Context-Aware Neural Fault Localization

Zhuo Zhang, Yan Lei, *Member, IEEE*, Xiaoguang Mao, Meng Yan, *Member, IEEE*,
Xin Xia, *Member, IEEE*, and David Lo, *Fellow, IEEE*

***Abstract*—Numerous fault localization techniques identify suspicious statements potentially responsible for program failures by discovering the statistical correlation between test results (i.e., *failing* or *passing*) and the executions of the different statements of a program (i.e., *covered* or *not covered*). They rarely incorporate a failure context into their suspiciousness evaluation despite the fact that a failure context showing how a failure is produced is useful for analyzing and locating faults. Since a failure context usually contains the transitive relationships among the statements of causing a failure, its relationship complexity becomes one major obstacle for the context incorporation in suspiciousness evaluation of fault localization. To overcome the obstacle, our insight is that leveraging the promising learning ability may be a candidate solution to learn a feasible model for incorporating a failure context into fault localization. Thus, we propose a context-aware neural fault localization approach (CAN). Specifically, CAN represents the failure context by constructing a program dependency graph, which shows how a set of statements interact with each other (i.e., data and control dependencies) to cause a failure. Then, CAN utilizes graph neural networks to analyze and incorporate the context (e.g., the dependencies among the statements) into suspiciousness evaluation. Our empirical results on the 12 large-sized programs show that CAN achieves promising results (e.g., 29.23% faults are ranked within top 5), and it significantly improves the state-of-the-art baselines with a substantial margin.**

***Index Terms*—Fault localization, graph neural networks, program dependecy graphs, suspiciousness.**

I. Introduction

FOR reducing the debugging cost, researchers have proposed various fault localization techniques to provide automated assistance in finding faults that cause failures (e.g., [1], [2], [3], [4], [5], [6], [7], [8], [9]). These techniques seek to develop effective suspiciousness evaluation models to evaluate the suspiciousness of a statement (or other program entities) of being faulty. Among them, Spectrum-based Fault Localization (SFL) [1], [10], [11], [12] and Deep-Learning-based Fault Localization (DLFL) [13], [14], [15] are two of the most popular techniques.

Fig. 1 shows the process of SFL and DLFL respectively. Given a programs P and a test suite T, the faulty program P contains a fault and runs against T. Then, SFL and DLFL both collect and abstract the runtime information of the faulty program P on its T as a matrix [1], [13], representing a statement[1] *covered* or *not covered* with a *passing* or *failing* result, and the matrix will be inputted into the next suspiciousness evaluation model. Next, SFL uses correlation coefficients to construct a SFL formula fx, evaluating the suspiciousness of each statement of being faulty in the program P; DLFL uses neural networks with the matrix as well as some other program information of P as the training dataset to learn a trained model for evaluating the suspiciousness of each statement of being faulty in the program P. Finally, they both output a ranking list of all statements of each faulty program in descending order of suspiciousness.

Despite the fact that they have delivered promising localization results [12], [13], [14], [16], they still have some limitations. Their basic idea is that a program entity executed by more failing test cases and less passing test cases should have a higher suspiciousness value of being faulty. Following this idea, they use neural networks or correlation coefficients to discover the statistical correlation between test results (i.e., *failing* or *passing*) and the various coverage types (e.g., statements, branches, du-pairs) [17] of a program (i.e., *covered* or *not covered*). The coverage information generally cannot be regarded as a failure context since it just shows whether a statement (i.e., a node) is covered or not, without the relationships (i.e., edges) among the nodes showing how a fault propagates to cause a failure. Thus, these approaches do not consider the inherent relationships (i.e., edges) presented by a failure context. For example, these approaches are not based on graph-based architectures and thus it is difficult for them to learn the relationships (i.e., edges) presented by a failure context.

In fact, a graph structure constructed by a failure context shows the process of how a failure is produced (e.g., the statement dependencies not just coverage), and is useful for analyzing and locating faults [18]. Therefore, just relying on the correlation

Manuscript received 30 July 2021; revised 31 March 2023; accepted 17 May 2023. Date of publication 23 May 2023; date of current version 18 July 2023. This work was supported in part by the National Natural Science Foundation of China under Grant 62272072, in part by the Fundamental Research Funds for the Central Universities under Grant 2022CDJDX-005, in part by the Major Key Project of PCL under Grant PCL2021A06, in part by the Chongqing Technology Innovation and Application Development Project under Grants CSTB2022TIAD-STX0007 and CSTB2022TIAD-KPX0067, and in part by the National Defense Basic Scientific Research Project under Grant WDZC20205500308. Recommended for acceptance by A. Orso. *(Corresponding author: Yan Lei.)*

Zhuo Zhang and Meng Yan are with the School of Big Data and Software Engineering, Chongqing University, Chongqing 400044, China (e-mail: zz8477@126.com; mengy@cqu.edu.cn).

Yan Lei is with the School of Big Data and Software Engineering, Chongqing University, Chongqing 400044, China, and also with the Peng Cheng Laboratory, Shenzhen, Guangdong Province 518066, China (e-mail: yanlei@cqu.edu.cn).

Xiaoguang Mao is with the College of Computer, National University of Defense Technology, Changsha, Hunan 410073, China (e-mail: xgmao@nudt.edu.cn).

Xin Xia is with the College of Computer Science and Technology, Zhejiang University, Hangzhou, Zhejiang 310027, China (e-mail: xin.xia@acm.org).

David Lo is with the School of Information Systems, Singapore Management University, Singapore 188065 (e-mail: davidlo@smu.edu.sg).

Digital Object Identifier 10.1109/TSE.2023.3279125

[1]Statements are the widely used program entity type. Other types can be branches, du-pairs, etc.

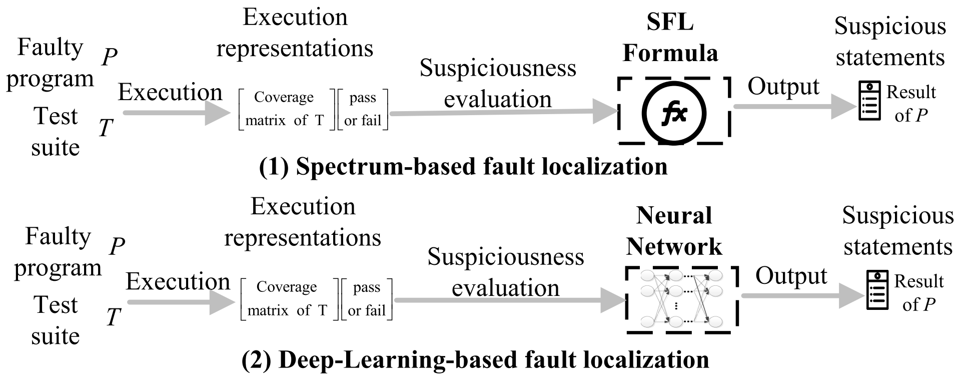


Fig. 1. The process of fault localization.

between test results and the coverage of each statement without considering the inherent relationships presented by a failure context can affect the accuracy of fault localization. Nevertheless, a failure context usually contains many program entities (e.g., statements) and their transitive relationships. Due to the transitive relationships among the entities, it is still difficult to analyze and combine the failure context even if much research (e.g., [18], [19], [20], [21], [22], [23]) has been conducted on reducing the complexity of the analysis. Thus, the high complexity also becomes one major challenge for incorporating the failure context into suspiciousness evaluation for fault localization.

To address the aforementioned challenges, we aim to model and learn a graph structure constructed from the failure context for fault localization. Recently, graph neural networks (GNNs) [24], [25] have been proposed to collectively aggregate information from graph structure, modeling input and/or output consisting of elements and their dependency. GNNs have been successfully employed to model graph-structured dependencies for various applications such as social science [26], [27], natural language processing [28], computer vision [29], vulnerability detection [30], [31], [32], [33], and many other research areas [34], [35]. The learning process of GNNs is to update the parameters and hidden state of nodes by capturing the dependencies of a graph via the message passing on edges, which perfectly simulates the propagation process of faults in the program. In this paper, we present a failure context (i.e., a program slice [18], [36]) as a graph (i.e., a program dependency graph [18]) including nodes (i.e., statements) and edges (i.e., dependencies). And GNNs can be a candidate solution of learning the transitive relationships of a failure context (i.e., a graph) for incorporating the failure context into suspiciousness evaluation of fault localization.

Therefore, we propose CAN: **C**ontext-**A**ware **N**eural fault localization, which uses program dependency graphs to model a failure context, and leverages graph neural networks to analyze and combine a failure context for suspiciousness evaluation of fault localization. Following the training and testing process of DLFL in Fig. 1, CAN also trains the model with all available samples of a faulty program and tests the trained model with a synthesized testing dataset. Specifically, in the training phase, CAN uses program slicing [18], [36] to construct a failure context denoted as a directed graph (i.e., a program dependency graph widely used in software testing and analysis) including the nodes (i.e., failure-inducing statements) and the edges (i.e., dependencies among these statements). Based on the graph, CAN utilizes a GNN to capture dependencies among statements and generate accurate statement embedding vectors correspondingly, which are difficult to be revealed by the state-of-the-art fault localization techniques (e.g., SFL and DLFL). Based on accurate statement embedding vectors, CAN obtains more reliable representations of the statements, and then trains the network with test cases for each bug. In the testing phase, CAN evaluates the suspiciousness of each statement of being faulty by testing the trained model using the one-hot statement coverage vectors, where each vector represents covering only one statement.

To evaluate our approach, we design and conduct a large-scale empirical study on 12 real-world programs (e.g., the programs of Defects4J, python, and gzip). We compare CAN with a total of 12 state-of-the-art fault localization baselines, including Dstar [12], MULTRIC [37], CNN-FL [13], and DeepRL4FL [16].

The results show that CAN improves the state-of-the-art baselines statistically significant and substantially, e.g., the average improvement for the most important metric [38], such as Top-5, as compared to the best-performing baseline is 14.31%.

The main contributions of this paper can be summarized as:

- We propose a context-aware fault localization approach (CAN) by modeling a failure context as a graph and learning the complex relationship (i.e., statement dependencies) among the nodes (i.e., statements) from the graph level for effective fault localization.
- We propose a GNN-based bug-specific learning scheme to better evaluate statement suspiciousness for each bug individually. To the best of our knowledge, it is the first time that graph neural networks are used in analyzing and combining a failure context for fault localization, demonstrating their promising context-aware potential of improving fault localization.
- We conduct a large-scale empirical study on 12 large real-life programs and 12 baselines, showing that our approach is effective to improve fault localization.
- We open source the replication package online[2] with an archival snapshot,[3] including the source code, datasets,

[2]https://github.com/oy-sarah/CAN
[3]https://zenodo.org/badge/latestdoi/291069322

docker implementation and running examples, for follow-up works.

The structure of the rest paper is organized as follows. Section II introduces the related work. Section III depicts our approach CAN. Section IV presents our large-scale empirical study. Section VII concludes the whole study and mentions future works.

II. Related Work

A. Graph Neural Networks

Recently, graph neural networks (GNNs) have emerged as an effective class of models for solving non-euclidean data structure problems, they could learn representations of nodes on graphs and collectively aggregate information from graph structure [24], [25]. Due to their convincing performance and high interpretability, GNNs have been utilized to be the standard toolkit for analyzing and learning from data on graphs and have been successfully applied to various applications [26], [27], [28], [29], [34], [35], [39], such as social sciences, computer vision, knowledge graphs, natural language processing, etc. GNNs are also applied for vulnerability detection (e.g., [30], [31], [32], [33]) whereas our work focuses on a different research topic on software fault localization.

In a graph, each node is naturally defined by its features and the related nodes. Thus, the target of GNN is to learn the state embeddings which contain the information of neighborhoods. The state embeddings are vectors of nodes and could be used to produce outputs such as the labels of the nodes. There are several advantages of GNNs compared with traditional neural networks such as convolutional neural networks, recurrent neural networks and multi-layer perceptron.

First, GNNs could handle the graph input properly via propagating on each node and ignoring the input order of nodes. It means that the output of GNNs is invariant for the input order of nodes. In contrast, traditional neural networks process the graph input by stacking the feature of nodes in a certain order, although the graph does not exist a natural order of nodes. Furthermore, if a traditional neural network is used to present a graph completely, all input sequences need to be traversed, incurring much computational overhead. Although programs are written in a certain order, they are not always executed line-by-line in a linear fashion. Instead, the features like function call, loops and conditionals cause the program jump around from line to line in a nonlinear and arbitrary fashion, in which fashion GNNs can work whereas traditional neural networks cannot.

Second, unlike regarding an edge as the feature of nodes in traditional neural networks, GNNs update the hidden state of nodes by capturing the dependencies of a graph via the message passing on edges. Thus, the propagation of GNNs is based on the graph structure instead of using it as part of features, i.e., GNNs can retain a state that represents the information from its neighborhood with arbitrary depth. Take a program dependency graph, widely used for representing a program, as an example. In a graph, the edges are the dependencies like data dependencies or control dependencies between two statements, and the massage passing in the graph simulates the propagation of errors among the statements in the program. In this context, GNNs can learn the dependencies while traditional neural networks cannot.

Third, just like the operating mechanism of the human brain that is almost based on a graph extracted from daily experience, GNN can analyze graph and capture salient features from a graph while traditional neural networks cannot do so well (without some loss of information).

Based on the above analysis, GNNs demonstrate the potential of analyzing and learning the transitive relationships of the failure context as a graph for fault localization.

$$
\begin{array}{cc}
\textit{N statements} & \textit{errors} \\
\begin{bmatrix} x_{11} & x_{12} & \cdots & x_{1N} \\ x_{21} & x_{22} & \cdots & x_{2N} \\ \vdots & \vdots & \ddots & \vdots \\ x_{M1} & x_{M2} & \cdots & x_{MN} \end{bmatrix} & \begin{bmatrix} e_1 \\ e_2 \\ \vdots \\ e_M \end{bmatrix}
\end{array}
$$

Fig. 2. The coverage and results of M executions.

B. Spectrum-Based Fault Localization

Spectrum-based Fault Localization (SFL) [1], [10], [13], [40], [41], [42] seeks to use correlation coefficients to discover the statistical correlation between test results (i.e., failing or passing) and the coverage of the different statements of a program (i.e., covered or not covered).

Given a program P with N statements, it is executed by a test suite T with M test cases, which contain at least one failing test case (see Fig. 2). SFL first defines a matrix as the input to represent the runtime information of a test suite including the coverage information of statements and the test results. Fig. 2 shows the definition of the matrix. The element $x_{ij} = 1$ means that the statement j is covered by the test case i, and $x_{ij} = 0$ otherwise. The $M \times N$ matrix records the coverage information of each statement by the test suite T. The error vector e represents the test results. The element e_i equals to 1 if the test case i failed, and 0 otherwise. The error vector shows the test results of each test case (i.e., failing or passing).

$$a_{np}(s_j) = \sum_{i \in np(s_j)} (1 - x_{ij}), \quad a_{ep}(s_j) = \sum_{i \in ep(s_j)} x_{ij}$$

$$a_{nf}(s_j) = \sum_{i \in nf(s_j)} (1 - x_{ij}), \quad a_{ef}(s_j) = \sum_{i \in ef(s_j)} x_{ij}$$

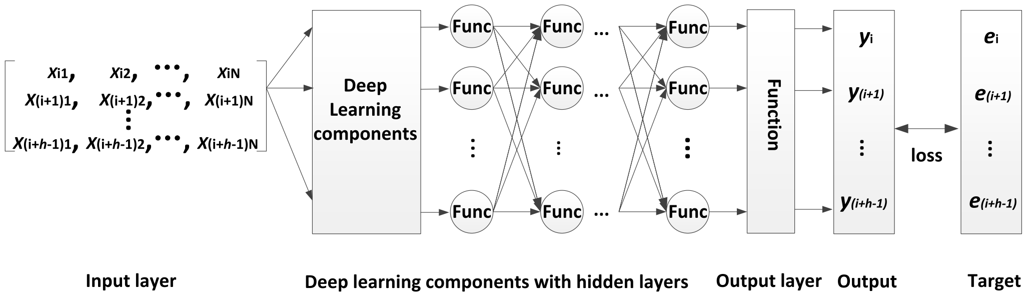
where,

$$
\begin{aligned}
np(s_j) &= \{i | (x_{ij} = 0) \wedge (e_i = 0)\}, \quad ep(s_j) \\
&= \{i | (x_{ij} > 0) \wedge (e_i = 0)\} \\
nf(s_j) &= \{i | (x_{ij} = 0) \wedge (e_i = 1)\}, \quad ef(s_j) \\
&= \{i | (x_{ij} > 0) \wedge (e_i = 1)\} \qquad (1)
\end{aligned}
$$

Based on the matrix in Fig. 2, SFL defines four variables in 1, where a_{np}, a_{nf}, a_{ep}, and a_{ef} for the statement j (i.e., s_j) denote the number of passing/failing test cases in which the statement was/wasn't executed.

TABLE I
SFL FORMULAS

Name		Formulas	Name	Formulas
ER1	Naish1	$\begin{cases} -1 & if\ a_{ne} > 0 \\ a_{np} & if\ a_{ne} \leq 0 \end{cases}$	GP02	$2\left(a_{ef} + \sqrt{a_{np}}\right) + \sqrt{a_{ep}}$
	Optimal_P	$a_{ef} - \frac{a_{ep}}{a_{ep}+a_{np}+1}$	GP03	$\sqrt{\left\|a_{ef}^2 - \sqrt{a_{ep}}\right\|}$
	GP13	$a_{ef}\left(1 + \frac{a_{ep}}{2a_{ep}+a_{ef}}\right)$	GP19	$a_{ef}\sqrt{\left\|a_{ep} - a_{ef} + a_{nf} - a_{np}\right\|}$
ER5	Wong1	a_{ef}	Dstar	$\frac{a_{ef}^{*}}{a_{nf}+a_{ep}}$
	Russel_Rao	$\frac{a_{ef}}{a_{ef}+a_{nf}+a_{ep}+a_{np}}$		
	Binary	$\begin{cases} 0, & if\ a_{ne} > 0 \\ 1, & if\ a_{ne} \leq 0 \end{cases}$	Ochiai	$\frac{a_{ef}}{\sqrt{(a_{ef}+a_{nf})(a_{ef}+a_{ep})}}$



$x_{i1}, x_{i2}, \cdots, x_{iN}$
$x_{(i+1)1}, x_{(i+1)2}, \cdots, x_{(i+1)N}$
$x_{(i+h-1)1}, x_{(i+h-1)2}, \cdots, x_{(i+h-1)N}$
Deep Learning components
Func
Function
y_i
$y_{(i+1)}$
$y_{(i+h-1)}$
loss
e_i
$e_{(i+1)}$
$e_{(i+h-1)}$
Input layer
Deep learning components with hidden layers
Output layer
Output
Target

Fig. 3. Suspiciousness evaluation using neural networks.

With the four variables for each statement, SFL defines many suspiciousness evaluation formulas using correlation coefficients to evaluate the suspiciousness of each statement being faulty. Researchers have conducted both theoretical [43], [44] and empirical analysis [12] on finding the optimal SFL formulas, i.e., ER1, ER5, GP02, GP03, GP19, Ochiai and Dstar (D*). Table I shows all the 7 optimal SFL formulas.[4] Some researchers further combine the multiple SFL formulas to improve fault localization, e.g., MULTRIC [37].

C. Deep-Learning-Based Fault Localization

Deep-Learning-based Fault Localization (DLFL) [13], [14], [45], [46] tries to utilize artificial neural network with hidden layers [15], [46], [47], [48], [49], [50], [51] to learn a localization model reflecting the statistical correlation between test results (i.e., failing or passing) and the coverage of the different statements of a program (i.e., covered or not covered). Among these DLFL techniques, MLP-FL [45], CNN-FL [13], BiLSTM-FL [14] and DeepRL4FL [16] are the representative and effective ones.

Fig. 3 shows the architecture of suspiciousness evaluation of DLFL: one input layer, deep learning components with several hidden connected layers, and one output layer. In the input layer, DLFL uses the matrix and the error vector of Fig. 2 as the training samples and their corresponding labels, respectively. In other words, h rows of the matrix $M \times N$ and its corresponding error vector are used as an input, which are the coverage information of h test cases and their corresponding test results starting from the i-th row, where $i \in \{1, 1+h, 1+2h, ..., 1+(\lfloor M/h \rfloor+1)\times h\}$. In deep learning components with several hidden connected layer, MLP-FL, CNN-FL/DeepRL4FL and BiLSTM-FL use multi-layer perceptron, convolutional neural network and bi-directional long short-term memory respectively. In the output layer, DLFL uses *Sigmoid* function [46] because values sent into a *Sigmoid* function will be 0 to 1. Each element in the result vector of the Sigmoid function has difference with the corresponding element of the target vector. Back propagation algorithm is used to update the parameters of the model, and the goal is to minimize the difference between training result y and error vector e. The network is trained iteratively.

Finally, DLFL learns a trained model reflecting the relationship between statement coverage and test results, and constructs a synthesized testing dataset (i.e., one-hot vectors [13], [15]) to test the trained model for evaluating the suspiciousness of each statement.

Besides the above suspiciousness evaluation, there are other recent related work, e.g., Learning-to-Rank technique [52], [53], composition technique [54], [55], unified debugging technique [56]. Our work focuses on deep learning technique not Learning-to-Rank technique, and the results [46] have shown deep learning technique has higher effectiveness than Learning-to-Rank technique [52], [53]. Composition technique [54], [55] studies the combination of different individual fault localization approaches whereas our work proposes an effective individual fault localization approach. Unified debugging technique [56]

[4]The * in D* formula is usually assigned to 2.

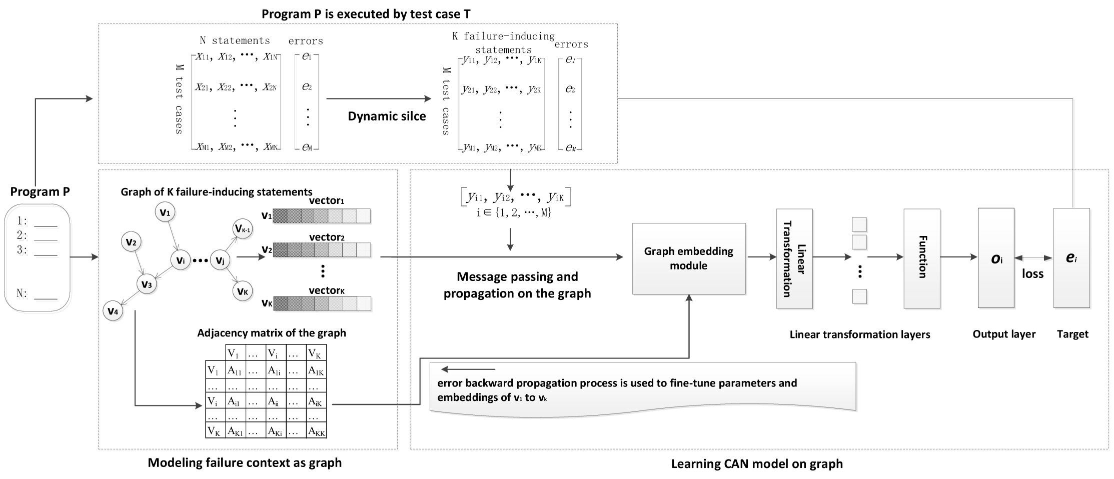

Program P is executed by test case T
N statements
errors
M test cases
Dynamic silce
K failure-inducing statements
Program P
Graph of K failure-inducing statements
vector₁
vector₂
vectorₖ
Adjacency matrix of the graph
Message passing and propagation on the graph
Graph embedding module
Linear Transformation
Function
loss
Linear transformation layers
Output layer
Target
error backward propagation process is used to fine-tune parameters and embeddings of v₁ to vₖ
Modeling failure context as graph
Learning CAN model on graph

Fig. 4. The architecture of CAN.

focuses on how to improve fault localization using the repair information whereas our work concentrates on how to improve fault localization without using the repair information.

Prior studies proposed mining of bug signatures [21], [22], [23], [57], [58]) (i.e., bug contexts) after identification of suspicious program elements for bug understanding. Different from such studies, our work improves the suspiciousness evaluation models with bug contexts. Our work can potentially be used in tandem with these existing works; we can run bug signature mining work after suspicious program elements are identified by our proposed approach.

III. Approach

This section will introduce CAN which applies graph neural networks to analyzing and combining failure contexts for fault localization. Fig. 4 shows the architecture of CAN.

CAN is different from traditional machine learning training and testing scheme. CAN tries to learn the mappings between statements and failures from all available test cases, and evaluate the suspiciousness of each statement individually. For training dataset, we use all available test cases as the training dataset to learn a model for the mappings because their execution information corresponds to a specific bug. For testing dataset, we employ one-hot coverage vectors to represent each of involved statements individually and then consider them as the testing dataset where a size equals to the number of statements of a failure context. The testing result of each one-hot coverage vector reflects the failing possibility of its covering statement.

We will first formulate the problem; then explain how to construct a graph from a subject program to represent a failure context; next depict the training phase, i.e., the process of learning node embeddings on graphs; finally describe the testing phase, i.e., the process of obtaining the fault localization result, which is a ranked list of the statements in descending order of suspiciousness.

A. Formulation

Given a faulty program *P* with *N* statements, it is executed by a test suite *T* with *M* test cases, which contain at least one failed test case. In Fig. 2, a matrix and an error vector are defined to denote the execution information of a program. CAN will use the matrix and the error vector (i.e., the upper left matrix and error vector in Fig. 4) as the training samples and their corresponding labels for graph neural networks, respectively. As a reminder, the element $x_{ij} = 1$ means that the statement *j* is executed by the test case *i*, and $x_{ij} = 0$ means otherwise. The element e_i equals to 1 if test case *i* failed, and 0 otherwise. Based on the faulty program and the information model (i.e., the matrix and error vector), CAN should figure out how to define and incorporate a failure context into suspiciousness evaluation of fault localization using graph neural networks.

B. Modeling Failure Context as Graph

It is easy to see that the information model defined in Fig. 2 contains no relationships among the statements in the error propagation process. Therefore, we need to construct a graph where nodes denote statements (or other program entities) and edges represent relationships among the statements. Program dependency graphs (PDG) [18] are the widely used graph structure for modeling the relationships among the statements, where nodes are statements (or other program entities) and edges are data or control dependencies between two statements. Unlike traditional machine learning techniques, the learning process of GNNs is to update the parameters and hidden state of nodes by capturing the dependencies of a graph via the message passing on edges. However, the PDG of the whole program usually consists of many subgraphs reflecting no relationship with the error output. Furthermore, when we input the PDG of the whole program including these subgraphs into CAN, the learning process may not perform well, and even cannot be convergent. Program slicing technique [18] is amongst the most popular ones to model a failure context. Based on the failure context, we can construct a connected PDG related to a failure with a smaller size in comparison to the PDG of the whole program.

Specifically, we construct a PDG in which the nodes denote statements in the failure context (i.e., the slice) and the edges

TABLE II
SUMMARY OF SUBJECT PROGRAMS

Program	Description	Versions	KLOC	Test
chart	JFreeChart	26	96	2205
math	Apache Commons Math	106	85	3602
lang	Apache commons-lang	65	22	2245
time	Joda-Time	27	53	4130
python	General-purpose language	8	407	3551
gzip	Data compression	5	491	12
libtiff	Image processing	12	77	78
space	ADL interpreter	38	6.1	13585
nanoxml_v1	XML parser	7	5.4	206
nanoxml_v2	XML parser	7	5.7	206
nanoxml_v3	XML parser	10	8.4	206
nanoxml_v5	XML parser	7	8.8	206

represent data or control dependencies. We utilize JSlice[5] and Javaslicer[6] for Java programs, WET[7] for C programs. They are all dynamic slicing tools [18], [36] relying on PDG. Based on these slicing tools, the slicing process of CAN is: 1) we convert the executed statements of a program into a representation, i.e., a form representing a PDG with data and control dependencies; 2) then we analyze the representations via dynamic data or control dependencies to identify those statements having dependencies on the program output (i.e., the slicing criterion in (2)) as a slice; 3) finally we use the API/method of these slicing tools to interpret the representation to construct a PDG for the slice, where a node is a statement in the slice and an edge of two nodes represents a data or control dependency among them.

Thus, a failure context is defined as follows:

A failure context: statements that directly or indirectly affect the computation of the faulty output value of a failure through chains of dynamic data or control dependencies are included in a failure context (i.e., a dynamic backward slice).

To compute a failure context, we use the following slicing criterion.

$$failSC = (outStm, incorrectVar, failEx) \tag{2}$$

Where, $outStm$ is an output statement whose value of a variable (i.e., $incorrectVar$) is incorrect in the execution of a failing test case (i.e., $failEx$). Since a small size of a failure context can help reduce the complexity of the subsequent analysis and save the computation cost, a small size of a failure context is preferable. Take the version 3 of the program nanoxml_v5 in Table II as an example. We use the smallest failure context and the largest failure context to conduct the localization process, respectively. The computation cost of the smallest one is 13.7 minutes (i.e., the lowest cost) while the cost of the largest one is 17.6 minutes (i.e., the highest cost). For multiple failing test cases, the one with the smallest number of executed statements usually has a small size of a failure context. Thus, CAN chooses the failing test case having the smallest number of executed statements to construct a slicing criterion in (2), and inputs this slicing criterion into program slicing technique to model a failure context. If there are several failing test cases with the same smallest number of executed statements, we randomly choose one of them to perform program slicing.

Suppose that a failure context has K statements. As shown in Fig. 4, after the computation of using the slicing criterion in (2), program slicing technique constructs a failure context as a program dependency graph, i.e., the directed graph $G = (V, \xi)$, showing how K failure-inducing statements of the program (i.e., the statements of a failure context) affect each other in the execution of a failing test case to cause a failure. In the directed graph G, a node (e.g., $v_i \in V$) represents a statement in the failure context, and an edge (e.g., $(v_i, v_j) \in \xi$) denotes a data or control dependency between the two statements of v_i and v_j. Each node (e.g., $v_i \in V$) will be embedded into a unified embedding space, and has its node vector (e.g., $vector_i \in R_d$ in (5)) representing the latent vector of the node (e.g., v_i) learned via graph neural networks, where d is the dimensionality.

Furthermore, we remove the coverage information of those statements not in the failure context and obtain a new $M \times K$ matrix that records the execution information of K failure-inducing statements in the test suite T. The element $y_{ij} = 1$ means that the j-th statement of the failure context (i.e., node v_j in the graph) is executed by test case i, and 0 otherwise.

After constructing the graph, the connection relationships between the graph nodes are further represented by an adjacency matrix A, i.e., the adjacency matrix is the representation of the failure context (i.e., the PDG of the slice) An element A_{ij} means the one in the i-th row and the j-th column of the matrix A. $A_{ij} = 1$ means that there is a directed edge e_{ij} from node i to node j; otherwise, $A_{ij} = 0$. A_{ij} shows whether the information of node i is propagated to node j. Some data or control dependencies in the PDG are spurious, e.g., $y = 3 * x - 3 * x + 1$ does not effectively depend on x. In this case, a simple data-flow analysis that does not simplify the expression will not realize it. Consequently, our approach may not be able to discount the influence of x on y.

C. Learning CAN Model on Graph (Training Phase)

Next we present how to train CAN model via graph neural networks. Graph neural networks could capture dependencies among statements and generate corresponding statement embedding vectors trained with test cases [25] according to the failure context.

Architecture of CAN: Fig. 4 shows the architecture of CAN including input components, graph embedding module, linear transformation layers and output layer. After the previous steps (i.e., modeling failure context as graph), we could acquire the input components that consist of three parts: an adjacency matrix A, node vectors $vector_i$ ($i \in \{1,2,...,k\}$) and the coverage information matrix $M \times K$. The graph embedding module is a 2-layer graph neural network. In the first layer, the input channel is the dimension of node vector equaling to the size of the failure context and the output channel is 2^h ($h = round(log_2^n) - 1$, where n is the input channel and *round* is the rounding function). In the second layer, the input channel is 2^h and the output

[5]http://jslice.sourceforge.net/
[6]https://github.com/hammacher/javaslicer
[7]http://wet.cs.ucr.edu/

K dimensional

$$\begin{array}{c|cccc} & x_1 & x_2 & \cdots & x_K \\ \hline \text{one-hot}_1 & 1 & 0 & \ldots & 0 \\ \text{one-hot}_2 & 0 & 1 & \ldots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \text{one-hot}_K & 0 & 0 & \ldots & 1 \end{array}$$

Fig. 5. One-hot statement coverage vectors.

channel is $2^{(h-1)}$. Then, a Gated Recurrent Unit (GRU) is used to incorporate the dependencies between pieces of node state information in different update rounds, and linear transformation layers use Sigmoid function. We use Adam as the optimizer. Back propagation is used to update the parameters.

Learning process in graph embedding module: Graph embedding module is the core component of CAN for the learning process. The node vector $vector_i$ is the state information of the node *i*. First, we initialize the state information of node *i* to a vector$vector_i{}^{(1)}\in R^d$ via using one-hot encodings, where *d* is the dimensionality of the vector and the superscript (1) denotes the 1st round of iteration. Then, during the *t*-th round of iteration, each central node *i* gathers all neighbor node information to obtain the node interaction embedding $m_i{}^{(t)}\in R^d$. In (3), CAN assigns different weights to each neighbor node in order to characterize its importance to the central node δ_{ij}, with function mapping performed through the neural network *a:* $R^d \times R^d \rightarrow R$, where *a* calculates the correlation coefficient between the central node *i* and its neighbor node *j* and then uses the softmax function to normalize the correlation coefficients of all neighboring nodes. The weight parameter of the neural network *a* is related to the round of information propagation. (4) computes the interaction context $m_i{}^{(t)}$ of the node. During the graph model's information propagation, $m_i{}^{(t)}$ is the interaction context of node *i* throughout the whole graph. $m_i{}^{(t)}$ is obtained by directly accumulating δ_{ij} and the product of the feature information $vector_j$ of the neighbor node *j* as well as the propagation matrix A_{ij} on the edge e_{ij}. In (5), the GRU considers the relationship between pieces of node state information in different update rounds, i.e., when the node updates during the *t*-th round, there is a time series relationship between the hidden layer vector expression $vector_i{}^{(t)}$ and the state information $vector_{(i)}{}^{(t-1)}$ of the previous round.

$$\delta_{ij} = soft\max(a(vector_i^{(t-1)}, vector_j^{(t-1)})) \tag{3}$$

$$m_i^{(t)} = \sum\nolimits_{j\in N_i} \delta_{ij} \cdot A_{ij} \cdot vector_j \tag{4}$$

$$vector_i^{(t)} = GRU(vector_i^{(t-1)}, m_i^{(t)}) \tag{5}$$

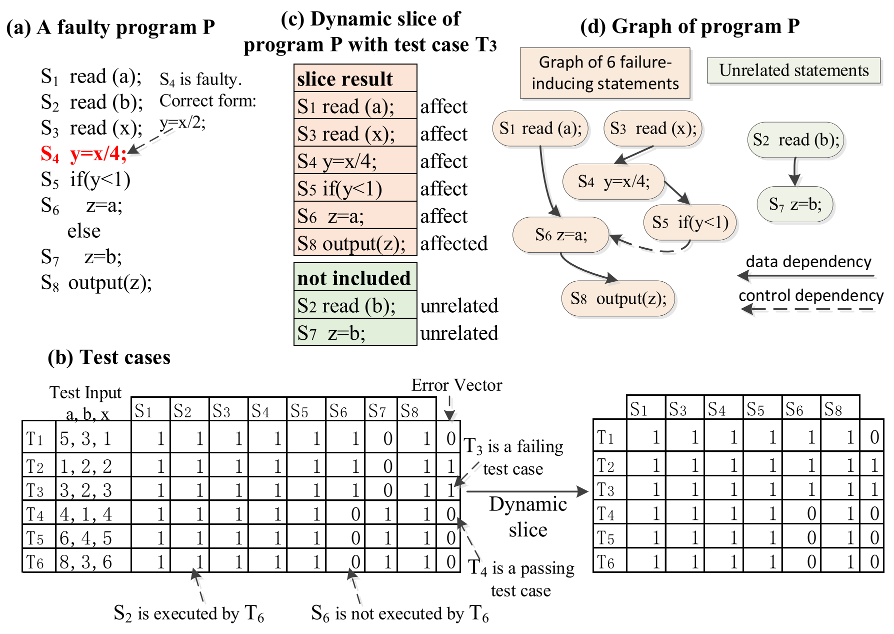
Training Procedure: During training, CAN utilizes graph embedding module and linear transformation layers to update the parameters and node vectors of $vector_1$ to $vector_K$. Specifically, each training iteration will conduct a node state information propagation process. Suppose that CAN uses the i-th row of the matrix *M*×*K* (i.e., [y_{i1},y_{i2},...,y_{iK}]) and its error vector (i.e., e_i). For y_{ij}, $y_{ij} = 1$ means that node v_j is executed by the test case i (i.e., T_i), and CAN feeds its node vector $vector_j$ into the neural network; $y_{ij} = 0$ means that node v_j is not executed by T_i, and CAN feeds zero vector into the neural network. After executing test case T_i, we choose the executed failure-inducing statements of T_i and feed their corresponding node vectors into the GNN components and conduct the node state information propagation process while keeping the node vectors of those unexecuted failure-inducing statements unchanged (i.e., the values are the result of the last training step). After node state information propagation process, there are linear transformation layers to output the result o_i ($i\in\{1,2,...,M\}$), the value of which ranges from 0 to 1. o_i has a difference with the corresponding target vector e_i, and the difference is the loss between the output layer and target. CAN will iterate the above process by traversing from the 1st to the last row of the matrix *M*×*K* and its error vector one by one, and using back propagation algorithm to update the parameters of the model and node vectors of $vector_1$ to $vector_K$ in graph *G*. Thea goal is to minimize the loss between training result *o* and error vector *e*. The algorithm goes forward from the input components calculating the outputs of each layer up to the output layer. Then it starts calculating derivatives going backwards through the layers and propagating the results to do less calculation by reusing all of the elements already calculated. We repeat the training of the model with the matrix *M*×*K* until the loss is small enough to reach convergence. Thus, the number of times of each test case selected during training is almost the same. As a reminder, according to the recent studies [59], [60], different types of test cases have different impacts on fault localization effectiveness, and thus the number of times of each test case selected during training may affect the learning outcome. Studying the impact of test cases on the learning process is another research direction that we leave for future work.

The learning rate impacts the speed of convergence. CAN adopts dynamic adjusting learning rate (see (6)) for two reasons. First, it can make large changes at the beginning of the training procedure when larger learning rate values are used. Second, it can decrease the learning rate with a smaller training updates for computing weights later, resulting in accurate weights more quickly. In (6), one *Epoch* means completing all training data once, *LR* represents learning rate, *DropRate* is the amount that learning rate is modified each time, and *EpochDrop* is how often to change the learning rate.

$$LR = LR * DropRate^{(Epoch+1)/EpochDrop} \tag{6}$$

D. *Suspiciousness Evaluation With Trained Model (Testing Phase)*

After obtaining the parameters of the model and the embedding of failure-inducing statements, the training process will learn a model with trained node vectors, reflecting the complex nonlinear relationship between the execution of a statement and a failure with the consideration of the statement dependencies. Thus, as shown in Fig. 5, we construct *K* one-hot statement coverage vectors as the testing dataset, with a size equal to the number of the failure-inducing statements, where each one-hot vector only covers one statement. Specifically, for a one-hot vector *one-hot*$_i$, x_i =1 means that the *i*-th statement of the failure context (i.e., the failing-inducing statements) is the only

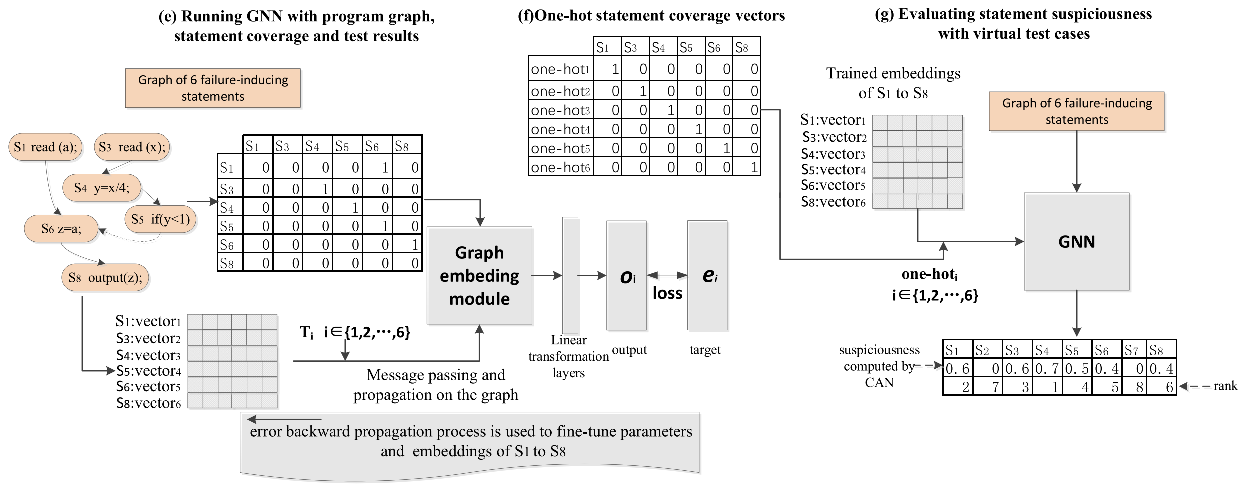


slice result	
S1 read (a);	affect
S3 read (x);	affect
S4 y=x/4;	affect
S5 if(y<1)	affect
S6 z=a;	affect
S8 output(z);	affected
not included	
S2 read (b);	unrelated
S7 z=b;	unrelated

	Test Input a, b, x	S1	S2	S3	S4	S5	S6	S7	S8	Error Vector
T1	5, 3, 1	1	1	1	1	1	1	0	1	0
T2	1, 2, 2	1	1	1	1	1	1	0	1	1
T3	3, 2, 3	1	1	1	1	1	1	0	1	1
T4	4, 1, 4	1	1	1	1	1	0	1	1	0
T5	6, 4, 5	1	1	1	1	1	0	1	1	0
T6	8, 3, 6	1	1	1	1	1	0	1	1	0

	S1	S3	S4	S5	S6	S8	
T1	1	1	1	1	1	1	0
T2	1	1	1	1	1	1	1
T3	1	1	1	1	1	1	1
T4	1	1	1	1	0	1	0
T5	1	1	1	1	0	1	0
T6	1	1	1	1	0	1	0

Fig. 6. An example illustrating CAN (modeling failure context as graph).



(e) Running GNN with program graph, statement coverage and test results
Graph of 6 failure-inducing statements
S1 read (a); S3 read (x); S4 y=x/4; S5 if(y<1) S6 z=a; S8 output(z);
S1:vector1 S3:vector2 S4:vector3 S5:vector4 S6:vector5 S8:vector6
Ti i∈{1,2,⋯,6}
Graph embeding module
Message passing and propagation on the graph
Linear transformation layers
Oi output
loss
ei target
error backward propagation process is used to fine-tune parameters and embeddings of S1 to S8
(f)One-hot statement coverage vectors
(g) Evaluating statement suspiciousness with virtual test cases
Trained embeddings of S1 to S8
S1:vector1 S3:vector2 S4:vector3 S5:vector4 S6:vector5 S8:vector6
Graph of 6 failure-inducing statements
GNN
one-hoti i∈{1,2,⋯,6}
suspiciousness computed by CAN
rank

	S1	S3	S4	S5	S6	S8
S1	0	0	0	0	1	0
S3	0	0	1	0	0	0
S4	0	0	0	1	0	0
S5	0	0	0	0	1	0
S6	0	0	0	0	0	1
S8	0	0	0	0	0	0

	S1	S3	S4	S5	S6	S8
one-hot1	1	0	0	0	0	0
one-hot2	0	1	0	0	0	0
one-hot3	0	0	1	0	0	0
one-hot4	0	0	0	1	0	0
one-hot5	0	0	0	0	1	0
one-hot6	0	0	0	0	0	1

	S1	S2	S3	S4	S5	S6	S7	S8
suspiciousness computed by CAN	0. 6	0	0. 6	0. 7	0. 5	0. 4	0	0. 4
rank	2	7	3	1	4	5	8	6

Fig. 7. An example illustrating CAN (training and testing).

one element that is covered. When the $one\text{-}hot_i$ is inputted to the trained model, the output is the estimation of execution result of being a failure by covering only the i-th statement of the failure context. The value of the result is between 0 and 1. The larger the value is, the more likely it is that the statement only covered by the $one\text{-}hot_i$ is the buggy statement to cause a failure. In this way, we can evaluate the suspiciousness of each statement being faulty by inputting the one-hot statement coverage vectors into the trained model. For those statements not in the failure context, CAN will assign the lowest suspiciousness value to them. When we encounter two or more erroneous statements, GNN may rank them with one above the other and we recognize the one with the higher rank as the final result. In different runs, the rank may be different but in a relatively small fluctuation range.

E. An Illustrative Example

Figs. 6 and 7 illustrate a faulty program P with 8 statements that contains a faulty statement S_4 to show how CAN works. Fig. 6(a) presents a program P with a faulty statement S_4. Fig. 6(b) shows 6 test cases with T_2 and T_3 being the failing test cases. Since the failing test cases of T_2 and T_3 have the same smallest number of executed statements, we randomly choose T_3 to construct the failure context. Fig. 6(c) shows the dynamic slice result of program P with the test case T_3 (i.e., the failure context), in which six statements are included while two ones are not included. We could observe the relationships among the statements in the slice result, S_1, S_3, S_4, S_5 and S_6 affect the output value of variable z at S_8. Fig. 6(d) illustrates graph of program P (i.e., the graph representation of the failure context), which lists the statements in the slice along the program dependence edges including both data dependencies and control dependencies from the starting point.

Fig. 7(e) shows the training process of CAN. CAN converts the program dependency graph of failure-inducing statements into the adjacency matrix and input it to GNN model. Then it trains the GNN model with the coverage data and inputs the errors vector into the target vector. There are 6 vectors representing the node embeddings of 6 statements on the graph. For example, S_1 is a node on the graph of program *P*, $vector_1$ is S_1's node embedding. Concretely, according to T_1=[1, 1, 1, 1, 1, 1] and its test result 0 on the right side of Fig. 6(b), we input ($vector_1$, $vector_2$, $vector_3$, $vector_4$, $vector_5$, $vector_6$) with the target 0 to the model; according to T_2=[1, 1, 1, 1, 1, 1] and its test result 1, we input trained ($vector_1$, $vector_2$, $vector_3$, $vector_4$, $vector_5$, $vector_6$) of the last step with the target 1 to the model; according to T_3=[1, 1, 1, 1, 1, 1] and its test result 1, we input trained ($vector_1$, $vector_2$, $vector_3$, $vector_4$, $vector_5$, $vector_6$) with the target 1 to the model; according to T_4=[1, 1, 1, 1, 0, 1] and its test result 0, we input trained ($vector_1$, $vector_2$, $vector_3$, $vector_4$, zero vector, $vector_6$) with the target 0 to the model; according to T_5=[1, 1, 1, 1, 0, 1] and its test result 0, we input trained ($vector_1$, $vector_2$, $vector_3$, $vector_4$, zero vector, $vector_6$) with the target 0 to the model; according to T_6=[1, 1, 1, 1, 0, 1] and its test result 0, we input trained ($vector_1$, $vector_2$, $vector_3$, $vector_4$, zero vector, $vector_6$) with the target 0 to the model. Repeat training the network with these data until the loss is small enough and reaches the condition of convergence. According to the small number of statements in this example, we set the convergence condition as that the value of loss between training result o and error vector e does not decrease for five epochs. After training, the GNN model reveals the complex nonlinear relationship between the statement embeddings in the graph and failures.

Finally, CAN constructs 6 one-hot statement coverage vectors (see Fig. 7(f)), where each only covers one statement. We input a one-hot vector into the trained network, and the output is the suspiciousness of the statement covered by the one-hot vector. For example, we input the one-hot vector $one\text{-}hot_1 = [1, 0, 0, 0, 0, 0]$ into the trained network, and the output is the suspiciousness of S_1, i.e., 0.6. In the same way, we could compute the suspiciousness of other statements. Since S_2 and S_7 are not included in the failure context, CAN assigns the lowest suspiciousness value to them, i.e., the value of 0. As depicted in Fig. 7(g), there is a ranked list of all statements in descending order of suspiciousness: (S_4, S_1, S_3, S_5, S_6, S_8, S_2, S_7). The faulty statement S_4 is ranked in the first place.

IV. Experiments

A. Experimental Setup

Benchmarks: The experiments use the widely used large-sized programs (e.g., [1], [2], [12], [13], [14], [46]) in fault localization, reflecting characteristic of practice and enabling comparable and reproducible studies. Table II summarizes the subject programs. For each program, it depicts a brief functional description (column 'Description'), the number of faulty versions (column 'Versions'), the number of thousand lines of statements (column 'KLOC'), and the number of test cases (column 'Test'). The first 4 programs [61] (i.e., *chart*, *math*, *lang*, and *time*) are from Defects4J.[8] The *python*, *gzip* and *libtiff* are collected from ManyBugs.[9] The *space* and the four separate releases of *nanoxml* are acquired from the SIR.[10]

We use JSlice, Javaslicer and WET for slicing Java and C Programs respectively. These tools cannot slice some faulty versions (e.g., all faulty versions of *Closure* in Defects4J), and we remove these versions in our evaluation.

Baselines: Prior studies [12], [37], [43], [44] have conducted theoretical [43], [44] and empirical analysis [12] on finding the optimal SFL formulas and their combination, i.e., Ochiai, ER5, GP02, GP03, Dstar, GP19 and ER1, MULTRIC. Furthermore, the recent results [13], [14], [16], [62] on DLFL have identified four representative and effective ones, i.e., MLP-FL, CNN-FL, BiLSTM-FL and DeepRL4FL. Therefore, the experiments use the 12 state-of-the-art fault localization approaches as the baselines. We implement the 12 baselines including the parameters as described in their publications.

Parameter settings and environment: We apply a grid search to identify the best settings of hyper-parameters: the initial learning rate *l* is tuned in {0.0001, 0.001, 0.002, 0.005, 0.01}, the dropout rate *d* is searched in {0.98, 0.96, 0.95, 0.94, 0.93} and the batch size β is in {10, 16, 32, 64}. To prevent overfitting, we tuned the *L2* regularization λ in $\{10^{-5}, 10^{-3}, 10^{-1}\}$. We utilize adam optimizer in CAN. The physical environment of the experiments is on a computer containing a CPU of Intel I5-2640 with 128G physical memory, and two 12G GPUs of NVIDIA TITAN X Pascal. The operating system is Ubuntu 16.04.3. We computed statistics of the data and plot display on the MATLAB R2016b.

B. Evaluation Metrics

We adopt 4 widely used metrics: *Top-N Accuracy:* [63], [64], *Mean Average Rank (MAR)* [46], *Mean First Rank (MFR)* [46], and *Relative Improvement (RImp)* [50], [65], [66]. A higher value of *Top-N Accuracy* means better localization effectiveness, while a lower value denotes better localization effectiveness for the other 3 metrics.

Top-N Accuracy: It denotes the percentage of faults located within the first *N* position of a ranked list of all statements in descending order of suspiciousness returned by a fault localization approach.

Mean Average Rank (MAR): It is the mean of the average rank of all faults using a localization approach.

Mean First Rank (MFR): For a fault with multiple faulty statements, locating the first one is critical since the others may be located after that. *MFR* is the mean of the first faulty statement's rank of all faults using a localization approach.

Relative Improvement (RImp): It is to compare the total number of statements that need to be examined to find all faults using CAN versus the number that need to be examined by using other fault localization approaches.

[8]Defects4J, http://defects4j.org
[9]ManyBugs, http://repairbenchmarks.cs.umass.edu/ManyBugs/
[10]SIR, http://sir.unl.edu/portal/index.php

TABLE III
Top-N, *MAR* and *MFR* Comparison of CAN Over 12 Fault Localization Approaches

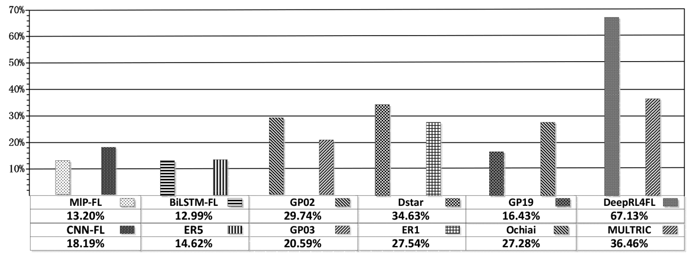
Comparison	top-1	top-3	top-5	MFR	MAR
CAN	**4.62%**	**20%**	**29.23%**	**49**	**75**
MLP-FL	0%	0%	1.54%	207	341
CNN-FL	1.54%	3.08%	3.08%	186	251
BiLSTM-FL	0%	0%	0%	221	403
DeepRL4FL	4.27%	14.93%	25.57%	127	215
ER5	0%	4.62%	6.15%	223	409
GP02	1.54%	7.69%	15.38%	257	573
GP03	3.08%	4.62%	10.77%	202	353
Dstar	3.08%	7.69%	21.54%	235	372
ER1′	3.08%	9.23%	18.46%	233	362
GP19	3.08%	9.23%	9.23%	251	388
Ochiai	1.54%	7.69%	20%	201	352
MULTRIC	3.08%	9.23%	23.08%	197	254

C. Experimental Results

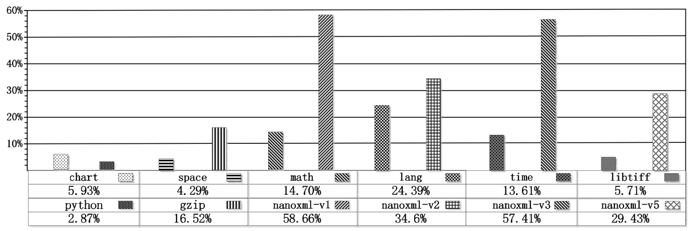
Top-N Accuracy, MAR and MFR: Parnin and Orso [64] conducted a user study of evaluating the usefulness of fault localization techniques in assisting developers, and recommend using the rank of the faulty statement to evaluate fault localization effectiveness. Since then, *Top-N*, *MAR* and *MFR* are widely used in fault localization. Afterwards comprehensive user studies (e.g., [38], [63]) show that it useful to help developers in debugging by setting Top-N within Top-5. For automated program repair (APR), since the APR techniques generate patches in the suspicious rank list from top to down and many APR techniques (e.g., SimFix [67] and TBar [68]) set clear time limitation, the improvement of Top-N, MAR and MFR metrics could also help the APR techniques that rely on the suspicious rank list and have limited time for each bug during the repair. Thus, our experiments use *Top-N* (i.e., N=1, 3, 5), *MAR*, and *MFR* to compare CAN with the 12 baselines. Table III presents their distribution among 12 fault localization approaches. As shown in Table III, CAN achieves very promising best localization effectiveness in all 5 scenarios in comparison to the baselines. Specifically, CAN can localize 4.62%, 20%, 29.23% faults when inspecting the Top-1, Top-3 and Top-5 ranked statements, The *MAR* and *MFR* are 49 and 75 respectively, achieving (127-49)/127=61.42% and (215-75)/215=65.12% relative improvement over each best-performing baseline respectively.

RImp distribution: For a detailed improvement, we adopt *RImp* to evaluate CAN. Fig. 8 shows the *RImp* distribution of our approach in two cases: the *RImp* on 12 fault localization approaches in Fig. 8(a) and the *RImp* on 12 subject programs in Fig. 8(b).

As shown in Fig. 8(a), the *RImp* score is less than 100% in all approaches, meaning that CAN improves localization effectiveness of all the fault localization approaches. The statements that need to be examined decrease ranging from 12.99% in *BiLSTM-FL* to 67.13% in *DeepRL4FL*. It also means that CAN, in comparison to the other approaches, obtains a maximum saving of 87.01% (100%-12.99%=87.01%) in *BiLSTM-FL* and the minimum saving is 32.87% (100%-67.13%=32.87%) in *Dstar* which indicates that CAN can save from 32.87% to 87.01% of the number of statements examined among the fault localization approaches.



(a) *RImp* on fault localization approaches.



(b) *RImp* on subject programs.

Fig. 8. *RImp* distribution of CAN on fault localization approaches and subject programs.

As shown in Fig. 8(b), the *RImp* score is less than 100% in all subject programs, meaning that CAN obtains improvements on all the programs. The statements that need to be examined decrease ranging from 2.87% in *python* to 58.66% in *nanoxml_v1*. It means that CAN needs to examine from 2.87% to 58.66% of statements that all the 12 fault localization approaches need to examine for locating all faults of the program *python* and *nanoxml_v1*, repsectively. Thus, CAN obtains a maximum saving of 97.13% (100%-2.87%=97.13%) in *python* and a minimum saving of 41.34% (100%-58.66%=41.34%) in *nanoxml_v1*, which indicates that CAN can save from 41.34% to 97.13% of the number of statements examined among all the programs.

Statistical comparison: To investigate whether the difference between the baselines and CAN is statistically significant, we adopt Wilcoxon-Signed-Rank Test [69] with a Bonferroni correction [70], which is a non-parametric statistical hypothesis test for testing the differences between pairs of measurements F(x) and G(y). The experiments performed 12 paired Wilcoxon-Signed-Rank tests by using the *ranks* of the faulty statements as the pairs of measurements F(x) and G(y). Each test uses both 2-tailed and 1-tailed p-value checking at the σ level of 0.05. Specifically, given a localization technique FL1, we use the list of the *ranks* of the faulty statements using CAN in all faulty versions of all programs as the list of measurements of F(x), while the list of measurements of G(y) is the list of the *ranks* of the faulty statements using FL1 in all faulty versions of all programs. Hence, in the 2-tailed test, CAN has SIMILAR effectiveness as FL1 when H_0 is accepted at the significant level of 0.05. And in the 1-tailed test (right), CAN has WORSE effectiveness than FL1 when H_1 is accepted at the significant level of 0.05. Finally, in the 1-tailed test (left), CAN has BETTER effectiveness than FL1 when H_1 is accepted at the significant level of 0.05.

TABLE IV
STATISTICAL RESULTS OF CAN OVER 12 LOCALIZATION APPROACHES

Dataset	MLP-FL	CNN-FL	BiLSTM-FL	DeepRL4FL	ER5	GP02	GP03	Dstar	ER1	GP19	Ochiai	MULTRIC
chart	0.67*	0.33**	0.78*	0.63*	0.84*	0.89**	0.9**	0.72*	0.78*	0.84**	0.72*	0.73*
gzip	1**	0.94*	1**	0.71*	0.94**	0.88*	0.88*	0.88*	0.88*	0.88**	1**	0.83*
libtiff	0.89**	0.89*	0.89*	0.69*	1**	1**	1**	1**	1**	1**	1***	1**
math	1**	1**	0.75*	0.74*	0.76*	0.74**	0.80*	0.65*	0.78*	0.68*	0.75*	0.81*
lang	0.79**	0.77*	0.9**	0.68*	0.72*	0.69*	1***	0.84*	0.74*	0.77*	0.78*	0.85*
python	1***	1**	1**	0.66*	1**	1**	1**	1***	1**	1**	1**	1**
space	1***	1***	1***	0.72*	1***	0.99***	0.99***	0.98***	0.99***	0.98***	0.98***	0.98***
time	0.72*	0.72*	1**	0.71*	1**	1***	0.78*	0.66*	0.68*	0.71*	0.69*	0.78*
nanoxml_v1	0.89**	0.78*	1**	0.73*	1***	0.56*	0.56*	0.76*	0.76*	1***	0.74*	1***
nanoxml_v2	0.89**	1**	0.89*	0.74*	1**	0.72*	0.83*	0.82*	0.83**	1***	0.78*	1***
nanoxml_v3	0.97**	0.89*	0.89*	0.76*	1***	0.54*	0.58*	0.78*	0.58*	0.81*	0.67*	0.83*
nanoxml_v5	0.84**	0.80*	0.84*	0.79**	0.92**	0.92**	0.92**	0.82*	0.92**	0.84**	0.72**	0.88**

***p<0.001, **p<0.01 and *p<0.05

To further assess the difference quantitatively, we use the nonparametric Vargha-Delaney A-test [71], to evaluate the magnitude of the difference by measuring effect size (scientific significance). For A-test, the bigger deviation of A-statistic from the value of 0.5, the greater difference of the two studied groups. A-test of greater than 0.64 (or less than 0.36) is indicative of ”medium“ effect size, and of greater than 0.71 (or less than 0.29) can be indicative of a promising ”large“ effect size [72].

Table IV shows the statistical results on this relationship, where the cells show the A-test values and different rages of p values of Wilcoxon-Signed-Rank Tests (i.e., ***p<0.001, **p<0.01 and *p<0.05). The p values are all less than 0.05, i.e., the *ranks* of the faulty statements of CAN are significantly smaller than those of all the 12 baselines in all programs. In addition, CAN mostly arrives at the promising “large” effect size, showing the significant amount of the difference between CAN and the 12 localization approaches. Therefore, it is statistically significant that CAN outperforms all the 12 baselines with a BETTER conclusion.

Thus, based on all the results and analysis, we can safely conclude that CAN significantly improves effectiveness of fault localization, showing that graph neural networks are potential to analyze and learn the transitive relationships of a failure context for improving fault localization effectiveness.

V. DISCUSSION

A. *Why is CAN Better?*

The experimental results show that CAN outperforms 12 localization approaches. It is natural to seek the factors from the characteristics of CAN over other architectures to understand why CAN significantly improves fault localization.

Let us use the example in Fig. 6 to understand the characteristics. SFL just uses the coverage information (i.e., a statement executed or not executed) denoted as a 6 × 8 matrix in Fig. 6(b) to identify the relationship between the statement and failures. It does not consider the relationships among statements of causing a failure. In addition to coverage information, CAN models a failure context as a program dependency graph and then use graph neural networks to successfully capture the error propagation path: $S_4 \rightarrow S_5 \rightarrow S_6 \rightarrow S_8$. When CAN trains the model, node state information propagation process successfully simulates the error propagation on the path $S_4 \rightarrow S_5 \rightarrow S_6 \rightarrow S_8$. Therefore, from the above characteristics of CAN, learning the complicated relationships in a context graph may be the key factor of CAN in contributing to fault localization effectiveness.

To verify the above discussion, we conduct the following experiments for two goals. One is to check whether learning the transitive relationships contributes to localization effectiveness. The other one is identify to what extent leaning the transitive relationships contributes to localization effectiveness.

Does learning the transitive relationships of CAN contribute to its effectiveness?: There are two types of relationships (i.e., edges): data dependencies and control dependencies. Thus, we use three cases: 1) randomly removing some of the edges (i.e., data and control dependencies) of the original graph of a failure context (denoted as CAN-R); 2) removing data dependencies and keeping control dependencies of the original graph of a failure context (denoted as CAN-C); 3) removing control dependencies and keeping data dependencies of the original graph of a failure context (denoted as CAN-D). Specifically, we randomly remove 20%-40% edges of the original graph of a failure context, and their removal will preserve the connectedness of the graph, resulting a new connected graph. If learning the transitive relationships in a context graph is a factor of CAN in contributing fault localization effectiveness, the effectiveness of CAN with the reduced graphs (i.e., CAN-R, CAN-C and CAN-D) should decrease. Table V shows the statistical results of the comparison of original CAN with the three cases in all programs using Vargha-Delaney A-test and Wilcoxon-Signed-Rank Test at the σ level of 0.05. As shown in Table V, all the p values are less than 0.05 and the most A-test values belong to the “medium” and “large” effect sizes. The results show that the effectiveness of CAN with all three cases decrease.

Thus, the results show that learning the transitive relationships in a context graph is a factor of CAN in contributing to fault localization effectiveness.

TABLE V
STATISTICAL RESULTS OF CAN OVER CAN-R, CAN-C AND CAN-D

Program	CAN vs CAN-R	CAN vs CAN-C	CAN vs CAN-D
chart	0.74**	0.81***	0.72**
gzip	0.72**	0.79**	0.71**
libtiff	0.64*	0.82***	0.74**
math	0.65**	0.79**	0.66*
lang	0.55*	0.81***	0.61*
python	0.67*	0.86***	0.63*
space	0.62*	0.83***	0.69*
time	0.71**	0.82***	0.58*
nanoxml_v1	0.71**	0.81***	0.68*
nanoxml_v2	0.83***	0.82***	0.71**
nanoxml_v3	0.74**	0.89***	0.64*
nanoxml_v5	0.81***	0.88***	0.75**
***p<0.001, **p<0.01 and *p<0.05			

TABLE VI
AVERAGE SIZE AND VARIANCE OF A FAILURE CONTEXT IN EACH PROGRAM

Program	chart	gzip	libtiff	math	lang	python
Size	15.85%	21.65%	21.57%	17.30%	22.13%	19.36%
Variance	0.06	0.09	0.13	0.02	0.07	0.07
Program	space	time	nanoxml_v1	nanoxml_v2	nanoxml_v3	nanoxml_v5
Size	13.17%	10.72%	44.60%	32.60%	37.62%	32.36%
Variance	0.11	0.07	0.12	0.10	0.08	0.10

TABLE VII
AVERAGE NUMBER AND VARIANCE OF OUTGOING BRANCHES PER NODE IN EACH PROGRAM

Program	chart	gzip	libtiff	math	lang	python
Number	4.98	12.95	5.93	7.68	7.61	9.25
Variance	3.39	1.71	7.37	10.45	12.34	3.46
Program	space	time	nanoxml_v1	nanoxml_v2	nanoxml_v3	nanoxml_v5
Number	4.93	5.33	5.48	4.96	6.13	5.39
Variance	4.86	4.25	7.62	6.31	5.14	6.17

TABLE VIII
TOP-N, *MAR* AND *MFR* COMPARISON OF CAN OVER CAN-RMSLICE

Comparison	top-1	top-3	top-5	MFR	MAR
CAN	4.62%	20%	29.23%	49	75
CAN-rmSlice	3.08%	12.96%	22.94%	94	185

To what extent does leaning the transitive relationships of CAN contribute to its effectiveness?: We use the 12 baselines working on a failure context without considering those statements not in a failure context, and compare CAN with the 12 baselines using a failure context (denoted as baseline-context). Since CAN and baseline-context work on the same failure contexts, if CAN significantly outperform the 12 baselines-context, it means that CAN is more effective to learn the transitive relationships whereas the 12 baselines is ineffective. It can also show that learning the transitive relationships of CAN significantly contributes to fault localization effectiveness. Furthermore, Some may argue that CAN removes a set of statements from the original program to construct a failure context (i.e., a program slice).

First, the average size of a failure context is 21% of the original programs, which is much more than the rank of the faulty statements CAN outputs (e.g., see Table III). Table VI shows the percentage which the average size of a failure context accounts for each program and its variance. As shown in Table VI, *nanoxml_v1*, *nanoxml_v2*, *nanoxml_v3* and *nanoxml_v5* have larger percentage than the other programs. Since the program dependency graph is constructed from the failure context (i.e., the slice), the failure context with a larger size may lead to more redundant statements involved in the learning process. It can affect the effectiveness of CAN. Thus, as shown in Fig. 8(b), CAN in these programs has lower effectiveness (i.e., higher *RImp* scores) as compared with the other programs. Furthermore, since the connectedness of the dependence matrix (i.e., the failure context) should affect both the computational cost and the sample efficiency of learning, Table VII shows the information of the adjacency matrices including the average number and the variance of the number of the outgoing branches per node in each subject. As shown in Table VII, the average number and the variance in all programs are 6.72 and 6.09, respectively.

Next, we further exclude slicing technique and use the graph of original source code for CAN (denoted as CAN-rmSlice) to check whether a failure context (i.e., a slice) contributes to the effectiveness of CAN. Due to the use of graph neural networks, CAN collectively aggregates information from graph structure [24], [25], i.e., the learning process of GNNs is to update the parameters and hidden state of nodes by capturing the dependencies of a graph via the message passing on edges. Thus, the input graph in CAN should be a connected graph. However, the PDG of the whole program may consist of numerous disconnected subgraphs, among which there are no correlations with each other and no message passing behavior. It means that when input the graph (i.e., PDG) of the whole program, the learning process of CAN works not well and even cannot reach a convergent result. To address the problem, we choose the subgraph which has correlation with the faulty output for CAN. Table VIII shows the comparison of CAN and CAN-rmSlice. The results show that CAN outperforms CAN-rmSlice, showing that a failure context (i.e., a slice) contributes to the effectiveness of CAN.

Thus, it seems necessary to use slicing to construct a failure context to extract the connected graph as small as possible while deleting redundant statements, which could reduce the learning noise for the model. It means that slicing could also focus the model to the likely failure context and improve the scalability of training. CAN uses a failing test case to construct a failure context. Thus, it is natural to raise a question whether CAN can be improved by using the relationships among different test cases to construct a slice even if the slice has a larger size. Since passing test cases do not show a failure, slicing is widely used on failing test cases for debugging [2], [19]. Therefore, we use the union of the slices of all failing test cases to represent the relationships among different test cases. We compare CAN using the union (denoted as CAN-U) with original CAN. Table IX shows the statistical results of this comparison in all programs using Vargha-Delaney A-test and Wilcoxon-Signed-Rank Test at the σ level of 0.05. As shown in Table IX, all the p values are less than 0.05 and all the A-test values belong to the promising "large" effect size. The results show that original CAN performs better than CAN-U. We observe that the average size of original failure context is 27.46%, 23.49%, 25.21%, 24.79%, 57.78%, 19.37%, 79.33%, 78.14%, 84.57%, and 81.48% of the union in *libtiff*, *math*, *lang*, *python*, *space*, *time*, *nanoxml_v1*, *nanoxml_v2*, *nanoxml_v3* and *nanoxml_v5*, respectively. Thus, the reason may be that CAN already shows a complete failure

TABLE IX
STATISTICAL RESULTS OF CAN OVER CAN-U

Program	Result	Program	Result
chart	0.76**	space	0.78**
gzip	0.75**	time	0.79***
libtiff	0.79***	nanoxml_v1	0.82***
math	0.77**	nanoxml_v2	0.80**
lang	0.81**	nanoxml_v3	0.83***
python	0.75**	nanoxml_v5	0.85***

***p<0.001, **p<0.01 and *p<0.05

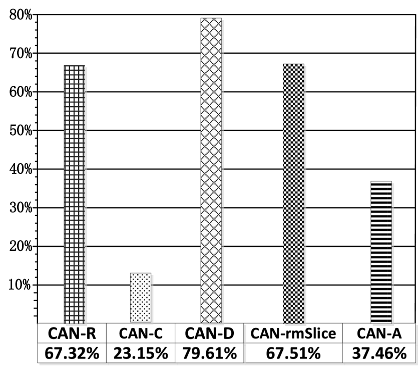


Fig. 9. *RImp* distribution of five approaches related to the ablation experiments of failure contexts.

TABLE X
STATISTICAL RESULTS OF CAN OVER 12 BASELINES-CONTEXT

Comparison	Result	Comparison	Result
CAN vs MLP-FL-context	0.77**	CAN vs GP03-context	0.82**
CAN vs CNN-FL-context	0.73*	CAN vs Dstar-context	0.71*
CAN vs BiLSTM-FL-context	0.87**	CAN vs ER1-context	0.74**
CAN vs DeepRL4FL-context	0.68*	CAN vs GP19-context	0.72*
CAN vs ER5-context	0.81**	CAN vs Ochiai-context	0.73*
CAN vs GP02-context	0.83**	CAN vs MULTRIC-context	0.71*

***p<0.01, **p<0.1 and *p<0.5

with a smaller size beneficial for the learning process even if the union considers the relationships among different test cases.

For a further evaluation, we adopt *RImp* to evaluate CAN with the above five approaches (i.e., CAN-R, CAN-C, CAN-D, CAN-rmSlice and CAN-U) related to the ablation experiments of failure contexts. Fig. 9 shows the *RImp* distribution. The *RImp* scores are all less than 100%, showing that CAN outperforms the five approaches. The results are also consistent with the above evaluation.

Finally, due to the noise reduction advantage of slicing, it is also desirable to compare CAN with 12 baselines using a failure context (denoted as baseline-context) to see whether CAN outperforms 12 baselines using a failure context (i.e., a sliced program). Table X shows the statistical results of CAN versus each of 12 baselines-context using Vargha-Delaney A-test and Wilcoxon-Signed-Rank Test at the σ level of 0.05. As shown in Table X, all the p values are less than 0.05 and all the A-test values belong to the promising "large" effect size. Thus, the results also show it is statistically significant that CAN still outperforms all the baselines on a failure context with a BETTER conclusion.

TABLE XI
STATISTICAL COMPARISON OF CAN OVER CAN-NODECONTENT

Program	Result	Program	Result
chart	0.51	space	0.74*
gzip	0.56	time	0.51
libtiff	0.53	nanoxml_v1	0.49
math	0.47	nanoxml_v2	0.71*
lang	0.51	nanoxml_v3	0.52
python	0.53	nanoxml_v5	0.73*

***p<0.001, **p<0.01 and *p<0.05

Based on the above discussion, learning the transitive relationships of a failure context significantly contributes to effective fault localization, serving as a key factor of CAN in significantly improving fault localization.

B. *How Do Other Variants Impact the Localization Effectiveness of CAN?*

Node content: There are several approaches that leverage graph structure of codes in the software engineering tasks [33], [73], [74], which consider node content as the initial representation of the nodes. Thus, we evaluate the contribution of node content to the effectiveness of CAN by using various types of representations in statements. Specifically, we use the "expression" type meaning a node contains an expression statement (e.g., arithmetic expression, assignment expression, relational expression and logical expression), the "function call" type denoting a node includes a function call statement, the "control" type representing a node includes a control statement (e.g., if, while, do while, for, switch, break, goto, continue and return), the "compound" type denoting a node includes a compound statement, the "empty" type representing a node includes an empty statement(e.g., idle loop). Table XI presents the statistical comparison of original CAN and the CAN with nodes embedding various types of node content (denoted as CAN-NodeContent) using Vargha-Delaney A-test and Wilcoxon-Signed-Rank Tests at the σ level of 0.05. CAN acquires BETTER and SIMILAR results, showing that node content contributes little to the effectiveness of CAN.

Graph-based architectures: CAN uses GNN architecture because it has a simple yet effective structure which is easy to be trained [25]. There are other representative graph-based architectures with sophisticated structures, e.g., graph attention network (GAT) [75] and gate graph attention neural network (GGANN) [76], [77]. It is interesting to see the effectiveness of other representative graph-based architectures on CAN. We use GAT (denoted CAN-GAT) and GGANN (denoted CAN-GGANN) for CAN, and compare their effectiveness with original CAN. Table XII presents the statistical results using Vargha-Delaney A-test and Wilcoxon-Signed-Rank Test at the

TABLE XII
COMPARISON OF CAN OVER CAN-GAT AND CAN-GGANN

CAN vs CAN-GAT			
Program	Result	Program	Result
chart	0.69*	space	0.71*
gzip	0.46	time	0.49
libtiff	0.52	nanoxml_v1	0.54
math	0.51	nanoxml_v2	0.72*
lang	0.47	nanoxml_v3	0.74*
python	0.53	nanoxml_v5	0.56
CAN vs CAN-GGANN			
Program	Result	Program	Result
chart	0.70*	space	0.50
gzip	0.55	time	0.48
libtiff	0.47	nanoxml_v1	0.72*
math	0.53	nanoxml_v2	0.55
lang	0.55	nanoxml_v3	0.71*
python	0.52	nanoxml_v5	0.62
***p<0.01, **p<0.1 and *p<0.5			

TABLE XIII
COMPARISON OF CAN OVER CAN-1LAYER AND CAN-3LAYERS

Program	Result	Program	Result
CAN vs CAN-1Layer			
chart	0.58	space	0.64
gzip	0.65	time	0.49
libtiff	0.66	nanoxml_v1	0.53
math	0.69*	nanoxml_v2	0.72*
lang	0.51	nanoxml_v3	0.73*
python	0.47	nanoxml_v5	0.51
CAN-3Layers vs CAN			
chart	0.52	space	0.51
gzip	0.55	time	0.68*
libtiff	0.47	nanoxml_v1	0.52
math	0.63	nanoxml_v2	0.51
lang	0.64	nanoxml_v3	0.66
python	0.52	nanoxml_v5	0.62
***p<0.001, **p<0.01 and *p<0.05			

σ level of 0.05. CAN acquires BETTER and SIMILAR results, showing that CAN has comparable effectiveness while the graph embedding module of CAN had a simpler structure and easier to be trained.

Furthermore, we compare the current CAN (i.e., 2-layers graph neural network) with different layers (i.e., CAN-1Layer and CAN-3Layers) to see whether different layers make a difference. CAN-1Layer and CAN-3Layers represent 1-layer and 3-layers graph neural network followed by a GRU, respectively. Table XIII presents the statistical results among different layers using Vargha-Delaney A-test and Wilcoxon-Signed-Rank Test at the σ level of 0.05. The results show that CAN acquires SIMILAR results over CAN-1Layer and CAN-3Layers in almost all cases.

C. *How Efficient is CAN?*

The state-of-the-art localization techniques can be roughly classified into two major types: SFL and DLFL. We discuss the efficiency of CAN over the two major types of fault localization techniques, respectively.

For SFL, its suspiciousness evaluation just analyzes the information of a statement covered or not covered, and thus incurs a very low overhead. Since CAN uses GNN to analyze a failure context including the transitive relationships, its overhead is much higher than SFL. However, without incorporating contextual information, the localization effectiveness of SFL is significantly lower than CAN.

For DLFL, since CAN focuses on a failure context, its overhead is much lower than the state-of-art DLFL techniques while showing much higher effectiveness. For training time, CAN consumes from 4.8 minutes to 53.4 minutes, and 20.7 minutes on average while the four DLFL techniques (i.e., MLP-FL, CNN-FL, BiLSTM-FL and DeepRL4FL) cost from 3.6 minutes to 19.35 hours, and 4.67 hours on average.

VI. THREATS TO VALIDITY

Threats to internal validity: Threats to internal validity relate to potential errors in our implementation. First, one potential threat to validity is the potential errors in the implementation of CAN and 12 baselines. To mitigate the threat, for the eight SFL techniques (i.e., Ochiai, ER5, GP02, GP03, Dstar, GP19, ER1, and MULTRIC), we implement them based on the widely used SFL source code GZoltar ;[11] for the four DLFL techniques (i.e., MLP-FL, CNN-FL, BiLSTM-FL, and DeepRL4FL), we reuse and enhance the source code from the previous studies [13], [14]. We also double-check the implementation and fully test our code, still there could be errors that we did not notice.

Threats to external validity: Threats to external validity relate to generalizability of our results. We adopt neural networks, whose outputs are not stable, meaning that the localization results are not the same through different training times. That drawback is caused by characteristic of deep learning technology. To make the results more reliable, we follow the conventional strategy by repeating the experiments ten times and using the average score as the experimental results. Specifically, the range of top-1 is from 4.43% to 4.78%, the range of top-3 is from 18.14% to 23.39% and the range of top-5 is from 26.67% to 30.32%.

Another threat to external validity is the subject programs used for our experiments. Our subject programs are commonly used in the field of software debugging, which are all from real-life development. However, the experimental results may not apply to all programs because there are still many unknown

[11] https://gzoltar.com/

and complicated factors in realistic debugging that could affect the experiment results. Thus, it is worthwhile to conduct the experiments on more large-sized programs to further strengthen the experimental results.

Threats to construct validity: Threats to construct validity relate to the suitability of our evaluation. One potential threat is that we adopt *Top-N Accuracy*, *MAR*, *MFR*, and *RImp* as the evaluation measures, and use Wilcoxon signed-rank test to investigate whether the improvement of our proposed approach over baselines is significant. The four evaluation metrics and the Wilcoxon signed-rank test have been widely used in many fault localization studies. According to the extensive use of the measures and the Wilcoxon signed-rank test, the threat is acceptably mitigated.

VII. Conclusion and Future Work

In this paper, we propose CAN: context-aware neural fault localization, to analyze and incorporate a failure context into suspiciousness evaluation for improving fault localization. CAN leverages program slicing to model a failure context denoted as a program dependency graph; then it constructs a graph neural network to analyze and learn the complicated relationships among the statements in the failure context; finally it uses the learned model to evaluate the suspiciousness of each statement of being faulty. We conducted the experiments on 12 real and large-sized programs and compared CAN with the 12 state-of-the-art fault localization approaches. The results show that CAN significantly improves fault localization effectiveness. e.g., the improvement for the most important metric [63], such as Top-5, as compared to the best-performing baseline is 14.31%.

In future, we plan to study the impact of test cases on the learning process and explore inter-procedural analysis to further leverage the learning ability of graph neural networks in improving fault localization. We also plan to compose our solution with other solutions proposed in the literature (e.g., multi-modal analysis [78], active learning [79]) and conduct a use study to explore the usefulness of our work in practice.

References

[1] L. Naish and Hua, "A model for spectra-based software diagnosis," *ACM Trans. Softw. Eng. Methodol.*, vol. 20, no. 3, pp. 1–32, 2011.

[2] W. E. Wong, R. Gao, Y. Li, A. Rui, and F. Wotawa, "A survey on software fault localization," *IEEE Trans. Softw. Eng.*, vol. 42, no. 8, pp. 707–740, Aug. 2016.

[3] T. D. B. Le, R. J. Oentaryo, and D. Lo, "Information retrieval and spectrum based bug localization: Better together," in *Proc. Joint Meeting Found. Softw. Eng.*, 2015, pp. 579–590.

[4] M. Zhang, X. Li, L. Zhang, and S. Khurshid, "Boosting spectrum-based fault localization using PageRank," in *Proc. 26th Int. Symp. Softw. Testing Anal.*, 2017, pp. 261–272.

[5] M. Zhang et al., "An empirical study of boosting spectrum-based fault localization via PageRank," *IEEE Trans. Softw. Eng.*, vol. 47, no. 6, pp. 1089–1113, Jun. 2021.

[6] X. Li, S. Zhu, M. d'Amorim, and A. Orso, "Enlightened debugging," in *Proc. 40th Int. Conf. Softw. Eng.*, 2018, pp. 82–92.

[7] S. Benton, X. Li, Y. Lou, and L. Zhang, "On the effectiveness of unified debugging: An extensive study on 16 program repair systems," in *Proc. IEEE/ACM 35th Int. Conf. Automated Softw. Eng.*, 2020, pp. 907–918.

[8] M. Papadakis and Y. L. Traon, *Metallaxis-FL: Mutation-Based Fault Localization*. Hoboken, NJ, USA: Wiley, 2015.

[9] S. Ma, Y. Liu, W.-C. Lee, X. Zhang, and A. Grama, "Mode: Automated neural network model debugging via state differential analysis and input selection," in *Proc. 26th ACM Joint Meeting Eur. Softw. Eng. Conf. Symp. Found. Softw. Eng.*, 2018, pp. 175–186.

[10] J. A. Jones and M. J. Harrold, "Empirical evaluation of the tarantula automatic fault-localization technique," in *Proc. IEEE/ACM 20th Int. Conf. Automated Softw. Eng.*, 2005, pp. 273–282.

[11] F. Keller, L. Grunske, S. Heiden, A. Filieri, A. van Hoorn, and D. Lo, "A critical evaluation of spectrum-based fault localization techniques on a large-scale software system," in *Proc. IEEE Int. Conf. Softw. Qual. Rel. Secur.*, 2017, pp. 114–125.

[12] S. Pearson et al., "Evaluating and improving fault localization," in *Proc. IEEE/ACM Int. Conf. Softw. Eng.*, 2017, pp. 609–620.

[13] Z. Zhang, Y. Lei, X. Mao, and P. Li, "CNN-FL: An effective approach for localizing faults using convolutional neural networks," in *Proc. IEEE 26th Int. Conf. Softw. Anal. Evol. Reengineering*, 2019, pp. 445–455.

[14] Z. Zhang, Y. Lei, X. Mao, M. Yan, L. Xu, and J. Wen, "Improving deep-learning-based fault localization with resampling," *J. Softw. Evol. Process*, vol. 33, pp. 1–22, 2020.

[15] W. E. Wong, V. Debroy, R. Golden, X. Xu, and B. Thuraisingham, "Effective software fault localization using an RBF neural network," *IEEE Trans. Rel.*, vol. 61, no. 1, pp. 149–169, Mar. 2012.

[16] Y. Li, S. Wang, and T. N. Nguyen, "Fault localization with code coverage representation learning," in *Proc. IEEE/ACM 43rd Int. Conf. Softw. Eng.*, 2021, pp. 661–673.

[17] R. Santelices, J. Jones, Y. Yu, and M. Harrold, "Lightweight fault-localization using multiple coverage types," in *Proc. IEEE 31st Int. Conf. Softw. Eng.*, 2009, pp. 56–66.

[18] B. Xu, J. Qian, X. Zhang, Z. Wu, and L. Chen, "A brief survey of program slicing," *ACM SIGSOFT Softw. Eng. Notes*, vol. 30, no. 2, pp. 1–36, 2005.

[19] Y. Zhang and R. Santelices, "Prioritized static slicing and its application to fault localization," *J. Syst. Softw.*, vol. 114, no. Apr., pp. 38–53, 2016.

[20] E. Pira, V. Rafe, and A. Nikanjam, "Searching for violation of safety and liveness properties using knowledge discovery in complex systems specified through graph transformations," *Inf. Softw. Technol.*, vol. 97, pp. 110–134, 2018.

[21] C. Sun and S. C. Khoo, "Mining succinct predicated bug signatures," in *Proc. Joint Meeting Found. Softw. Eng.*, 2013, pp. 576–586.

[22] Z. Zuo, S.-C. Khoo, and C. Sun, "Efficient predicated bug signature mining via hierarchical instrumentation," in *Proc. Int. Symp. Softw. Testing Anal.*, 2014, pp. 215–224.

[23] D. Lo, H. Cheng, and X. Wang, "Bug signature minimization and fusion," in *Proc. IEEE 13th Int. Symp. High-Assurance Syst. Eng.*, 2011, pp. 340–347.

[24] F. Scarselli, M. Gori, A. C. Tsoi, M. Hagenbuchner, and G. Monfardini, "The graph neural network model," *IEEE Trans. Neural Netw.*, vol. 20, no. 1, pp. 61–80, Jan. 2009.

[25] J. Zhou et al., "Graph neural networks: A review of methods and applications," 2018, *arXiv:1812.08434*.

[26] W. Hamilton, Z. Ying, and J. Leskovec, "Inductive representation learning on large graphs," in *Proc. Adv. Neural Inf. Process. Syst.*, 2017, pp. 1024–1034.

[27] T. N. Kipf and M. Welling, "Semi-supervised classification with graph convolutional networks," 2016, *arXiv:1609.02907*.

[28] Z. Li, X. Ding, and T. Liu, "Constructing narrative event evolutionary graph for script event prediction," 2018, *arXiv:1805.05081*.

[29] K. Marino, R. Salakhutdinov, and A. Gupta, "The more you know: Using knowledge graphs for image classification," 2016, *arXiv:1612.04844*.

[30] Z. Li et al., "VulDeePecker: A deep learning-based system for vulnerability detection," in *Proc. Netw. Distrib. Syst. Secur. Symp.*, 2018, pp. 1–15.

[31] Z. Li, D. Zou, S. Xu, H. Jin, Y. Zhu, and Z. Chen, "SySeVR: A framework for using deep learning to detect software vulnerabilities," *IEEE Trans. Dependable Secure Comput.*, vol. 19, no. 4, pp. 2244–2258, Jul./Aug. 2022.

[32] Z. Li, D. Zou, S. Xu, Z. Chen, Y. Zhu, and H. Jin, "VulDeeLocator: A deep learning-based fine-grained vulnerability detector," *IEEE Trans. Dependable Secure Comput.*, vol. 19, no. 4, pp. 2821–2837, Jul./Aug. 2022.

[33] Y. Zhou, S. Liu, J. Siow, X. Du, and Y. Liu, "Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks," in *Proc. Adv. Neural Inf. Process. Syst.*, 2019, pp. 1–11.

[34] E. Khalil, H. Dai, Y. Zhang, B. Dilkina, and L. Song, "Learning combinatorial optimization algorithms over graphs," in *Proc. Adv. Neural Inf. Process. Syst.*, 2017, pp. 6348–6358.

[35] T. Hamaguchi, H. Oiwa, M. Shimbo, and Y. Matsumoto, "Knowledge transfer for out-of-knowledge-base entities: A graph neural network approach," 2017, *arXiv:1706.05674*.

[36] H. Agrawal and J. R. Horgan, "Dynamic program slicing," *ACM SIGPlan Notices*, vol. 25, no. 6, pp. 246–256, 1990.

[37] J. Xuan and M. Monperrus, "Learning to combine multiple ranking metrics for fault localization," in *Proc. IEEE Int. Conf. Softw. Maintenance Evol.*, 2014, pp. 191–200.

[38] X. Xia, L. Bao, D. Lo, and S. Li, ""Automated debugging considered harmful" considered harmful: A user study revisiting the usefulness of spectra-based fault localization techniques with professionals using real bugs from large systems," in *Proc. IEEE Int. Conf. Softw. Maintenance Evol.*, 2016, pp. 267–278.

[39] Z. Wu, S. Pan, F. Chen, G. Long, C. Zhang, and S. Y. Philip, "A comprehensive survey on graph neural networks," *IEEE Trans. Neural Netw. Learn. Syst.*, vol. 32, no. 1, pp. 4–24, Jan. 2021.

[40] M. Wen et al., "Historical spectrum based fault localization," *IEEE Trans. Softw. Eng.*, vol. 47, no. 11, pp. 2348–2368, Nov. 2021.

[41] J. A. Jones, "Fault localization using visualization of test information," in *Proc. IEEE Int. Conf. Softw. Eng.*, 2004, pp. 54–56.

[42] A. J. C. V. G. R. Abreu and P. Zoeteweij, "An evaluation of similarity coeffcients for software fault localization," in *Proc. IEEE 12th Pacific Rim Int. Symp. Dependable Comput.*, 2006, pp. 39–46.

[43] X. Xie, T. Y. Chen, F. C. Kuo, and B. Xu, "A theoretical analysis of the risk evaluation formulas for spectrum-based fault localization," *ACM Trans. Softw. Eng. Methodol.*, vol. 22, no. 4, 2013, Art. no. 31.

[44] X. Xie, F. C. Kuo, T. Y. Chen, S. Yoo, and M. Harman, "Provably optimal and human-competitive results in SBSE for spectrum based fault localisation," in *Proc. Int. Symp. Search Based Softw. Eng.*, Berlin Heidelberg, Springer, 2013, pp. 224–238.

[45] W. Zheng, D. Hu, and J. Wang, "Fault localization analysis based on deep neural network," *Math. Problems Eng.*, vol. 2016, pp. 1–11, 2016.

[46] X. Li, W. Li, Y. Zhang, and L. Zhang, "DeepFL: Integrating multiple fault diagnosis dimensions for deep fault localization," in *Proc. 28th ACM SIGSOFT Int. Symp. Softw. Testing Anal.*, 2019, pp. 169–180.

[47] Y. Lecun, Y. Bengio, and G. E. Hinton, "Deep learning," *Nature*, vol. 521, no. 7553, 2015, Art. no. 436.

[48] S. Wang, T. Liu, and L. Tan, "Automatically learning semantic features for defect prediction," in *Proc. 38th Int. Conf. Softw. Eng.*, 2016, pp. 297–308.

[49] J. Guo, J. Cheng, and J. Cleland-Huang, "Semantically enhanced software traceability using deep learning techniques," in *Proc. IEEE 39th Int. Conf. Softw. Eng.*, 2017, pp. 3–14.

[50] L. C. Briand, Y. Labiche, and X. Liu, "Using machine learning to support debugging with tarantula," in *Proc. IEEE Int. Symp. Softw. Rel.*, 2007, pp. 137–146.

[51] W. E. Wong and Y. QI, "BP neural network-based effective fault localization," *Int. J. Softw. Eng. Knowl. Eng.*, 2009, pp. 573–597.

[52] X. Li and L. Zhang, "Transforming programs and tests in tandem for fault localization," *Proc. ACM Prog. Lang.*, vol. 1, no. OOPSLA, pp. 1–30, Oct. 2017.

[53] J. Sohn and S. Yoo, "FLUCCS: Using code and change metrics to improve fault localization," in *Proc. 26th ACM SIGSOFT Int. Symp. Softw. Testing Anal.*, 2017, pp. 273–283.

[54] D. Zou, J. Liang, Y. Xiong, M. D. Ernst, and L. Zhang, "An empirical study of fault localization families and their combinations," *IEEE Trans. Softw. Eng.*, vol. 47, no. 2, pp. 332–347, Feb. 2021.

[55] Lucia, D. Lo, and X. Xia, "Fusion fault localizers," in *Proc. IEEE/ACM 29th Int. Conf. Automated Softw. Eng.*, 2014, pp. 127–138.

[56] Y. Lou et al., "Can automated program repair refine fault localization? A unified debugging approach," in *Proc. 29th ACM SIGSOFT Int. Symp. Softw. Testing Anal.*, 2020, pp. 75–87.

[57] H. Hsu, J. A. Jones, and A. Orso, "Rapid: Identifying bug signatures to support debugging activities," in *Proc. IEEE/ACM 23rd Int. Conf. Automated Softw. Eng.*, 2008, pp. 439–442.

[58] H. Cheng, D. Lo, Y. Zhou, X. Wang, and X. Yan, "Identifying bug signatures using discriminative graph mining," in *Proc. 18th Int. Symp. Softw. Testing Anal.*, New York, NY, USA, 2009, pp. 141–152.

[59] Z. Zhang, Y. Lei, X. Mao, M. Yan, L. Xu, and J. Wen, "Improving deep-learning-based fault localization with resampling," *J. Softw., Evol. Process*, vol. 33, no. 3, 2021, Art. no. e2312.

[60] H. Xie, Y. Lei, M. Yan, Y. Yu, X. Xia, and X. Mao, "A universal data augmentation approach for fault localization," in *Proc. 44th Int. Conf. Softw. Eng.*, 2022, pp. 48–60.

[61] D. Jalali and M. D. Ernst, "Defects4J: A database of existing faults to enable controlled testing studies for Java programs," in *Proc. Int. Symp. Softw. Testing Anal.*, 2014, pp. 437–440.

[62] Z. Zhang, Y. Lei, X. Mao, M. Yan, L. Xu, and X. Zhang, "A study of effectiveness of deep learning in locating real faults," *Inf. Softw. Technol.*, vol. 131, 2021, Art. no. 106486.

[63] P. S. Kochhar, X. Xia, D. Lo, and S. Li, "Practitioners' expectations on automated fault localization," in *Proc. 25th Int. Symp. Softw. Testing Anal.*, 2016, pp. 165–176.

[64] C. Parnin and A. Orso, "Are automated debugging techniques actually helping programmers?," in *Proc. Int. Symp. Softw. Testing Anal.*, 2011, pp. 199–209.

[65] V. Debroy, W. E. Wong, X. Xu, and B. Choi, "A grouping-based strategy to improve the effectiveness of fault localization techniques," in *Proc. IEEE Int. Conf. Qual. Softw.*, 2010, pp. 13–22.

[66] Y. Lei, X. Mao, Z. Dai, and C. Wang, "Effective statistical fault localization using program slices," in *Proc. IEEE Comput. Softw. Appl. Conf.*, 2012, pp. 1–10.

[67] K. Liu, A. Koyuncu, D. Kim, and T. F. Bissyandé, "TBar: Revisiting template-based automated program repair," in *Proc. 28th ACM SIGSOFT Int. Symp. Softw. Testing Anal.*, 2019, pp. 31–42.

[68] J. Jiang, Y. Xiong, H. Zhang, Q. Gao, and X. Chen, "Shaping program repair space with existing patches and similar code," in *Proc. 27th ACM SIGSOFT Int. Symp. Softw. Testing Anal.*, 2018, pp. 298–309.

[69] G. W. Corder and D. I. Foreman, *Nonparametric Statistics for Non-Statisticians: A Step-by-Step Approach*, vol. 78. Hoboken, NJ, USA: Wiley, 2010.

[70] H. Abdi, "The Bonferonni and Šidák corrections for multiple comparisons," *Encyclopedia Meas. Statist.*, vol. 3, pp. 103–107, 2007.

[71] A. Arcuri and L. Briand, "A practical guide for using statistical tests to assess randomized algorithms in software engineering," in *Proc. 33rd Int. Conf. Softw. Eng.*, 2011, pp. 1–10.

[72] A. Vargha and H. D. Delaney, "A critique and improvement of the CL common language effect size statistics of McGraw and Wong," *J. Educ. Behav. Statist.*, vol. 25, no. 2, pp. 101–132, 2000.

[73] S. Chakraborty, R. Krishna, Y. Ding, and B. Ray, "Deep learning based vulnerability detection: Are we there yet," *IEEE Trans. Softw. Eng.*, vol. 48, no. 9, pp. 3280–3296, Sep. 2022.

[74] Y. Li, S. Wang, and T. N. Nguyen, "Vulnerability detection with fine-grained interpretations," in *Proc. 29th ACM Joint Meeting Eur. Softw. Eng. Conf. Symp. Found. Softw. Eng.*, 2021, pp. 292–303.

[75] P. Velickovic, G. Cucurull, A. Casanova, and A. Romero, "Graph attention networks," in *Proc. Int. Conf. Learn. Representations*, 2018, pp. 1–16.

[76] Y. Li, D. Tarlow, M. Brockschmidt, and R. Zemel, "Gated graph sequence neural networks," in *Proc. 4th Int. Conf. Learn. Representations*, 2016, pp. 1–20.

[77] J. Wu, J. Xu, X. Meng, H. Zhang, and Z. Zhang, "Enabling reliability-driven optimization selection with gate graph attention neural network," *Int. J. Softw. Eng. Knowl. Eng.*, vol. 30, no. 11n12, pp. 1641–1665, 2020.

[78] T. Hoang, R. J. Oentaryo, T.-D. B. Le, and D. Lo, "Network-clustered multi-modal bug localization," *IEEE Trans. Softw. Eng.*, vol. 45, no. 10, pp. 1002–1023, Oct. 2019.

[79] L. Gong, D. Lo, L. Jiang, and H. Zhang, "Interactive fault localization leveraging simple user feedback," in *Proc. IEEE 28th Int. Conf. Softw. Maintenance*, 2012, pp. 67–76.