after 2 iterations, we can see that GGNN can only learn the information propagated from local neighbourhood nodes within 2 steps in Figure 3(a). For the possible long-distance dependencies such as node 8 and node 2, GGNN must increase the iteration number to get more information from remote nodes. Due to the aggregation operation of each node (regardless of which implementation of GGNN variants), the propagated node information gets diluted when the intermediate nodes repeatedly compute during multiple iterations. That is, if we want node 8 to get the information from node 2, the aggregation operations of node 6 will be computed for five times, node 5 for four times and so on. In this way, the information of node 2 will be absorbed by the intermediate nodes. Thus GGNN usually achieves the best performance when the iteration number is not very big. This makes it cannot capture the long-term dependencies well, which is also observed in Section 4.

To avoid this limitation of existing GGNNs, we propose a novel method to encode the control flow paths in a CFG. Intuitively, for each control flow path (as shown in Figure 1(c)), we can directly adopt LSTM to encode it and learn the long-term dependencies. But that will lose the global structure of CFG and result in much repeated computation. Also, considering that the control flow paths of one CFG usually share many common nodes, we do not train different LSTM models for different control flow paths. Instead, the model parameters of these nodes are shared through one single graph-structured LSTM unit during the traversal of the graph. Therefore, the control flow knowledge can be efficiently captured even for the long-distance dependencies.

Specifically, suppose we have the vectors of nodes $\{x_1, ...x_K\}$, the forward edges E_F and the backward edges E_B, we first convert the edges into adjacency matrices $A_F = symmetric(E_F), A_B = symmetric(E_B)$, where the dimension of one matrix is $D|K| \times D|K|$ and the nodes are arranged by the position they appear in the CFG. Then we design a graph-structured LSTM unit by adapting the standard LSTM [18] to graph topology for embedding the CFG. Fundamentally, our model differs from standard LSTM in that we compose such units to a graph structure by the breadth-first traversal from BEGIN to EXIT nodes, whereas a standard LSTM composes the units in a sequential way. We start the process by initializing the hidden states and memory cells with zeros, which will be jointly updated based on the transitions of forward edges and backward

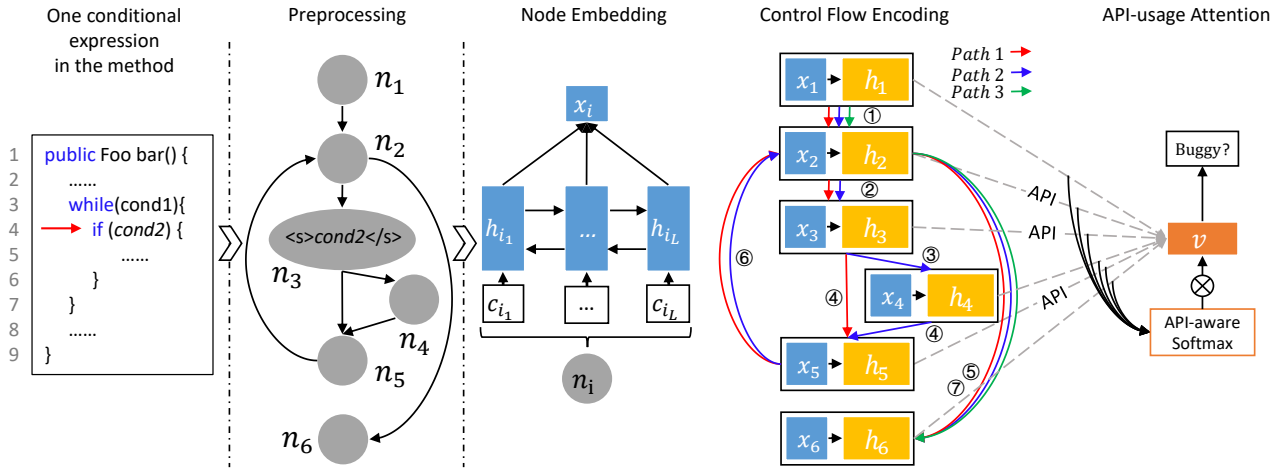


Figure 2: Overview of our approach. The red arrow in the left denotes the condition expression "cond2" under inspection. The circled numbers indicate the encoding order of the edges. The "Path 1" is ($n_1 \rightarrow n_2 \rightarrow n_3 \rightarrow n_5 \rightarrow n_2 \rightarrow n_6$), "Path 2" is ($n_1 \rightarrow n_2 \rightarrow n_3 \rightarrow n_4 \rightarrow n_5 \rightarrow n_2 \rightarrow n_6$), and "Path 3" is ($n_1 \rightarrow n_2 \rightarrow n_6$). The "API" annotation in the right denotes that the corresponding node contains an API use.

edges successively. Since the loops may exist in CFGs, such as the forward and backward edges between node 6 and node 7 in Figure 1, we encode the forward and backward edges separately to avoid the deadlocks during training and can perform batch training easily.

For the forward edges, we consider multiple paths together to calculate the hidden state h_k^f of the k-th node. First, we obtain the hidden states of its predecessors by matrix multiplication, that is:

$$\widetilde{h_k^f} = A_F(k)[h_1^f \cdots h_K^f] \tag{2}$$

$A_F(k)$ denotes the direct predecessors of the node in the k-th column of A_F. Then, we adapt the formulas of a standard LSTM unit to support the previous hidden state $\widetilde{h_k^f}$ and the aggregation of multiple memory cells by the following equations:

$$\begin{aligned}
i_k &= \sigma(W_i x_k + U_i \widetilde{h_k^f} + b_i), \\
f_k &= \sigma(W_f x_k + U_f \widetilde{h_k^f} + b_f), \\
o_k &= \sigma(W_o x_k + U_o \widetilde{h_k^f} + b_o), \\
u_k &= tanh(W_u x_k + U_u \widetilde{h_k^f} + b_u), \\
c_k^f &= i_k \odot u_k + \sum_{p=1}^{P_k} f_k \odot c_p^f, \\
h_k^f &= o_k \odot tanh(c_k^f).
\end{aligned} \tag{3}$$

Here c_p^f is the memory cell of one predecessor of node k for the dependency information and P_k is the number of the predecessors. Other notations are similar to the standard LSTM, for example, i_k, f_k, o_k are the input gate, forget gate and output gate respectively. σ is an activation function and $\odot$ is element-wise multiplication. $W_i, W_f, W_o, W_u, U_i, U_f, U_o, U_u$ are weight matrices and b_i, b_f, b_o, b_u are bias terms. As shown in the top half of Figure 3(b), such node information propagation along all forward edges can better capture long-term dependencies than GGNN. For example, node 8 can eventually get the information from node 1. This process encodes the first and third paths of Figure 1(c) together since there are only forward edges in them.

After that, we further capture the backward flow of the node and combine it with the forward one to obtain the new hidden states h_k for the integral control flow information:

$$\begin{aligned}
h_k^b &= A_B(k)[h_1^f \cdots h_K^f], \\
h_k &= h_k^f + h_k^b.
\end{aligned} \tag{4}$$

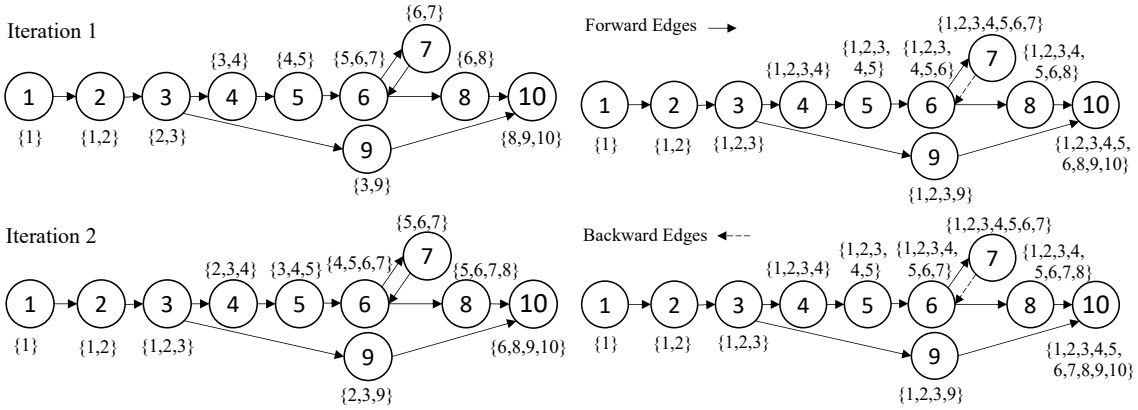
For instance in the bottom of Figure 3(b), the information of node 7 is transferred to node 6 by a backward edge and is further combined with node 8 and node 10 along forward edges. Eventually, the second path of Figure 1(c) is encoded.

Similar to BiLSTM, we also enhance the context information of each node. We get the additional information through reverse traversal. That is, we transpose the matrices A_F and A_B, and calculate the hidden states $\overleftarrow{h_k}$ from n_K to n_1 with Equations 3 and 4. The two kinds of hidden states $\overrightarrow{h_k}$ and $\overleftarrow{h_k}$ are concatenated as the node representations.

Compared with existing GGNNs, the information of nodes in control flow paths is efficiently propagated through the forward and backward edges, even for the long-term dependencies. In this way, our CFGNN model can learn the control flow knowledge better than the related models.

3.4 API-Usage Attention Mechanism

The usefulness of APIs has been stressed by many studies in various source code-related tasks [15, 58, 59]. Because conditional expressions often contain API calls, and the usage patterns of API-related statements implicitly exist in each control flow path. If there is an anti-pattern in one control flow path, it will be beneficial to capture



(a) Graph Embedding in GGNN with iteration number of 2

(b) Control Flow Encoding of our CFGNN

Figure 3: Comparison of the node information propagation in existing GGNN and our CFGNN, where the nodes within braces mean that the information from these nodes has been propagated to the current node.

it for detecting the condition error. For example, in Figure 1, when the condition "*threads.isEmpty()*" of node 3 is false, the API-related statement "*return Result.healthy()*" of node 9 is obviously wrong since at least one deadlocked thread exists. In contrast, the API usage between node 3 and node 8 "*Result.unhealthy()*" conforms to the anti-pattern that reports an unhealthy warning when no deadlocked threads exist.

To help better utilize the knowledge of APIs for capturing the interconnections between conditional and API-related nodes, we design an API-usage attention module. The attention mechanism has been demonstrated effective in learning essential code semantics [54, 55]. Hence, we adapt it by combining the attention weights with API annotations.Specifically, we annotate the nodes $\{n_1, ..., n_K\}$ with binary signs $\{s_1, ..., s_K\}$. We mark the API-related nodes with 1s and the others with 0s, which are used as one input for the attention module. Then we design an API-aware softmax function to adjust the scores of API-related nodes, so that the model can better learn the API usage in the control flow paths. We first get the max value of their hidden states as the query vector q and then compute the scores of each node with the following equations:

$$\begin{aligned} g_k &= \sigma(W_g q + U_g h_k + b_g) \odot (W_q s_k + b_q), \\ \alpha_k &= \frac{exp(g_k)}{\sum_{k=1}^{K} exp(g_k)}, \\ v &= \sum_{k=1}^{K} \alpha_k h_k, \end{aligned} \tag{5}$$

where W_g, U_g, W_q are weight matrices and b_g, b_q are bias terms. Here g_k is the importance score automatically adjusted by a linear function with s_k as the parameter, which incorporates the annotation information of API-related nodes. When the mark is 1, its score will be higher than that of the statement marked with 0, so it can automatically distinguish the API call statements in the control flow path, and pay more attention to the abnormal usage of APIs.

The probability of the condition-related bug for the marked node of the code fragment is calculated by

$$\hat{y} = sigmoid(W_p v + b_p) \in [0, 1], \tag{6}$$

where W_p is the weight matrix and b_p is a bias term. To train the model, we employ the binary cross-entropy loss that is defined as

$$\mathcal{L}(\Theta, \hat{y}, y) = \sum_{n=1}^{N} (-(y_n \cdot log(\hat{y}_n) + (1 - y_n) \cdot log(1 - \hat{y}_n))), \tag{7}$$

where y is the ground truth for each sample, Θ represents the parameters to be learned and N is the total number of instances in the training set. During testing, we make the prediction as $Y = 1$ if $\hat{y}_i \geq \delta$ otherwise $Y = 0$, where δ is the threshold.

4 EVALUATION

4.1 Dataset

Our dataset is constructed based on real-world data collected from GitHub. More specifically, we first downloaded the top 2k popular Java repositories without forked ones based on their stars and forks, then retrieved the commit histories of these repositories.

We built a tool to automatically identify bug-fixing commits for condition-related bugs. For each repository, we check the commits and corresponding commit messages. Following common practice [27, 38, 58], we use issue IDs and keywords to identify the bug-fixing commits. We treat a commit as bug-fixing commit if it contains an issue ID and its commit message contains one of the following keywords: {fix, solve, error, bug, issue, fault, mistake, incorrect}. To further improve the data quality, we apply a recent BERT-based tool [53] to filter out the commits that are not bug-fixing. Next, we fetch the buggy version of the source code by rolling the commits back. After that we scan all the patches in them and extract the methods, where each method contains at least one change limited to a conditional expression, for example, line 3 in Figure 1(a). The buggy conditional statements of them are treated as positive samples for building the dataset, whose labels are 1s. In order to get the

Table 1: The statistics of the dataset

Data	Number	
Source Code	#TotalConditions	573,332
	#BuggyConditions	77,689
	#NormalConditions	495,643
	MaxLen	16,705
	AvgLen	82.7
	#UniqTokens	533,258
CFG	MaxNodeNum	5,422
	AvgNodeNum	37.5
	MaxNodeLen	397
	AvgNodeLen	6.5

negative samples, we extract all the normal conditional statements from the files that buggy conditions belong to, and randomly select one from each method. These negative samples are labeled as 0s. After removing duplicated samples, we got 573,332 samples as the dataset for evaluation, including 77,689 positives and 495,643 negatives.

The detailed statistics of our curated dataset are listed in Table 1. For source code, MaxLen and AvgLen denote the maximal and average length of each method in terms of tokens, repectively. #UniqTokens means the number of all unique tokens. For the corresponding CFGs, MaxNodeNum and AvgNodeNum denote the maximal and average number of nodes in each CFG. MaxNodeLen and AvgNodeLen represent the maximal and average number of tokens in each node.

4.2 Experimental Setup

We conduct experiments on the prepared dataset to evaluate the effectiveness of our proposed model in detecting condition-related bugs. The dataset is split into training set, validation set, and testing set with fractions of 80%, 10%, and 10%, respectively. We use *Spoon* [36] to parse all the Java methods into CFGs and tokenize the statements of the nodes with *javalang* [5]. The tools can direct parse source code without compilation, which enable us to deal with arbitrary code fragments.

The configurations of our models are as follows. We limit the vocabulary size to 100k by keeping the most frequent words because too large vocabulary may lead to poor performance. The out-of-vocabulary words are replaced with *UNK*. We set the maximum number of nodes and the length of statements (in terms of #tokens) of the CFGs to 150 and 20, because such settings cover most elements of CFGs. We set the embedding size and the dimensions of hidden states to 128. The batch size is set to 64 and the maximum epoch is 10. We use the best trained parameters of the 10 saved checkpoints for the later prediction according to their performances on the validation set. We adopt the widely-used Adam [23] as the optimizer with learning rate 0.001 for training our model. The threshold δ for predicting labels is 0.5 by default.

All the experiments are conducted on a Ubuntu 16.04 server with 16 cores of 2.4GHz CPU, 128GB RAM and a Tesla V100 GPU with 32GB memory.

4.3 Baselines

We consider recent bug detection approaches that are applicable for detecting bugs at the statement or method level, and adopt the commonly-used metrics including precision, recall and F1-score for comparison. We also consider two representative models that detect vulnerabilities. To help the baselines detect buggy conditional expressions, we insert symbols around their corresponding tokens or nodes as our model does. We briefly introduce the baselines below.

- **Bugram**. Bugram [42] is an n-gram language model based bug detection technique that builds the n-grams to assess the token sequences of source code with their probability in the learned model, and report low probability sequences as potential bugs. We built a 3-gram language model as recommended in the paper and treat it as a valid report if one sequence exists in the target condition.
- **DeepBugs**. DeepBugs [37] is a learning approach to name-based bug detection that translates each code example into a vector by embedding identifier names from ASTs. We set the embedding size to 128 and keep other settings as default.
- **LC-Attention**. LC-Attention is a neural representation of local context for bug detection [27]. It extracts the paths of ASTs and uses the attentional GRU layer and Convolutional layer with a Multi-Head Attention to capture the syntactic structure of source code. We use the default hyperparameters except the same settings in our model.
- **CFG+GGNN**. GGNN is one kind of graph neural networks that has been applied for representing programs and detecting specific bugs (e.g., variable misuse) [6, 44]. It encodes the the graph representation of code by simultaneously aggregates all incoming messages of nodes as their hidden states and repeats the process for a fixed iteration number to propagate information. We borrow it by taking CFG as the input for condition-related bug detection and try the iteration number in the range of 2 to 9 to choose the one with the best result.
- **Hybrid Graph+GGNN**. Some work [6] detects variable misuses by representing program source code as graphs. It is based on AST to transform source code into program graphs and learn representations over them with GGNN. Additional edges are added to connect different uses and updates of syntax tokens corresponding to variables for capturing the flow of control and data through a program. We use the default settings as recommended.
- **GINN**. GINN [44] is a recent graph neural network for learning semantic embeddings of program graphs and has been evaluated on detecting variable misuses and null dereferences. It makes abstractions of program graph representations, uses the hierarchy of intervals (i.e. subgraphs) that generally represent looping constructs for scaling the learning to large graphs.
- **Devign**. Devign [60] is a popular vulnerability detection tool that combines CPG and GGNN to embed the semantics of code and then applies a convolutional layer on it. We keep the configurations as default.

Table 2: The effectiveness of different models for condition-related bug detection

Models	Precision	Recall	F1-score
Bugram	11.2	28.0	16.0
DeepBugs	48.5	22.6	30.8
AST+LC-Attention	44.2	26.9	33.4
VulCNN	37.2	24.8	29.7
Devign	50.3	29.2	36.9
Hybrid Graph+GGNN	50.5	28.3	36.3
CFG+GGNN	51.6	28.9	37.1
GINN	53.1	29.9	38.3
CFGNN	**55.5**	**39.6**	**46.2**

- **VulCNN**. VulCNN [47] is one recent vulnerability detection tool based on PDG of source code. It applies centrality analysis to transform PDGs to image formats and uses CNN for detection. We keep the configurations as default.

4.4 Results

We investigate the following research questions to provide a thorough analysis of the experimental results.

RQ1: How does our approach perform compared with baselines for condition-related bug detection?

In this research question, we aim to measure the effectiveness of our approach and how different knowledge can help detect condition-related bugs. Table 2 presents the experimental results on our dataset, where the best ones are shown in bold.

First, we can see that Bugram performs worst among the compared methods. Even though the recall is good to some extent, the precision is only 11.2%, which is much lower than other approaches. This is because N-gram language model largely depends on the co-occurrence of tokens and the context of the given code is restricted to a fixed length. The narrow context limits its capability for distinguishing normal condition expressions from buggy ones.

Deep learning-based approaches improve significantly over Bugram, which largely attributes to the capability of neural networks for capturing code semantics. For example, DeepBugs achieves an F1-score close to 30%, although it also does not encode the structural information. LC-Attention utilize AST to capture the syntax information, including tokens and paths extracted from ASTs respectively. But it performs worse than most graph-based approaches in terms of F1-score, since it can not capture the control flow information that is important for condition-related bugs.

Even though VulCNN takes PDG as input, it does not utilize graph neural networks to encode the structural and dependency information, which results in a lower performance than other graph-based approaches. In addition, Hybrid Graph+GGNN and Devign obtain comparable performances with CFG+GGNN. The reason is that such hybrid graphs take AST as the backbone with some additional edges of control flow and data flow, which include but not highlight the effect of control flow information. This indicates that blending too much information together may not be very beneficial for detecting condition-related bugs. GINN achieves a better performance than GGNN on CFGs, since it takes abstraction and hierarchy of intervals that represent loops to learn from large program graphs and can capture the long-term dependencies better in a way. However, generally GINN is designed for large graphs produced by hierarchical loops but not for the long-term dependencies of control flow paths. Overall, it seems that all the baselines are even not as good as a random classifier. However, our dataset is imbalanced since the same is true of real-world projects, so we cannot think that way. After the experiment, we get the results of Random with 13.9% (P), 49.9 (R) and 21.7% (F1), which is substantially lower than most approaches.

Finally, we can see that our CFGNN achieves the best performance for all evaluation metrics among the compared methods. Because CFGNN can capture global control flow information well by encoding control flow paths and learn the API-usage knowledge.

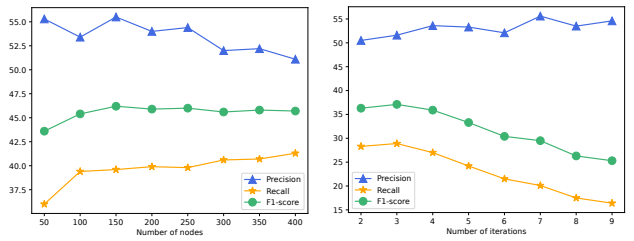
Table 3: Effectiveness of each major component of the proposed approach

Descriptions	P	R	F1
Node Average+Graph Average	52.8	17.6	26.4
Node Embedding+Graph Average	52.3	25.2	34.0
Node Embedding+CFE	54.2	36.1	43.3
Node Embedding+CFE+OD-Attention	55.3	37.9	45.0
CFGNN	**55.5**	**39.6**	**46.2**

RQ2: How effective are the main components of CFGNN?

CFGNN includes three main components, namely node embedding, control flow encoding, and API-usage attention mechanism. To know their effectiveness, we explore different design alternatives from three aspects. First, we directly adopt the average pooling on embedded vectors of tokens at the same node (Node Average). Based on this, we utilize the average pooling again on the node vectors of the graph (Node Average+Graph Average), which serves as a basic reference. Then, we correspondingly replace them with our node embedding (Node Embedding) and control flow encoding (CFE). Based on these two components, we further introduce different knowledge from API invocations (our CFGNN by default) and object declaration statements (Node Embedding+CFE+OD-Attention) through attention mechanism, respectively.

We present the experimental results in Table 3. Compared with the basic model, Node Embedding+Graph Average achieves a much better F1-score, which indicates that node embedding can learn more node semantics by capturing the sequential dependencies of tokens in one node. When adding control flow encoding (Node Embedding+CFE), the performance improves significantly (+9.3% in terms of F1-score) over ignoring the control flow (i.e. Node Embedding+Graph Average). The reason is that our control flow encoding has a strong capability to describe the control flow and node dependencies, and can capture global information of CFG. Attention mechanism introduces some different knowledge from API invocations or object declarations, which is helpful for bug detection on the whole. Consequently, the models with attention component have a better performance than those without it. Introducing API knowledge can extract dependencies between APIs well and obtain deep semantics of the program. Introducing object declarations



(a) Performance of CFGNN with different numbers of nodes (b) Performance of GGNN with different numbers of iterations

Figure 4: Performance of CFGNN and GGNN with different hyper-parameters

knowledge can help to understand the variable semantics in the conditional statements to judge whether there is a bug. In addition, the model with API-Attention is better than that with OD-Attention, which indicates that the API knowledge is more useful than object declarations in our bug detection framework.

In summary, the three main components of our approach (Node Embedding, Control Flow Encoding, and API-Attention) are shown to be effective and can combine different aspects of semantics to achieve better performance.

RQ3: Will the number of nodes affect the capability of our model for capturing global control flow information?

In order to explore that whether too many nodes will affect the extraction of global and long-distance dependencies of CFGs, we mainly conduct two experiments. One is evaluating the performance of our CFGNN in different number of nodes. In the previous experiments, we set the max number of nodes as 150. In this research question, we set different max number of nodes respectively to train the model and evaluate the performance. The number of nodes is from 50 to 400 with the step of 50. The other is evaluating the performance of GGNN with different iterations for the purpose of comparing our model to existing graph neural networks. The iteration number of GGNN is from 2 to 9 with the step of 1.

Figure 4(a) shows the performance of CFGNN with different numbers of nodes. When the number is too small, it will lead to the loss of some overall structural information and thus the performance is lower. As the number grows, the performance increases and finally changes little when it reaches a high coverage of nodes. As shown in Figure 4(a), the recall is gradually ascending with fluctuation and the precision is optimal at 150 nodes, whereas F1-score is relatively stable on the whole. Figure 4(b) shows the performance of GGNN with different iterations. We can see that the performance of GGNN is sensitive to the iteration number and decreases greatly with the increasing of iteration number in general. As explained in Section 3, during the node information propagation, the large iteration number will lead to the repeated computations of the aggregation operations in the intermediate nodes and make the node information diluted. Finally, when the iteration number is small (i.e., 3), GGNN can capture the local node dependencies well and achieve the best F1 value.

Table 4: The performances of different approaches for detecting real condition-related bugs from Defects4J

Approach	#Detected	#Reported	P	R	F1
Findbugs	1	11	9.1	1.9	3.1
Infer	0	0	-	-	-
Bugram	6	140	4.3	11.1	6.2
GINN	9	109	8.3	16.7	11.1
CFGNN	14	122	**11.5**	**25.9**	**15.9**

In summary, our CFGNN has a strong ability to capture long-distance dependencies among nodes and its F1-score is relatively stable across different node numbers. Compared with existing GGNN, our model can better learn the control flow information of CFGs.

RQ4: Can the proposed approach detect real condition-related bugs in unseen projects? Although we have evaluated the proposed approach on our collected dataset, it is hard to guarantee its generalizability for unseen projects. In this RQ, we further evaluate our approach by collecting real-world condition-related bugs from six unseen projects in the widely-used benchmark Defects4J [22]. We manually checked all the bugs from the six projects, and got 3,246 conditions with 54 condition-related bugs as reference (the bugs have been confirmed by previous researchers). To evaluate the usefulness of our proposed approach in reality, we run CFGNN by scanning these projects, record the reported bugs and calculate the metrics based on the confirmed bugs. We compare our approach with four representative methods: 1) Bugram, traditional simple but effective n-gram language model; 2) GINN, the best deep learning method for bug detection in existing work according to Table 2; 3) Findbugs, a traditional static analysis-based tool [20], which can work for the six complete and compilable projects of Defects4J; and 4) a more advanced static tool Infer [3]. Other baselines are excluded due to their relatively worse performance previously, which keep alignment with the results on this benchmark. Hence, we omit the discussion of those approaches. We chose Findbugs because it is a classic static tool that has been used as the baseline [27]. And again, we considered Infer to enhance it as GINN did [44]. For those static tools, we record the reported bugs that matches one conditional statement according to error messages, so as to avoid the interference of other kinds of bugs. The results are listed in Table 4, where #Reported and #Detected mean the reported positives and true positives respectively.

Comparing Table 4 and Table 2, we can see that the detection performance of Bugram, GINN and CFGNN decreases for unseen projects. This is because the unseen projects may have some additional features that do not exist in the training set. Findbugs detected only one condition-related bug, and Infer did not find any one. These static tools perform much worse than learning-based ones because they match predefined code patterns for detecting bugs. However, it is difficult to define rules for the buggy conditions that are highly relevant to code logic. Besides, Infer mainly reports memory-related bugs with a single line, thus the number of warnings is 0. Compared with the other four methods, our CFGNN achieves the best results because it can capture the global control

flow information and the API knowledge in CFGs. In summary, our CFGNN can detect the greatest number (i.e. 14) of condition-related bugs from the 122 warnings.

4.5 Examples

To further investigate our approach, we conduct a qualitative study by analyzing the following two examples shown in Table 5. We present the positions of buggy condition expressions (marked by "//Inspected") and their fixes for ease of illustration.

Case 1. This case (about the *onLoadFinished* function) shows a condition-related bug caused by the miss-checking of the variable "data". To fix it, the developer checks the size of it by invoking an API "getCount()". Since it is comparatively simple, our approach and most of the baselines (except Bugram) successfully detected and reported it as a bug by learning common patterns of its usage. One possible reason is that Bugram detects bugs based on the low joint probability of the short token sequence, while the inspected condition expression is quite common and thus has a high probability.

Case 2. This case (about the *onBindViewHolder* function) shows a more complex situation where multiple factors lead to the condition-related bug. While all the baselines failed to detect it, our model worked well. According to the fix of the buggy code, there can be two main factors. The first is that the buggy condition expression is closely associated with the global context information. For example, the statements that deal with the variables "holder" and "position" have a strong control dependency relationship among them. This causes the changes of multiple related lines. The second is that the use of Android API "getAdapterPosition" for avoiding redundant holders will change the outer variable "mLastPosition" by mistake. Thus the APIs of "mHolders.get()" and "'mHolders.put()" are invoked instead. It is clear that only considering the syntactic knowledge or local dependency information as what the baselines did is not sufficient to identify such a bug. On the contrary, our model captures the global control flow dependency and API knowledge of CFG, and can successfully find the buggy condition.

Table 5: Two cases of condition-related bugs detected by different approaches

Case 1

```
void onLoadFinished(Loader<Cursor> loader, final Cursor data)
{
-      if (data != null){ //Inspected
+      if (data != null && data.getCount() > 0){
          data.moveToFirst();
          final ExchangeRate exchangeRate =
              ExchangeRatesProvider.getExchangeRate(data);
          if (state == State.INPUT)
                  amountCalculatorLink.
                      setExchangeRate(exchangeRate);
       }
}
```

Case 2

```
void onBindViewHolder(ViewHolder holder, int position)
{
    mAdapter.onBindViewHolder(holder, position);
//Inspected
-    if (!isFirstOnly || holder.getAdapterPosition()
-       > mLastPosition) {
+    if (!isFirstOnly || position > mLastPosition ||
+       !Integer.valueOf(position).
+       equals(mHolders.get(holder))) {
          for (Animator anim : getAnimators(holder.itemView)) {
            anim.setDuration(mDuration).start();
            anim.setInterpolator(mInterpolator);
          }
-         mLastPosition = holder.getAdapterPosition();
+         mHolders.put(holder, position);
    } else {
         ViewHelper.clear(holder.itemView);
    }
  }
```

4.6 Discussion

4.6.1 Practicality. As described, our goal is to help developers find condition-related bugs and hopefully provide some clues (i.e., the locations of the buggy conditions) for debugging it. Up to this point, we have only evaluated our approach on seen and unseen projects with known ground-truths. It remains unclear whether our tool can be used in practice for finding new/unknown bugs. Therefore, we perform condition-related bug detection on three popular projects including Apache Log4j [1], Gephi [4] and Dlink [2]. We use our trained model to scan all the condition statements of Java methods in the project files and collect reported warnings. Note that it is not necessary to specify the positions of conditional statements here. As we state in Section 3, we can automatically identify the positions of those conditions in *if* and *while* statements through syntax parsing (e.g., javalang), therefore no extra effort needed for developers at this stage. Then we manually check all the warnings by comparing them with similar samples in our training set or further investigating the related code to confirm the bugs. In total, we found 26 new condition-related bugs that were not reported by the developers before. We had reported the found bugs to the developers, and 7 bugs were confirmed by developers. We helped fix 4 bugs through issues or pull requests, and 3 bugs were fixed by themselves during CI. While the other 19 bugs did not get responses from developers and are pending for further confirmation.

4.6.2 Limitation. Even though we have made good progress on detecting condition-related bugs, our work also has some limitations. First, we only focus on condition-related bugs while there have been many general-purpose bug detectors. The starting point is that existing studies revealed the high frequency of bugs related to conditions, whereas few of them explored the solutions. Therefore, similar to some work (e.g., DeepBugs and GINN) that detects other specific types of bugs, we propose to detect buggy conditions and investigate how to incorporate more domain knowledge for the detection. A follow-up limitation is that overall the performance is not very satisfactory for a fully automated application. In particular, the fraction of false positives is comparatively high (e.g., about 45%). However, it should be noted that our dataset is imbalanced as it is in real-world projects. When taking true negatives into consideration, the number of false positives is not big (< 5%). Furthermore, it is indeed challenging to automatically detect such bugs since they are tied to complex code logic. In practice, one can increase prediction

threshold to further reduce false positives, and we hope our work can inspire more research attention.

5 THREATS TO VALIDITY

Internal Validity. When building datasets, although we followed the previous studies to collect buggy code from the real-world projects, we cannot guarantee that all these bugs are real bugs. Strictly speaking, we should collect those with test cases and run them to confirm all the bugs. However, it is not feasible in this work because we need a large-scale dataset. In the future, we will explore better techniques (e.g., more advanced tools than [53] used in this paper) for improving the quality of data.

External Validity. The code of most baselines are publicly available except for Bugram since the repository has been removed as reported in [27]. To mitigate it, we followed the practice of the prior work and tried our best to replicate Bugram carefully. We will eliminate this threat as soon as the tool is publicly available. Moreover, since none of the baselines are proposed specifically for detecting condition-related bugs, it may seem unfair for us to consider them as baselines. However, as far as we know, there is no existing work that aims to detect condition-related bugs. Fundamentally, those bug detectors especially general-purpose ones are supposed to work for all types of bugs including condition-related bugs. Also, the baselines share the same training set and test set with ours and we have tried best configurations. Altogether, we believe the comparison is fair in our evaluation.

6 RELATED WORK

6.1 Studies on Condition-Related Bugs

There have been many studies on different types of bugs due to the diversity, universality and complexity of the bugs. Among these studies, some involve condition-related bugs, which occur in conditional expressions (e.g., the headers of *if* and *while* statements) and are common in programs. For example, Martinez et al. [34] have shown that *if* conditions are among the most error-prone elements in Java programs. The dataset in Xuan et al. [50] contains 16 bugs with buggy *if* conditions and 6 bugs with missing preconditions from real-world projects. Campos et al. [10] conduct a manual analysis involving 369 real bug fixes from five open-source projects, which shows that 45% of these bugs occur in *if* conditional statements.

At the same time, some methods for repairing condition-based bugs have been proposed. Accurate Condition Synthesis [49] focuses on condition synthesis, and could produce precise patches that have a relatively high probability to be correct. Nopol [50] takes a buggy program and a test suite as the input to generate a patch with a conditional expression, which can effectively fix bugs with buggy *if* conditions and missing preconditions. Staged Program Repair (SPR) [33] combines staged program repair and condition synthesis to generate correct repairs, working productively with a set of parameterized transformation schemas.

6.2 Deep Learning-Based Bug Detection

Traditional methods about bug detection, such as static analysis, rule mining, testing techniques and so on, have been studied for a long time. Recently, deep learning based approaches have been applied to automated bug detection. DeepBugs [37] is proposed to detect name-based bugs, which embeds identifier names from ASTs and converts each code example into a vector for detection. LC-Attention [27] extracts the paths along the nodes of ASTs, and models them by combining the attentional GRU layer and Convolutional layer with a Multi-Head Attention for bug detection. Gated Graph Neural Networks (GGNN) is used to embed the program graphs adapted from ASTs or CFGs for detecting variable misuses [6, 44], which can represent both the syntactic and semantic information of source code. Marko et al. [41] trains the LSTM and pointer networks to localize and repair variable-misuse bugs. Also, deep learning-based vulnerability detection has been proposed [26, 29, 30, 47, 60]. Vuldeepecker [30] extracts code slices and uses BiLSTM to encode them for detecting vulnerabilities. Devign [60] uses graph neural network for graph-level classification through learning on a rich set of code semantic representations to identify code vulnerability. VulCNN [47] converts the source code of a function into an image based on PDG, and trains a CNN model to detect large-scale source code vulnerabilities. Different from the above work, our model focuses on detecting the condition-related bugs by capturing the global control flow dependencies and API knowledge.

7 CONCLUSION

In this paper, we propose a CFG-based Graph Neural Network for detecting the condition-related bugs. The proposed model captures the global control flow information in CFGs and leverages the API knowledge with API-usage attention. We build a large-scale condition-related bug dataset with over 573k buggy and non-buggy conditional expressions for evaluation. The experimental results demonstrate the effectiveness of our approach.

In future work, we plan to improve our CFGNN model by refining the attention mechanism to better incorporate API-usage, exploring how to more efficiently capture control flow information, and applying our model to other types of bugs beyond condition-related ones. We also aim to integrate our model into practical software development tools to aid programmers in bug detection and prevention in real-world scenarios. Our source code and data are publicly available at https://github.com/zhangj111/ConditionBugs.

ACKNOWLEDGMENTS

This work was supported partly by National Key Research and Development Program of China (No.2022YFB4502003), partly by National Natural Science Foundation of China (No. 62072017, 62141209) and Huawei. This research is also partly supported by Australian Research Council Discovery Projects (DP200102940, DP220103044), and partly by the National Research Foundation, Singapore and the Cyber Security Agency under its National Cybersecurity R&D Programme (NCRP25-P04-TAICeN), the National Satellite of Excellence in Trustworthy Software Systems (NSOE-TSS) project under the National Cybersecurity R&D (NCR) Grant award no. NRF2018NCR-NSOE003-0001, and the NRF Investigatorship NRF-NRFI06-2020-0001. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not reflect the views of National Research Foundation, Singapore and Cyber Security Agency of Singapore.

REFERENCES

[1] 2022. Apache Log4j. https://github.com/apache/logging-log4j2.
[2] 2022. Dlink. https://github.com/DataLinkDC/dlink.
[3] 2022. Facebook Infer. https://fbinfer.com/.
[4] 2022. Gephi. https://github.com/gephi/gephi.
[5] 2022. javalang. https://github.com/c2nes/javalang.
[6] Miltiadis Allamanis, Marc Brockschmidt, and Mahmoud Khademi. 2018. Learning to Represent Programs with Graphs. In *International Conference on Learning Representations*.
[7] Dzmitry Bahdanau, Kyung Hyun Cho, and Yoshua Bengio. 2015. Neural machine translation by jointly learning to align and translate. In *3rd International Conference on Learning Representations, ICLR 2015*.
[8] Pan Bian, Bin Liang, Wenchang Shi, Jianjun Huang, and Yan Cai. 2018. NAR-miner: discovering negative association rules from code for bug detection. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 411–422. https://doi.org/10.1145/3236024.3236032
[9] Nghi DQ Bui, Yijun Yu, and Lingxiao Jiang. 2019. Bilateral dependency neural networks for cross-language algorithm classification. In *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 422–433. https://doi.org/10.1109/saner.2019.8667995
[10] Eduardo Cunha Campos and Marcelo de Almeida Maia. 2017. Common bug-fix patterns: A large-scale observational study. In *2017 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*. IEEE, 404–413. https://doi.org/10.1109/esem.2017.55
[11] Brian Cole, Daniel Hakim, David Hovemeyer, Reuven Lazarus, William Pugh, and Kristin Stephens. 2006. Improving your software using static analysis to find bugs. In *Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*. 673–674. https://doi.org/10.1145/1176617.1176667
[12] Dawson Engler, David Yu Chen, Seth Hallem, Andy Chou, and Benjamin Chelf. 2001. Bugs as deviant behavior: A general approach to inferring errors in systems code. *ACM SIGOPS Operating Systems Review* 35, 5 (2001), 57–72. https://doi.org/10.21236/ada419584
[13] Vikas Garg, Stefanie Jegelka, and Tommi Jaakkola. 2020. Generalization and representational limits of graph neural networks. In *International Conference on Machine Learning*. PMLR, 3419–3430.
[14] James Gosling, Bill Joy, Guy L. Steele, Gilad Bracha, and Alex Buckley. 2014. *The Java Language Specification, Java SE 8 Edition* (1st ed.). Addison-Wesley Professional.
[15] Xiaodong Gu, Hongyu Zhang, Dongmei Zhang, and Sunghun Kim. 2016. Deep API learning. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. 631–642. https://doi.org/10.1145/2950290.2950334
[16] Sudheendra Hangal and Monica S Lam. 2002. Tracking down software bugs using automatic anomaly detection. In *Proceedings of the 24th International Conference on Software Engineering. ICSE 2002.* IEEE, 291–301. https://doi.org/10.1145/581376.581377
[17] Abram Hindle, Earl T Barr, Mark Gabel, Zhendong Su, and Premkumar Devanbu. 2016. On the naturalness of software. *Commun. ACM* 59, 5 (2016), 122–131. https://doi.org/10.1109/icse.2012.6227135
[18] Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long short-term memory. *Neural computation* 9, 8 (1997), 1735–1780.
[19] David Hovemeyer and William Pugh. 2004. Finding bugs is easy. *Acm sigplan notices* 39, 12 (2004), 92–106. https://doi.org/10.1145/1052883.1052895
[20] David Hovemeyer and William Pugh. 2007. Finding more null pointer bugs, but not too many. In *Proceedings of the 7th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*. 9–14. https://doi.org/10.1145/1251535.1251537
[21] Guoliang Jin, Linhai Song, Xiaoming Shi, Joel Scherpelz, and Shan Lu. 2012. Understanding and detecting real-world performance bugs. *ACM SIGPLAN Notices* 47, 6 (2012), 77–88. https://doi.org/10.1145/2345156.2254075
[22] René Just, Darioush Jalali, and Michael D Ernst. 2014. Defects4J: A database of existing faults to enable controlled testing studies for Java programs. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*. 437–440. https://doi.org/10.1145/2610384.2628055
[23] Diederik P Kingma and Jimmy Ba. 2014. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980* (2014).
[24] Taek Lee, Jaechang Nam, DongGyun Han, Sunghun Kim, and Hoh Peter In. 2011. Micro interaction metrics for defect prediction. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*. 311–321. https://doi.org/10.1145/2025113.2025156
[25] Yujia Li, Daniel Tarlow, Marc Brockschmidt, and Richard Zemel. 2015. Gated graph sequence neural networks. *arXiv preprint arXiv:1511.05493* (2015).
[26] Yi Li, Shaohua Wang, and Tien N Nguyen. 2021. Vulnerability detection with fine-grained interpretations. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 292–303. https://doi.org/10.1145/3468264.3468597
[27] Yi Li, Shaohua Wang, Tien N Nguyen, and Son Van Nguyen. 2019. Improving bug detection via context-based code representation learning and attention-based neural networks. *Proceedings of the ACM on Programming Languages* 3, OOPSLA (2019), 1–30. https://doi.org/10.1145/3360588
[28] Zhenmin Li and Yuanyuan Zhou. 2005. PR-Miner: automatically extracting implicit programming rules and detecting violations in large software code. *ACM SIGSOFT Software Engineering Notes* 30, 5 (2005), 306–315. https://doi.org/10.1145/1095430.1081755
[29] Zhen Li, Deqing Zou, Shouhuai Xu, Hai Jin, Yawei Zhu, and Zhaoxuan Chen. 2021. SySeVR: A Framework for Using Deep Learning to Detect Software Vulnerabilities. *IEEE Transactions on Dependable and Secure Computing* (2021), 1–1. https://doi.org/10.1109/TDSC.2021.3051525
[30] Zhen Li, Deqing Zou, Shouhuai Xu, Xinyu Ou, Hai Jin, Sujuan Wang, Zhijun Deng, and Yuyi Zhong. 2018. VulDeePecker: A Deep Learning-Based System for Vulnerability Detection. In *25th Annual Network and Distributed System Security Symposium, NDSS 2018, San Diego, California, USA, February 18-21, 2018*. The Internet Society. https://doi.org/10.14722/ndss.2018.23158
[31] Benjamin Livshits and Thomas Zimmermann. 2005. Dynamine: finding common error patterns by mining software revision histories. *ACM SIGSOFT Software Engineering Notes* 30, 5 (2005), 296–305. https://doi.org/10.1145/1081706.1081754
[32] V Benjamin Livshits and Monica S Lam. 2003. Tracking pointers with path and context sensitivity for bug detection in C programs. In *Proceedings of the 9th European software engineering conference held jointly with 11th ACM SIGSOFT international symposium on Foundations of software engineering*. 317–326. https://doi.org/10.1145/949952.940114
[33] Fan Long and Martin Rinard. 2015. Staged program repair with condition synthesis. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. 166–178. https://doi.org/10.1145/2786805.2786811
[34] Matias Martinez and Martin Monperrus. 2015. Mining software repair models for reasoning on the search space of automated program fixing. *Empirical Software Engineering* 20, 1 (2015), 176–205. https://doi.org/10.1007/s10664-013-9282-8
[35] Raimund Moser, Witold Pedrycz, and Giancarlo Succi. 2008. A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction. In *Proceedings of the 30th international conference on Software engineering*. 181–190. https://doi.org/10.1145/1368088.1368114
[36] Renaud Pawlak, Martin Monperrus, Nicolas Petitprez, Carlos Noguera, and Lionel Seinturier. 2015. Spoon: A Library for Implementing Analyses and Transformations of Java Source Code. *Software: Practice and Experience* 46 (2015), 1155–1179. https://doi.org/10.1002/spe.2346
[37] Michael Pradel and Koushik Sen. 2018. Deepbugs: A learning approach to name-based bug detection. *Proceedings of the ACM on Programming Languages* 2, OOPSLA (2018), 1–25. https://doi.org/10.1145/3276517
[38] Baishakhi Ray, Vincent Hellendoorn, Saheel Godhane, Zhaopeng Tu, Alberto Bacchelli, and Premkumar Devanbu. 2016. On the" naturalness" of buggy code. In *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*. IEEE, 428–439. https://doi.org/10.1145/2884781.2884848
[39] Mike Schuster and Kuldip K Paliwal. 1997. Bidirectional recurrent neural networks. *IEEE transactions on Signal Processing* 45, 11 (1997), 2673–2681. https://doi.org/10.1109/78.650093
[40] Michele Tufano, Cody Watson, Gabriele Bavota, Massimiliano Di Penta, Martin White, and Denys Poshyvanyk. 2018. Deep learning similarities from different representations of source code. In *2018 IEEE/ACM 15th International Conference on Mining Software Repositories (MSR)*. IEEE, 542–553. https://doi.org/10.1145/3196398.3196431
[41] Marko Vasic, Aditya Kanade, Petros Maniatis, David Bieber, and Rishabh Singh. 2018. Neural Program Repair by Jointly Learning to Localize and Repair. In *International Conference on Learning Representations*.
[42] Song Wang, Devin Chollak, Dana Movshovitz-Attias, and Lin Tan. 2016. Bugram: bug detection with n-gram language models. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*. 708–719. https://doi.org/10.1145/2970276.2970341
[43] Song Wang, Taiyue Liu, and Lin Tan. 2016. Automatically learning semantic features for defect prediction. In *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*. IEEE, 297–308. https://doi.org/10.1145/2884781.2884804
[44] Yu Wang, Ke Wang, Fengjuan Gao, and Linzhang Wang. 2020. Learning semantic program embeddings with graph interval neural network. *Proceedings of the ACM on Programming Languages* 4, OOPSLA (2020), 1–27. https://doi.org/10.1145/3428205
[45] Andrzej Wasylkowski and Andreas Zeller. 2011. Mining temporal specifications from object usage. *Automated Software Engineering* 18, 3 (2011), 263–292. https://doi.org/10.1109/ase.2009.30
[46] Andrzej Wasylkowski, Andreas Zeller, and Christian Lindig. 2007. Detecting object usage anomalies. In *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*. 35–44. https://doi.org/10.1145/1287624.1287632
[47] Yueming Wu, Deqing Zou, Shihan Dou, Wei Yang, Duo Xu, and Hai Jin. 2022. VulCNN: An Image-inspired Scalable Vulnerability Detection System. In *2022*

IEEE/ACM 44th International Conference on Software Engineering (ICSE). 2365–2376. https://doi.org/10.1145/3510003.3510229

[48] Yichen Xie and Alex Aiken. 2007. Saturn: A scalable framework for error detection using boolean satisfiability. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 29, 3 (2007), 16–es. https://doi.org/10.1145/1040305.1040334

[49] Yingfei Xiong, Jie Wang, Runfa Yan, Jiachen Zhang, Shi Han, Gang Huang, and Lu Zhang. 2017. Precise condition synthesis for program repair. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. IEEE, 416–426. https://doi.org/10.1109/icse.2017.45

[50] Jifeng Xuan, Matias Martinez, Favio Demarco, Maxime Clement, Sebastian Lamelas Marcote, Thomas Durieux, Daniel Le Berre, and Martin Monperrus. 2016. Nopol: Automatic repair of conditional statement bugs in java programs. *IEEE Transactions on Software Engineering* 43, 1 (2016), 34–55. https://doi.org/10.1109/tse.2016.2560811

[51] Fabian Yamaguchi, Nico Golde, Daniel Arp, and Konrad Rieck. 2014. Modeling and discovering vulnerabilities with code property graphs. In *2014 IEEE Symposium on Security and Privacy*. IEEE, 590–604. https://doi.org/10.1109/sp.2014.44

[52] Jiaxuan You, Rex Ying, and Jure Leskovec. 2019. Position-aware graph neural networks. In *International Conference on Machine Learning*. PMLR, 7134–7143.

[53] Sarim Zafar, Muhammad Zubair Malik, and Gursimran Singh Walia. 2019. Towards standardizing and improving classification of bug-fix commits. In *2019 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*. IEEE, 1–6. https://doi.org/10.1109/esem.2019.8870174

[54] Jian Zhang, Xu Wang, Hongyu Zhang, Hailong Sun, and Xudong Liu. 2020. Retrieval-based neural source code summarization. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. 1385–1397. https://doi.org/10.1145/3377811.3380383

[55] Jian Zhang, Xu Wang, Hongyu Zhang, Hailong Sun, Yanjun Pu, and Xudong Liu. 2020. Learning to handle exceptions. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*. 29–41. https://doi.org/10.1145/3324884.3416568

[56] Jian Zhang, Xu Wang, Hongyu Zhang, Hailong Sun, Kaixuan Wang, and Xudong Liu. 2019. A novel neural source code representation based on abstract syntax tree. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 783–794. https://doi.org/10.1109/icse.2019.00086

[57] Gang Zhao and Jeff Huang. 2018. DeepSim: Deep Learning Code Functional Similarity. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Lake Buena Vista, FL, USA) *(ESEC/FSE 2018)*. ACM, New York, NY, USA, 141–151. https://doi.org/10.1145/3236024.3236068

[58] Hao Zhong and Zhendong Su. 2015. An empirical study on real bug fixes. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, Vol. 1. IEEE, 913–923. https://doi.org/10.1109/icse.2015.101

[59] Hao Zhong, Suresh Thummalapenta, Tao Xie, Lu Zhang, and Qing Wang. 2010. Mining API mapping for language migration. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1*. 195–204. https://doi.org/10.1145/1806799.1806831

[60] Yaqin Zhou, Shangqing Liu, Jingkai Siow, Xiaoning Du, and Yang Liu. 2019. Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks. *Advances in neural information processing systems* 32 (2019).

Received 2023-02-16; accepted 2023-05-03