Neural SZZ Algorithm

Lingxiao Tang, Lingfeng Bao¶, Xin Xia¶ and Zhongdong Huang
College of Computer Science and Technology, Zhejiang University, China
{tlx,lingfengbao,hzd}@zju.edu.cn, xin.xia@acm.org

***Abstract*—The SZZ algorithm has been widely used for identifying bug-inducing commits. However, it suffers from low precision, as not all deletion lines in the bug-fixing commit are related to the bug fix. Previous studies have attempted to address this issue by using static methods to filter out noise, e.g., comments and refactoring operations in the bug-fixing commit. However, these methods have two limitations. First, it is challenging to include all refactoring and non-essential change patterns in a tool, leading to the potential exclusion of relevant lines and the inclusion of irrelevant lines. Second, applying these tools might not always improve performance.**

In this paper, to address the aforementioned challenges, we propose NEURALSZZ, a deep learning approach for detecting the root cause deletion lines in a bug-fixing commit and using them as input for the SZZ algorithm. NEURALSZZ first constructs a heterogeneous graph attention network model that captures the semantic relationships between each deletion line and the other deletion and addition lines. To pinpoint the root cause of a bug, NEURALSZZ uses a learning-to-rank technique to rank all deletion lines in the commit. To evaluate the effectiveness of NEURALSZZ, we utilize three datasets containing high-quality bug-fixing and bug-inducing commits. The experiment results show that NEURALSZZ outperforms various baseline methods, e.g., traditional machine learning-based approaches and Bi-LSTM in identifying the root cause of bugs. Moreover, by utilizing the top-ranked deletion lines and applying the SZZ algorithm, NEURALSZZ demonstrates better precision and F1-score compared to previous SZZ algorithms.

***Index Terms*—SZZ Algorithm, Deep Learning, Heterogeneous Graph Attention Network, Learning to Rank**

I. INTRODUCTION

Modern software development relies heavily on version control systems (VCSs) to manage source code, track changes, and facilitate the collaboration among developers. As one of the most popular VCSs used today, Git is especially well-suited for distributed development scenario. A commit in Git is a fundamental unit of change that represents a snapshot of the repository at a particular point in time. However, some commits may introduce bugs, and these are known as bug-inducing commits [1], [2]. These commits contain crucial information about how bugs are introduced in software development, and as a result, have raised increasing research attention. Many studies have been conducted to identify the features of bug-inducing commits [3]–[6], make use of these commits to enable Just-In-Time (JIT) defect detection [1], [7]–[9] and determine the affected software versions of a vulnerability [10].

¶ Corresponding authors.

The SZZ [11] algorithm is a primary approach for identifying bug-inducing commits in projects. Given a bug-fixing commit, the SZZ algorithm identifies the bug-inducing commits by examining the previous commits and finds the one that made the last change to the deletion lines in the bug-fixing commit. However, the SZZ algorithm is known to have low precision because of the existing noise in bug-fixing commits. To solve the problem, many variants of the SZZ algorithm have been proposed [12]–[14]. Most of the variants use static methods to improve the precision of the SZZ algorithm. They attempt to filter out non-essential changes in the bug-fixing commit, such as comments and refactoring operations.

Despite the advancements made by the existing works, there are still limitations. One such limitation is the difficulty of integrating all types of refactoring operations into a single tool. RA-SZZ algorithm [14] integrates the RefDiff tool [15] that can only detect 13 types of refactoring operations, leading to the potential inclusion of irrelevant lines [16]. A newer tool proposed by Tsantalis et al. [17] supports 15 types of refactoring operations. However, it is still far from the number of refactoring types (i.e., 65 in total) identified by Fowler [18]. Moreover, these tools may also exclude relevant lines [16]. For instance, RefDiff [15] can not pinpoint the line numbers associated with refactoring operations, particularly in method bodies, which means that it may mark some relevant lines as refactoring operations by mistake. Thus, applying these refactoring detection tools may not improve precision much, and in some cases, may even worsen the performance. For instance, the original SZZ algorithm outperforms RA-SZZ on the dataset of Rosa et al. [19] in terms of F1-score (0.50 vs. 0.39).

Therefore, the goal of our study is to provide a new way to improve the precision of the SZZ algorithm. In this study, we propose a deep learning based approach named NEURALSZZ that prioritizes all deletion lines in a bug-fixing commit based on their likelihood of being the root cause of the bug. To achieve this, NEURALSZZ makes use of crucial factors that have not been utilized in prior research, i.e., semantic meanings of the deletion statements and their relationships with other statements. Specifically, it constructs a heterogeneous graph for each bug-fixing commit, where the nodes represent the deletion and addition statements, and the edges represent the relationships between them. NEURALSZZ extracts edges based on various types of graphs, including control flow graph and data dependency graph. We use a Heterogeneous Graph Attention Network (HAN) [20] model to generate embeddings for each statement in the graph. Finally, we train a pairwise rank model to rank all deletion statements in the graph based on their embeddings.

To train and evaluate our model, we first combine three high-quality datasets collected by Wen et. al [21], Song et. al [22], and Neto et. al [16] into one. For each bug in the merged dataset, we manually annotate the root causes based on the given bug-fixing and inducing commit. Then, we evaluate our proposed approach by answering the following research questions:

RQ1: How effective is NEURALSZZ compared to baselines for identifying the root cause in the bug-fixing commits?

We perform a 10-fold cross-validation to evaluate the effectiveness of NEURALSZZ. Various baseline methods are included in comparison, including five machine learning based approaches (e.g., RF, LR, SVM, XGB and KNN) and Bi-LSTM. The result shows that our approach can identify the root cause more effectively than baselines, improving the best baseline by 8.5% in recall at the top 1.

RQ2: How effective is NEURALSZZ in the cross-project setting?

In order to verify the generalizability of NEURALSZZ, we conducted additional evaluations on its performance under cross-project scenario. For this purpose, we employed the developer-informed oracle provided by Wen et. al [21] as the test set, while the other two were utilized as training sets. Our findings indicate that NEURALSZZ also outperforms the baselines under cross-project scenario.

RQ3: How effective is NEURALSZZ compared to previous SZZ algorithms in detecting the bug-inducing commits?

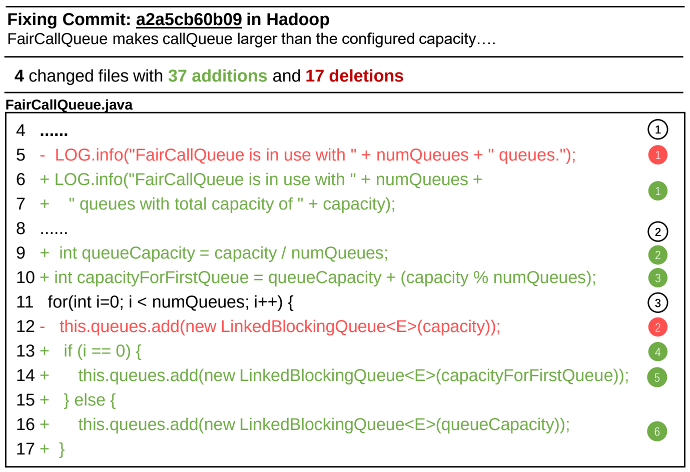
Our goal is to explore whether utilizing the deletion lines ranked by NEURALSZZ, which are supposed to be the root cause, can enhance the SZZ algorithms. We apply the original SZZ algorithm by inputting the top 1, 2, and 3 deletion lines ranked by NEURALSZZ to identify bug-inducing commits. Compared to the best baseline, NEURALSZZ can enhance the F1-score by 40.7%, 36.7%, and 36.2% for the top 1, top 2, and top 3 deletion lines, respectively.

RQ4: How effective are the key components of NEURALSZZ?

We also perform an ablation experiment to verify the effectiveness of the key designs. The result shows that both CodeBERT and HAN have improved the performance of NEURALSZZ.

In summary, we have the following contributions:

- To the best of our knowledge, we are the first to leverage the semantic meanings of deletion lines and their relationships with other modified lines to improve the precision of the SZZ algorithm.
- We collect a dataset containing 675 bug-fixing commits and their corresponding bug-inducing commits. We manually annotate the root causes of bug-fixing commits based on the given bug-inducing commits. We provide a replication package to foster future work, in line with good research practices [23].



Fixing Commit: a2a5cb60b09 in Hadoop
FairCallQueue makes callQueue larger than the configured capacity….

4 changed files with **37 additions** and **17 deletions**

FairCallQueue.java

```
4   ......                                                              ①
5   -  LOG.info("FairCallQueue is in use with " + numQueues + " queues.");   1
6   + LOG.info("FairCallQueue is in use with " + numQueues +
7   +     " queues with total capacity of " + capacity);                1
8   ......                                                              ②
9   +  int queueCapacity = capacity / numQueues;                         2
10  + int capacityForFirstQueue = queueCapacity + (capacity % numQueues);   3
11    for(int i=0; i < numQueues; i++) {                                ③
12  -   this.queues.add(new LinkedBlockingQueue<E>(capacity));          2
13  +   if (i == 0) {                                                   4
14  +       this.queues.add(new LinkedBlockingQueue<E>(capacityForFirstQueue));   5
15  +   } else {
16  +       this.queues.add(new LinkedBlockingQueue<E>(queueCapacity));   6
17  +   }
```

Fig. 1: A Motivation Example

- Experimental results show that our approach can rank root cause deletion lines and identify bug-inducing commits more effectively.

II. BACKGROUND

In this section, we first introduce the SZZ algorithm and its variants. Then we present our motivation example.

A. SZZ Algorithms

B-SZZ. The original SZZ algorithm (B-SZZ) was proposed by Sliwerski et al [11]. It tracks back to the last changes that introduced the deleted/modified lines in the bug-fixing commit, marking them as bug-inducing commits. To achieve this, it uses the annotate function provided by the version control system.

AG-SZZ. Kim et al. proposed AG-SZZ [12]. They improved B-SZZ by using an annotation graph to filter out blank lines, flagging comments, and cosmetic changes in bug-fixing commits. The annotation graph can provide more comprehensive information about line changes and movement than the annotate function.

MA-SZZ. Da Costa et al. proposed MA-SZZ [13], which filters out meta-changes from potential bug-inducing changes. Meta-changes include branch changes, merge changes, and property changes, each of which does not change the source code.

RA-SZZ. Neto et al. [14] noticed that previous SZZ algorithms incorrectly identify bug-inducing changes due to the impact of refactoring operations. To address this, they integrated the refactoring detection tools RefDiff and Refactoring Miner into the RA-SZZ algorithm they proposed.

B. Motivation Example

Figure 1 presents a bug-fixing commit in Hadoop [24]. Four files have been modified in this commit, resulting in 37 additions and 17 deletions in total. Due to the page limitation, we only show the most significant part of the patch. According to the commit message, the root cause of the bug corresponds to `FairCallQueue.java`, Line 12.

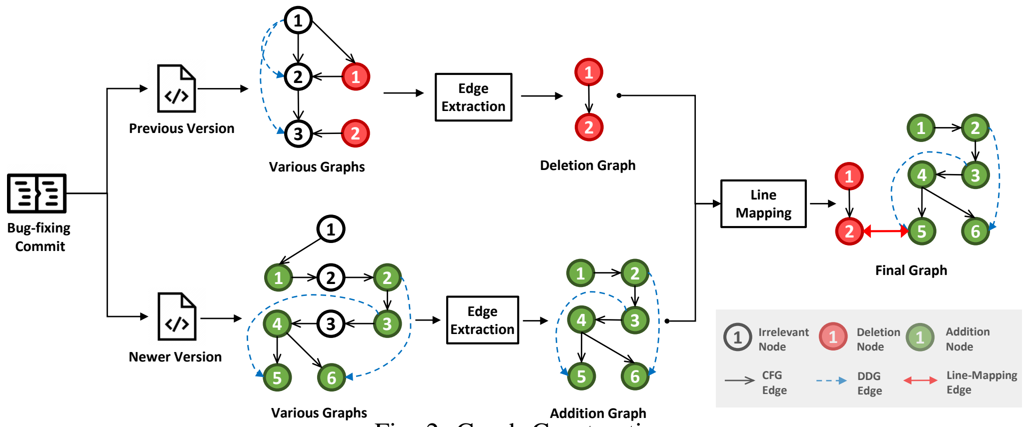


Fig. 2: Graph Construction

The changes in Line 13-17 ensure that the capacity of the `FairCallQueue` is distributed equally among all the sub-queues, with the first sub-queue having an excess capacity. The original implementation (Line 12) simply used the `capacity` variable as the capacity of each sub-queue, which could result in the total capacity of the `FairCallQueue` being larger than the configured capacity, thus causing the bug.

Running the original SZZ algorithms directly on all the modifications (17 deletion lines) in this bug-fixing commit could introduce a considerable amount of noise. This might result in inaccurate or misleading findings. Furthermore, it's important to note that these changes are not refactors. Thus, the other SZZ algorithms (e.g., RA-SZZ) can not remove them. As shown in Figure 1, there are 17 deletion lines. But many of them are irrelevant to the bug, such as the changes that are related to the `Log` function and those in the test files. Therefore, **if we consider the semantic meanings of changed lines, we can identify that those lines are related to logs and tests and remove this type of noise.**

Furthermore, we observe that **relations between changed lines in the bug-fixing commit can help identify the root cause.** In the motivation example, it may not be immediately clear what causes the bug if we only look at the deletion of Line 12. However, taking into account the relationship between Line 12 and Line 13-16 enables us to gain more insight. By doing so, we can see that the previous version of the code does not account for different scenarios when adding an item to the queue. This leads to the addition of an `if` statement in the new version. Therefore, we can conclude that deleting Line 12 has the highest probability of being the root cause of the bug.

III. Approach

Inspired by the above observations, we propose a new approach named NeuralSZZ to effectively detect the root cause among all deletion lines. Our approach leverages a heterogeneous graph attention network to make full use of the semantic meanings of changed lines as well as the relations between them in a bug-fixing commit.

Our approach consists of three steps: ❶ **Graph Construction:** Given a bug-fixing commit, we first extract nodes and edges by analyzing its changed lines to construct a graph. ❷ **Heterogeneous Graph Attention Network (HAN):** We train a HAN that allows each node to update its embedding based on its meta-path-based neighbors and different types of meta-paths in the graph. ❸ **Ranking Deletion Nodes (Lines):** A rank model is trained to rank all deletion lines in a pair-wise manner, aiming to identify the one that is most likely to be the root cause among all deletion lines.

A. Graph Construction

Figure 2 presents the process of graph construction. The node IDs in this figure correspond to the IDs in Figure 1. Given a bug-fixing commit, we use the following steps to construct a graph:

Node Extraction: We first extract the Java source code of the previous and newer versions in the bug-fixing commit. Then, we use the *JavaParser* tool [25] to build an Abstract Syntax Tree (AST) for the previous and newer files (i.e., AST_{pre} and AST_{new}), respectively. We map the deletion lines in the bug-fixing commit into a node of AST_{pre} and mark it as a deletion node. Similarly, we extract addition nodes by mapping the addition lines in the bug-fixing commit into the nodes of AST_{new}.

Edge Extraction: Based on the extracted nodes, we build edges by considering different relationships among them based on various graphs, including control flow graphs (CFG) [26], data dependency graphs (DDG) [27], call graphs (CG) [28], and class member reference graphs (CMFG). We first use the static analysis tool *Joern* [29] to build these graphs for the previous and newer versions of source code in the bug-fixing commit, respectively. For both deletion and addition nodes, we use a depth-first-search algorithm to search for paths in each type of graph. If there exists a path between two nodes in one type of graph, we add an edge whose type depends on the type of graph (e.g., CFG or DDG edge). Thus, we construct one graph for the previous version (G_{pre}) and one for the newer version (G_{new}), respectively. Furthermore, we merge the two graphs by mapping the deletion and addition nodes with a line mapping algorithm. If lines in a deletion node can be mapped to lines in another addition node, we add a line mapping edge between them. We implement the line mapping algorithm using the Abstract Syntax Tree mapping tool *GumTree* [30].

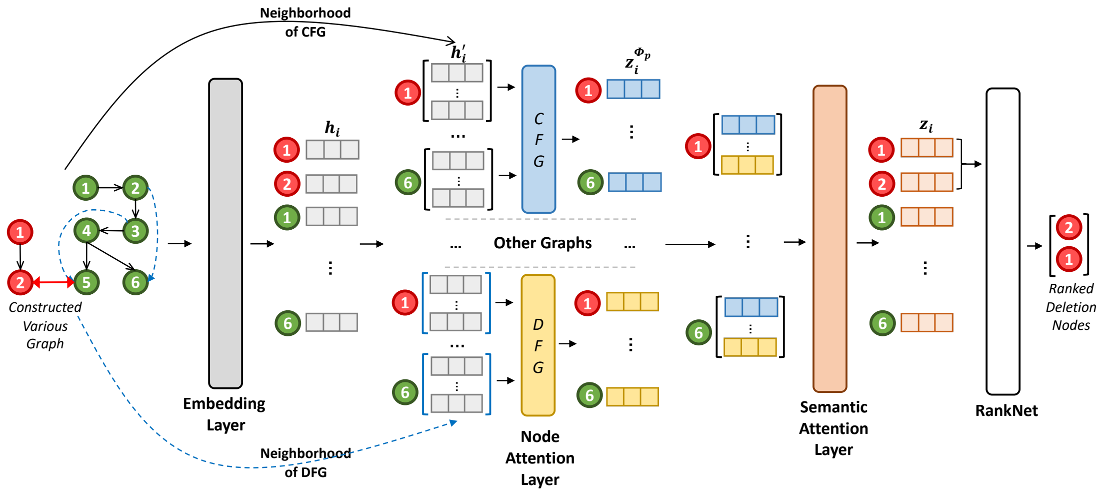


Fig. 3: Overview of NeuralSZZ

Figure 2 presents the detailed graph construction process of our motivation example (see Figure 1). We begin by extracting nodes from the changed lines in the bug-fixing commit. For example, the addition lines 6 and 7 (they form one statement) correspond to addition node 1 in G_{new}, while the deletion line 3 corresponds to deletion node 1 in G_{pre}. Then, we employ the depth-first-search algorithm to extract edges from the original graph produced by *Joern*. For instance, we determine that addition node 1 can establish a path to addition node 2 without passing through any other addition node in the CFG produced by *Joern*. As a result, we add a control-flow edge between them. We repeat this process for each type of graph to retrieve all edges between any pair of relevant nodes. Finally, we identify that deletion node 2 can be mapped to addition node 5, thus we add a line mapping edge between them to obtain the final graph.

B. Heterogeneous Graph Attention Network

Traditional machine learning and graph neural networks [31]–[33] often fail to effectively leverage the diverse types of nodes and edges presented in a heterogeneous graph [34], leading to the loss of diverse information embedded in the heterogeneous graph. Nevertheless, Heterogeneous Graph Attention Network (HAN) [20] overcomes this limitation by incorporating the semantic-level attention that captures the significance of various meta-paths. Figure 3 presents the detailed architecture of HAN.

Node Embedding Layer. Each node in the heterogeneous graph corresponds to one statement. We first utilize CodeBERT [35] as the node embedding layer to embed statements into fixed-length vectors. CodeBERT is a widely adopted pre-trained language model that has demonstrated top performance in various code-related tasks [35]. It can capture the semantic meaning of code statements and provide rich node representations that are suitable for our graph neural network. For each node i, we leverage CodeBERT to get its corresponding embedding h_i.

HAN Layer. The Heterogeneous Graph Attention Network (HAN) layer is applied after the node embedding step. It begins by performing node-level attention, where for each type of meta-path (e.g., control-flow edge, data-flow edge, call edge), node i learns embeddings from its meta-path neighbors. This process can be described as:

$$h_i^{'} = M_{\phi i} \cdot h_i \tag{1}$$

$$\alpha_{ij}^{\Phi} = \frac{exp(\sigma(a_{\Phi}^T \cdot [h_i^{'} \,\|\, h_j^{'}]))}{\sum_{k\in N_i^{\Phi}} exp(\sigma(a_{\Phi}^T \cdot [h_i^{'} \,\|\, h_k^{'}]))} \tag{2}$$

$$z_i^{\Phi} = \Big\|_{k=1}^{K} \sigma(\sum_{j\in N_i^{\Phi}} \alpha_{ij}^{\Phi} h_j^{'}) \tag{3}$$

In Equation 1, $M_{\phi j}$ is a transformation matrix that projects the node features into a predefined feature space. Equation 2 computes α_{ij}^{Φ}, which denotes the attention weight that reflects the importance of node j to node i. To compute the attention weight, we concatenate the projected embedding $h_i^{'}$ of node i with its neighbor's embedding $h_j^{'}$, and further multiply the concatenated embedding $[h_i^{'} \,\|\, h_j^{'}]$ with a vector a_{Φ} to get the initial weight. The a_{Φ} is optimized during model training. Then, we normalize all weights to get the final weight α_{ij}^{Φ} of node j to node i. Based on the calculated node attention, we further calculate the updated node embedding z_i^{Φ} by merging the embeddings of its neighbors N_i^{Φ} regarding meta-path Φ (see Equation 3). To address the high variance problem in graph data, HAN extends the node-level attention to multi-head attention [20], which repeats the node-level attention K times and concatenates all the calculated embeddings.

Then, HAN calculates the semantic-level attention to learn the importance of each meta-path. Given the meta-path set $\{\Phi_1, \Phi_2, ..., \Phi_p\}$, the importance of each meta-path is calculated as:

$$w_{\Phi_p} = \frac{1}{V}\sum_{i\in V} q^T \cdot tanh(W \cdot z_i^{\Phi_p} + b) \tag{4}$$

where w_{Φ_p} denotes the attention weight of meta-path p, while W, b, and q are learnable parameters.

Finally, HAN combines the node embeddings under various meta-paths $\left\{z_i^{\Phi_1}, z_i^{\Phi_2}, ..., z_i^{\Phi_n}\right\}$ to get the final node embedding z_i:

$$z_i = \sum_{p=1}^{P} w_{\Phi_p} \cdot z_i^{\Phi_p} \tag{5}$$

C. Ranking Deletion Nodes

After obtaining the embeddings of each node, we further train a model for ranking the deletion nodes. Specifically, we leverage the RankNet model [36], a *pairwise* based ranking method, due to its effectiveness in real-world ranking problems [37]–[39].

The RankNet model is trained by learning the relative priorities of deletion nodes in a pairwise manner. Specifically, for each training pair of nodes $\langle n_i, n_j \rangle$, the RankNet first assigns a score to each node, denoted as s_i and s_j, respectively. Then, the learned probability that n_i ranks higher than n_j is calculated as: $P_{ij} = \frac{1}{1+e^{-(s_i - s_j)}}$. The ground truth probability of the relative priority within the node pair is defined as:

$$\overline{P}_{ij} = \begin{cases} 1 & n_i\, is\, root\, cause\, node\, and\, n_j\, is\, not \\ 0 & n_j\, is\, root\, cause\, node\, and\, n_i\, is\, not \\ 0.5 & otherwise \end{cases} \tag{6}$$

Finally, the RankNet model is trained with a cross-entropy loss defined as:

$$L = -\overline{P}_{ij} log P_{ij} - (1 - \overline{P}_{ij}) log(1 - P_{ij}) \tag{7}$$

During inference, the trained RankNet model assigns a score to each deletion node, which is directly utilized to determine the overall priority of the deletion nodes.

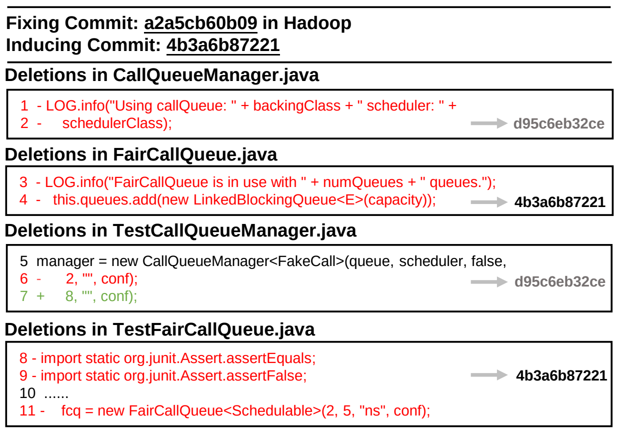
IV. Data Preparation

In this section, we first describe the dataset construction process of bug-fixing and bug-inducing commits. Then, we show the process of manual annotation for the root cause. Finally, we present the annotation results.

A. Dataset

To train our model and evaluate the performance of our method, we require a high-quality dataset containing both bug-fixing and bug-inducing commits. Since there are many noises in the dataset of bug-introducing commits generated by SZZ algorithms [21], we do not use these datasets in our study. Previous studies have constructed datasets that contain bug-fixing commits and the corresponding reliable bug-introducing commits by manual verification based on bug reports [21] or utilizing test cases [22]. However, these datasets often have a limited number of bug-fixing commits. Therefore, we combine the following three reliable datasets to construct a comprehensive dataset:

Dataset1 was collected by Wen et. al [21]. They search the information of bug-inducing commits in the bug reports and manually verified all the candidates to ensure the quality of the data.



Fixing Commit: a2a5cb60b09 in Hadoop
Inducing Commit: 4b3a6b87221

Deletions in CallQueueManager.java

```
1  - LOG.info("Using callQueue: " + backingClass + " scheduler: " +
2  -     schedulerClass);                                        → d95c6eb32ce
```

Deletions in FairCallQueue.java

```
3  - LOG.info("FairCallQueue is in use with " + numQueues + " queues.");
4  -   this.queues.add(new LinkedBlockingQueue<E>(capacity));     → 4b3a6b87221
```

Deletions in TestCallQueueManager.java

```
5  manager = new CallQueueManager<FakeCall>(queue, scheduler, false,
6  -      2, "", conf);                                          → d95c6eb32ce
7  +      8, "", conf);
```

Deletions in TestFairCallQueue.java

```
8 - import static org.junit.Assert.assertEquals;
9 - import static org.junit.Assert.assertFalse;                  → 4b3a6b87221
10  ......
11 -    fcq = new FairCallQueue<Schedulable>(2, 5, "ns", conf);
```

Fig. 4: The Annotation Process for the Motivation Example

Dataset2 was built by Song et. al [22]. They utilized tests in the code repository to pinpoint bug-inducing and bug-fixing commits. In particular, a commit is labeled as bug-inducing if a test fails for that commit but succeeds for the preceding commit. Conversely, if the same test passes for a commit following the bug-inducing commit, it is identified as the bug-fixing commit associated with the bug-inducing commit. This approach leverages tests as a reliable indicator of bug inducing and fixing commits.

Dataset3 was gathered by Neto et al [16]. They use the detailed information provided by the Defects4J dataset [40], including documents of changes in the version control system and patches with the exact changes to re-introduce the bug. After carefully analyzing the information, they isolate the truly bug-fix modifications from those that were not intended to fix the bug and identify the bug-inducing commits.

Table I presents the summary of the statistics for the three datasets, including the number of bug-fixing commits and bug-inducing commits in each dataset. It is important that some bug-fixing commits in the original dataset are not found on GitHub, and we remove them from further analysis. Additionally, we also report the count of deletion lines in the patches. If a patch has fewer than five deletion lines, we categorize it as a small patch (SMALL). Otherwise, we classify it as a large patch (LARGE). This information helps us to understand the size and characteristics of the patches in our dataset for further analysis.

B. Manual Annotation for Root Cause

Based on the three reliable datasets containing bug-fixing commits and bug-inducing commits, we further determine which deletion lines in the fixing commits are the root cause of the bugs. First, we use the `git blame` tool to identify the commits that last modified each deletion line in the bug-fixing commits. For each deletion line in a bug-fixing commit, if the inducing commits are absent from the identified commits, we consider it as not the root cause for the bug. If all identified commits do not contain the bug-inducing commits, we exclude the bug from our dataset. On the other hand, if the identified commits include the inducing commits, we consider it a potential root cause of the bug. Second, we manually review all the candidate lines to identify the

TABLE I: The statistics of the bugs and corresponding bug fixing commits in three datasets

Dataset	Project	#Bug-Fixing	#Bug-Inducing	#SMALL	#LARGE
DATASET1	accumulo	35	55	20	15
	ambari	38	44	17	21
	hadoop	53	57	28	25
	lucene	70	145	41	29
	oozie	45	50	23	22
	Total	241	351	129	112
DATASET2	jsoup	63	63	35	28
	fastjson	222	222	144	78
	verdict	53	53	11	42
	closure-templates	32	32	7	25
	twilio-java	39	39	14	25
	...(120 more projects)	548	548	328	220
	Total	957	957	539	418
DATASET3	mockito	32	53	13	19
	joda-time	23	27	12	11
	commons-math	85	111	44	41
	commons-lang	53	65	36	16
	closure-compiler	98	122	61	37
	Total	291	378	166	124

TABLE II: Annotation results

Dataset	#Bugs	#Bug-inducing	#Nodes	#Edges
DATASET1	157	219	2,677	5,283
DATASET2	284	284	5,659	12,965
DATASET3	234	316	2,186	4,398
Total	675	819	10,522	22,646

final lines that represent the actual root cause of the bugs. Figure 4 presents an example of the annotation process. We mark all commits that last modified deletion lines in the figure. Since commits in `CallQueueManager.java` and `TestCallQueueManager.java` do not contain the bug-inducing commit, we simply drop them out. Moreover, deletion lines in `TestFairCallQueue.java` are all related to test, and Line 3 in `FairQueue.java` is a log function and unrelated to the commit message, thus we also exclude them. So we can find that the root cause lies in Line 3 in `FairQueue.java`. For each deletion node in the graph, if it contains lines that are the root cause of the bugs, it is marked as a root cause node. Otherwise, it is marked as not a root cause node.

C. Annotation Result

After the filtering in annotation, the statistics of our final dataset are presented in Table II. This dataset comprises data from 87 open-source projects on GitHub and includes a total of 675 bug-fixing commits. The table provides information on the number of bug-inducing commits in each dataset, as well as the total number of nodes and edges in the patch graph. Specifically, the final dataset consists of 10,522 nodes and 22,646 edges.

V. EXPERIMENT SETUP

A. Experiment Setting

The experimental environment consisted of a server equipped with an NVIDIA GTX 3090 GPU, Intel Xeon 6226R CPU, running on Ubuntu OS. For graph learning, we utilize the pre-trained CodeBERT model [35] from the Hugging Face library following [41] and use the default HAN implementation [20] from PyTorch. For node ranking, we implement the RankNet algorithm on PyTorch, which is the same as [42].

In our study, we select three reliable datasets containing bug-fixing and bug-inducing commits. We divide the whole dataset into ten parts and perform 10-fold cross-validation [43] to verify the effectiveness of our approach. In each run of experiments, we use bugs in one part as the test set and those in the other nine parts as the training set. We report the average values of each metric in the experimental results. We also do the cross-project prediction to ensure the generalizability of our model.

Because we use a pair-wise ranking model, we need to group all deletion nodes in each commit as pairs. Since the number of training pairs is exponential to the number of deleted lines in the commit, we extract up to 100 pairs from each commit. We do this to prevent large commits from producing too many training pairs. By extracting a limited number of pairs from each commit, we obtain a total of 17,027 pairs on average to train the model.

B. Baselines

We compare our approach with the following baselines in identifying the root cause by ranking deletion lines in bug-fixing commits:

- **Machine Learning (ML) based baselines:** We select several ML algorithms, including Random Forest (RF), Linear Regression (LR), Support Vector Machine (SVM), XGBoost (XGB), and K-Nearest Neighbor (KNN). These approaches are commonly used in recent research related to code and text processing, such as security bug prediction [44] and commit-level software vulnerability assessment [45]. For these ML-based baselines, we constructed features using the

TABLE III: The performance comparisons between our approach and baselines in ranking deletion lines

Approach	Recall@1	Recall@2	Recall@3	MFR
RF	0.694	0.811	0.882	3.295
LR	0.701	0.813	0.872	3.541
SVM	0.714	0.806	0.869	3.215
XGB	0.718	0.811	0.867	3.133
KNN	0.677	0.792	0.860	2.773
Bi-LSTM	0.656	0.746	0.820	3.448
NEURALSZZ	**0.779**	**0.841**	**0.886**	**2.425**

- bag-of-words approach and limited the vocabulary size to 10K following [45], [46].
- **Bi-LSTM** [47]: We select it as a deep learning based baseline, which has been also used in previous studies, such as security patch identification [44]. We set the vocabulary size to 10K and used pre-trained 300-dimensional Glove [48] word embeddings. [44], [45].

The classifiers compute the probability of each deletion line in a bug-fixing commit being the root cause. We rank all deletion lines in a bug-fixing commit based on their probabilities.

To investigate whether our approach can improve the precision of the SZZ algorithm, we apply the B-SZZ algorithm on deletion lines on top of the ranking list. We compare the result with other SZZ algorithms. We select the SZZ algorithms in Section II-A, i.e., SZZ, AG-SZZ, MA-SZZ, and RA-SZZ. The implementation of all these SZZ algorithms is from the replication package provided by Rosa et al [19].

C. Evaluation Metrics

To evaluate the performance of identifying the root cause in bug-fixing commits, we use the widely-used measurements as follows:

Recall@N. The Top-N metric computes the number of bug-fixing commits that have at least one root cause deletion line localized within the top N positions in the ranked list. Previous studies have shown that developers tend to pay attention to a small number of elements at the top of the ranked list [49]. Therefore, we limit the value of N to 1, 2, and 3.

Mean First Rank (MFR). For all deletion lines in a commit, the first rank means the ranking of the first root cause deletion line in the list. MFR calculates the mean of first ranks for all bug-fixing patches.

To evaluate the effectiveness of our approach in identifying bug-inducing commits, we employ two metrics used in the previous studies [10], [19], i.e., *precision* and *recall*. We compare our approach with traditional SZZ algorithms by selecting the top k deletion lines from the ranking list and applying the B-SZZ algorithm to them to find the bug-inducing commits. We calculate the recall metric using the bug-inducing commits found by the top k deletion lines, denoted as $Recall_k$. Similarly, we obtain the $Precision_k$ and $F1-score_k$ metrics. We limit the value of k to 1, 2, and 3.

VI. Experiment Results

In this section, we present the experiment results of the four research questions.

TABLE IV: The performance comparisons between our approach and baselines in cross-project prediction

Approach	Recall@1	Recall@2	Recall@3	MFR
RF	0.697	0.783	0.834	2.866
LR	0.643	0.834	0.859	2.528
SVM	0.681	0.789	0.866	2.630
XGB	0.675	0.802	0.853	2.318
KNN	0.675	0.770	0.847	3.369
Bi-LSTM	0.541	0.689	0.847	5.095
NEURALSZZ	**0.786**	**0.860**	**0.891**	**2.197**

A. RQ1. Effectiveness of NEURALSZZ in identifying the root cause

Table III presents the results of NEURALSZZ and the baselines on identifying deletion lines as the root cause in bug-fixing commits. The results show that XGB outperforms all other ML baselines in recall@1. On the other hand, LR performs better than the other ML baselines in recall@2. For recall@3, RF achieves the best score. Finally, KNN performs the best in MFR. The DL approach performs relatively poorly compared to the ML baselines, which aligns with the findings of Wu et al. [46]. According to their study, simple text classification methods performed better than specially designed deep-learning approaches on clean datasets. This could be due to the use of pre-trained Glove word embeddings, which are trained on natural languages rather than code and may affect the performance of the DL approach.

Our evaluation shows that NEURALSZZ outperforms all the baselines across all metrics. It achieves recall scores of 0.779, 0.841, and 0.886 at the top 1, 2, and 3, respectively. It surpasses the best baseline by 8.5% in the recall at top 1. Moreover, NEURALSZZ achieves a better mean first rank of 2.425, improving upon the best baseline by 12.4%. To ensure validity, we also conduct statistical tests, specifically applying the Wilcoxon signed-rank test [50] at a 95% significance level on all metrics. The results show that p-values are smaller than 0.05, signifying statistically significant improvement at a 95% confidence level. Therefore, we believe that NEURALSZZ is a more effective method for identifying the root cause in the bug-fixing commits than the baselines.

RQ-1: NEURALSZZ *is more effective in identifying the root cause than the baselines, improving recall at top 1 by 8.5%. Additionally,* NEURALSZZ *achieves an improvement of 12.5% in the mean first rank compared to the best baseline.*

B. RQ2. Effectiveness of NEURALSZZ in cross-project setting

In cross-project prediction, we use DATASET1 as the test set, while DATASET2 and DATASET3 are used as the training set. We choose DATASET1 as the test set because it is a developer-informed oracle, thus it has the best quality. Note that the data from these three datasets have no overlap. Table IV presents the performance comparisons between NEURALSZZ and commonly used ML and DL baselines. The best results are highlighted in bold. Among the ML baselines, LR performs

TABLE V: The performance comparisons between our approach and baselines for finding bug-inducing commits

Approach	Precision	Recall	F1-score
BSZZ	0.376	**0.730**	0.496
AG-SZZ	0.348	0.604	0.441
MA-SZZ	0.319	0.543	0.401
RA-SZZ	0.333	0.466	0.388
NEURLSZZ@1	**0.834**	0.598	**0.698**
NEURLSZZ@2	0.728	0.635	0.678
NEURLSZZ@3	0.685	0.667	0.676

the worst in identifying the root cause deletion nodes at the top of the ranking list. However, the performance of all machine learning approaches in terms of recall@2 and recall@3 is close. Notably, KNN, an unsupervised approach, performs much worse in terms of the MFR metric. The DL approach also performs worse than the ML baselines

NEURALSZZ outperforms all other baselines in terms of all ranking metrics mentioned above. It achieves the highest recall scores at the top 1, 2, and 3 positions, with values of 0.786, 0.860, and 0.891, respectively. These represent improvements of 12.8%, 3.12%, and 2.89% over the best baseline. Additionally, NEURALSZZ achieves the best mean first rank of 2.19, which is an improvement of 5.51% compared to the best baseline. Among all the metrics, we observed that the improvement on recall@1 is particularly significant, as it can greatly reduce manual efforts for developers. These statistics provide strong evidence of the effectiveness of NEURALSZZ in identifying the root cause deletion nodes and its higher performance compared to the other baselines.

RQ-2: *The performance of* NEURALSZZ *in cross-project prediction is superior to that of the baselines, with notable improvements in the recall at top 1, 2, and 3 by 12.8%, 3.12%, and 2.89%, respectively. Additionally,* NEURALSZZ *achieves an improvement of 5.51% in the mean first rank compared to the best baseline.*

C. RQ3. Effectiveness of NEURALSZZ *in identifying bug-inducing commits*

Table V presents the results of both the SZZ algorithms and our approach in identifying the bug-inducing commits. As shown in the table, we can observe that the precisions of all previous SZZ algorithms are relatively low, resulting in a low F1-score. Among all previous SZZ algorithms, we can see that the performance of RA-SZZ is the lowest, which is consistent with many previous studies [19], [21].

Here, NEURLSZZ@N means that we use top N deletion lines in the ranking list to identify bug-inducing commits. From the table, we can see that NEURALSZZ outperforms all baselines on all metrics except for recall. Despite the decrease in the recall, our approach demonstrates a significant improvement in precision. In particular, NEURALSZZ enhances precision by 121.8%, 93.6%, and 82.1% for the top 1, top 2, and top 3 lines, respectively. Since recall and precision are equally important, we use F1-score as the primary evaluation metric to avoid bias. F1-score can measure whether an increase in precision outweighs the decrease in recall. NEURALSZZ achieves a better balance between precision and recall than the baselines. This is evident in the F1 scores, where NEURALSZZ achieves F1 scores of 0.698, 0.678, and 0.676 for the top 1, top 2, and top 3 lines. It outperforms the baselines by 40.7%, 36.7%, and 36.2%, respectively. Therefore, we believe that NEURALSZZ is more effective in detecting bug-inducing commits than original SZZ algorithms.

RQ-3: NEURALSZZ *achieves the best trade-off between precision and recall among all the baselines, outperforming them significantly.* NEURALSZZ *improves the best-performing baseline by 40.7% on the F1 score metric and by 121.8% on the precision metric. This shows that* NEURALSZZ *is better at identifying bug-inducing commits and provides a better balance between precision and recall.*

D. RQ4. Key designs of NEURALSZZ

In this RQ, we want to investigate the effectiveness of the key designs in NEURALSZZ. We also use the top 1, 2, and 3 lines in the ranking list to identify bug-inducing commits. NEURALSZZ consists of two key components: the codeBERT embedding layer, which captures semantic meaning from statements in the nodes, and the heterogeneous graph attention network, which enables each node to update its embedding from its neighbors. To evaluate the effectiveness of each component, we compare NEURALSZZ with its two variants: NEURALSZZ-c and NEURALSZZ-h. Each of them lacks one of the key designs. In NEURALSZZ-c, we replace codeBERT with Doc2vec [51], a machine learning algorithm that can convert statements to fixed-length vectors. There are two main types of Doc2vec models: Distributed Memory (DM) and Distributed Bag of Words (DBOW). The DM model takes into account the context of each word and the overall document, while the DBOW model only considers the words themselves. In this experiment, we apply the DM model. We use the output embeddings produced by Doc2vec as input for HAN. In NEURALSZZ-h, we remove the HAN layer and input codeBERT embeddings directly to the RankNet model. NEURALSZZ-c and NEURALSZZ-h share the same graph construction process with NEURALSZZ.

Table VI compares the performance of NEURALSZZ with its two variants: NEURALSZZ-c and NEURALSZZ-h, in identifying bug-inducing commits. The best results are highlighted in bold. NEURALSZZ outperforms both variants in all metrics except recall in the top 3 nodes in the ranking list. But the difference is only 0.4%. Comparing the performance of NEURALSZZ with NEURALSZZ-c and NEURALSZZ-h, we observe that the F1-score is improved by 3.35% in the top 1 node. The results confirm that incorporating CodeBERT and HAN can enhance the performance of our model.

While both key designs make contributions to performance, there are priorities between them. In this experiment, we

TABLE VI: The performance comparisons in ablation study

Model	top@1			top@2			top@3		
	Precision	Recall	F1-score	Precision	Recall	F1-score	Precision	Recall	F1-score
NeuralSZZ-h	0.807	0.575	0.672	0.698	0.634	0.664	0.677	**0.671**	0.674
NeuralSZZ-c	0.723	0.525	0.612	0.670	0.593	0.629	0.619	0.630	0.624
NeuralSZZ	**0.834**	**0.598**	**0.698**	**0.728**	**0.635**	**0.678**	**0.685**	0.667	**0.676**

TABLE VII: The statistics of bug-fixing commits that NeuralSZZ fails to correctly identify the root cause

Project	#LARGE	#SMALL	#DELETED
ambari	8	0	63
lucene	8	1	23
accumulo	2	0	8
hadoop	3	0	35
oozie	8	0	30

TABLE VIII: The performance comparisons between NeuralSZZ and V-SZZ for vulnerabilities

Approach	Precision	Recall	F1-score
BSZZ	0.359	0.687	0.472
AG-SZZ	0.521	0.731	0.608
MA-SZZ	0.418	0.761	0.540
RA-SZZ	0.433	0.591	0.499
V-SZZ	0.505	**0.836**	0.630
NeuralSZZ@1	**0.670**	0.662	**0.666**
NeuralSZZ@2	0.591	0.635	0.612
NeuralSZZ@3	0.547	0.768	0.639

observed that the node's own embedding plays a more critical role than its neighbors in identifying bug-inducing commits.

RQ-4: *Based on the experimental results presented in Table 5, incorporating both CodeBERT and HAN has improved the performance of the* NeuralSZZ *model in identifying bug-inducing commits. However, the results also indicate that CodeBERT makes a more significant contribution to the improvements in performance than HAN.*

VII. Discussion

A. Manual Analysis of Failed Cases

In this subsection, we manually analyze those bug-fixing commits where NeuralSZZ fails to correctly rank the deletion lines. We examine the highest-ranked lines in the corresponding ranking list to gain insights into the reasons for the failed ranking. The aim of the study is to provide an illustration of the factors contributing to the ranking failures. We use the results of the cross-project setting in RQ2 for this analysis.

Table VII presents the statistics of all bug-fixing commits that NeuralSZZ fails to correctly rank in Dataset1. For each project, the second and third columns present the number of large commits and small commits that fail to rank and the fourth column presents the average deleted lines in all commits. The table shows that the majority of the failed bug-fixing commits are large commits, except the one in the `lucene` project. Additionally, the average number of deleted lines in all the failed commits is high.

After conducting a manual analysis, we identified the following reasons why NeuralSZZ fails to correctly identify the root cause:

- Sometimes NeuralSZZ still fails to accurately capture the semantic meaning of deleted statements. A typical example is the only small bug-fixing commit in the `lucene` project, where NeuralSZZ wrongly identifies an `import` statement as the root cause. In total, there are four bug-fixing commits associated with this issue.
- Refactoring operations can also have an influence on the accuracy of NeuralSZZ. For example, in the commit 4c83c2200c of the `lucene` project, NeuralSZZ incorrectly ranks a deletion line that is part of a refactoring operation where the statement was encapsulated into a new function. This issue affects two bug-fixing commits in total.
- The third reason is that a few bug-fixing commits are excessively large. For example, consider the bug-fixing commit d330d406 in the `oozie` project. It consists of seven changed files and 144 deletion lines. NeuralSZZ fails to identify the root cause due to a large number of noisy information. It may need more comprehensive details to achieve the desired outcome.

To improve NeuralSZZ, we can incorporate refactoring detection tools to identify and eliminate noise in commits. Additionally, we can consider utilizing large language models, which have demonstrated impressive performance in various software engineering tasks [52]–[54], to tackle complex bug-fixing commits.

B. NeuralSZZ *for Vulnerabilities*

Vulnerabilities are a special kind of bugs that can significantly impact software systems. Hence, we are also interested in the effectiveness of NeuralSZZ in detecting bug-inducing commits by identifying the root cause for vulnerabilities. Bao et al. have proposed V-SZZ, a specialized SZZ algorithm designed for vulnerabilities [10]. They find that vulnerabilities tend to exist in the initial versions of the software. As a result, V-SZZ identifies the initial commits that introduce the deletion line in the vulnerability-fixing commit and regards them as vulnerability-inducing commits.

We use the dataset of 72 Java vulnerabilities annotated by Bao et al.. Note that this dataset only contains small fixing commits with equal or less than five deletion lines. We follow the same approach of identifying the root cause as described in Section IV-B. Specifically, we use NeuralSZZ to rank all deletion lines and select the top three deletion lines to apply the V-SZZ algorithm. Table VIII presents the performance of the previous SZZ algorithms and NeuralSZZ.

As shown in the table, NeuralSZZ@1 achieves the best performance in terms of precision and F1-score. It enhances the precision by 15% using the top 1 deletion line in the ranking list. Nonetheless, its recall significantly declines in comparison to V-SZZ. Overall, NeuralSZZ@1 has the highest F1-score, indicating that identifying the root cause can help improve the effectiveness of the SZZ algorithm for vulnerabilities.

C. Time Efficiency

It is important to consider the time efficiency when deploying the model. In this case, the whole inference time is 93.2 seconds on the entire test set using a simple GPU RTX3090. On average, the model takes about 0.59 seconds to process each bug-fixing commit.

D. Threats to Validity

Construct Validity. One potential threat to the construct validity of our study lies in the measurements used. To address this, we applied multiple measurements in our study, including common metrics for ranking algorithms and metrics for identifying bug-inducing commits.

Internal Validity. Errors may arise when we identify the root cause for each commit since the annotators are not the developers of the projects in our dataset. To address the potential errors, we made use of the bug-inducing commits provided in our dataset, which helped to narrow down the number of deletion lines that needed to be examined. Another threat to the internal validity is the strong assumption made in the paper, where we consider the top three deletion lines as the root cause. Indeed, there are some bugs whose root causes are unrelated to deletion lines, such as missing a null pointer check. However, the SZZ algorithm and its variants cannot find the inducing commits for those kinds of bugs. The objective of this research is to enhance the precision of the SZZ algorithm while maintaining an acceptable level of recall. Consequently, we exclude all commits whose root causes cannot be identified through deletion lines in comparison.

External Validity. Threats to external validity refer to the generalizability of our approach. One potential issue is the limited number of bugs in our dataset, which consists of 675 commits in total. However, this number is comparable to the number of bugs analyzed in prior studies [13], [55], [56]. Furthermore, we collect 17,027 pairs on average to train the model. To ensure generalizability, we also do cross-fold validation and cross-project prediction. Another potential threat is that our study only includes Java projects. In future research, we plan to examine more bugs in different programming languages to increase the generalizability of our approach.

VIII. Related Work

We summarize the related work from the following perspectives:

SZZ Applications in Software Engineering. The original SZZ algorithm and its variants have a wide application in software engineering research. A number of empirical investigations rely on the SZZ algorithm, such as code reviews [57], code smells [58], developer collaboration [59], [60], and technical debt [61]. For example, Bavota et al. [57] conducted a study to compare the probability of inducing a bug between unreviewed and reviewed commits. They used the SZZ algorithm to retrieve bug-inducing commits.

SZZ has also been applied in the field of defect prediction, where researchers have used the algorithm to extract bug-inducing commits and build datasets for training and evaluating their proposed models [1], [7], [8]. One example is the large-scale study of change-level defect prediction conducted by Kamei et al. [1]. In their study, they used B-SZZ to label bug-inducing commits and built a dataset for training and evaluating their proposed models. Afterward, Fan et al. [9] evaluated the performance of different types of SZZ algorithms. Keshavarz et al. [62] applied several filtering steps to reduce false positives in the dataset. These steps included considering the issue report date, the size of the commit, and the presence of trivial changes.

Detecting Noise in Commits. Many researchers focus on detecting noise in commits. The noise here often means refactoring operations [15], [17], nonessential changes [63] and casualty changes [64]. For example, the RefDiff tool [15] can identify 13 types of refactoring operations, while a newer tool proposed by Tsantalis et al [17]. supports 15 types of operations. Diffcat [63] aims to detect nonessential changes which include renaming of variables, methods, or classes, or trivial changes such as adding a keyword in programs written in Java. To detect noise better, Sejfia et al. [64] propose the taxonomy of casualty changes, which refers to changes that happen as a result of other changes and do not alter the logic of the program. They implement a tool called CASCADE to isolate this type of noise in commits. Our approach differs from these static methods in two ways. Compared to previous work, we use a deep learning model to capture the semantic meanings of changed codes and their relationships with other changed codes. Moreover, instead of focusing on identifying and filtering out the noise, we utilize a ranking model to detect the changed lines that are most likely to be the root cause.

Commit Detangling. Various approaches have been proposed to divide composite commits into smaller changes that are easier to comprehend and review [65]–[68], which is referred to as *commit detangling*. For example, Wang et al. [65] use a heuristic approach based on code dependency analysis and similarity detection to decompose commits for code review. However, our approach differs from commit detangling methods as it aims to identify the root cause of a bug-fixing commit, rather than dividing a complex commit into a single-activity commit. It is worth noting that a bug-fixing commit may be highly complicated but only related to a single activity, which falls outside the scope of commit detangling.

IX. Conclusion and Future Work

In this paper, we present our approach NeuralSZZ that prioritizes deletion lines according to their likelihood of being the root cause of a bug in bug-fixing commits. Our approach leverages the semantic meaning of each deletion line and its relationships with other deletion and addition lines. We then apply the SZZ algorithm on the deletion lines that are on top of the ranking list. To train and evaluate our model, we combine three high-quality datasets and manually annotate the root cause deletion lines in bug-fixing commits based on their corresponding bug-inducing commits. Our results demonstrate that NeuralSZZ outperforms baseline methods in ranking deletion lines and significantly improves the precision and F1-score of previous SZZ algorithms. In the future, we plan to extend our research by gathering additional high-quality datasets of bug-fixing and bug-inducing commits and evaluating our model on various programming languages.

X. Acknowledgment

This project is supported by the National Key Research and Development Program of China (No. 2021YFB2701102), the Fundamental Research Funds for the Central Universities (No. 226-2022-00064), and the National Science Foundation of China (No. U20A20173 and No. 62141222)

References

[1] Y. Kamei, E. Shihab, B. Adams, A. E. Hassan, A. Mockus, A. Sinha, and N. Ubayashi, "A large-scale empirical study of just-in-time quality assurance," *IEEE Transactions on Software Engineering*, vol. 39, no. 6, pp. 757–773, 2012.

[2] S. Kim, E. J. Whitehead, and Y. Zhang, "Classifying software changes: Clean or buggy?" *IEEE Transactions on software engineering*, vol. 34, no. 2, pp. 181–196, 2008.

[3] M. Asaduzzaman, M. C. Bullock, C. K. Roy, and K. A. Schneider, "Bug introducing changes: A case study with android," in *2012 9th IEEE Working Conference on Mining Software Repositories (MSR)*. IEEE, 2012, pp. 116–119.

[4] M. L. Bernardi, G. Canfora, G. A. Di Lucca, M. Di Penta, and D. Distante, "Do developers introduce bugs when they do not communicate? the case of eclipse and mozilla," in *2012 16th European Conference on Software Maintenance and Reengineering*. IEEE, 2012, pp. 139–148.

[5] G. Canfora, M. Ceccarelli, L. Cerulo, and M. Di Penta, "How long does a bug survive? an empirical study," in *2011 18th Working Conference on Reverse Engineering*. IEEE, 2011, pp. 191–200.

[6] J. Ell, "Identifying failure inducing developer pairs within developer networks," in *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, 2013, pp. 1471–1473.

[7] T. Jiang, L. Tan, and S. Kim, "Personalized defect prediction," in *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. Ieee, 2013, pp. 279–289.

[8] S. McIntosh and Y. Kamei, "Are fix-inducing changes a moving target? a longitudinal case study of just-in-time defect prediction," in *Proceedings of the 40th International Conference on Software Engineering*, 2018, pp. 560–560.

[9] Y. Fan, X. Xia, D. A. Da Costa, D. Lo, A. E. Hassan, and S. Li, "The impact of mislabeled changes by szz on just-in-time defect prediction," *IEEE transactions on software engineering*, vol. 47, no. 8, pp. 1559–1586, 2019.

[10] L. Bao, X. Xia, A. E. Hassan, and X. Yang, "V-szz: automatic identification of version ranges affected by cve vulnerabilities," in *Proceedings of the 44th International Conference on Software Engineering*, 2022, pp. 2352–2364.

[11] J. Śliwerski, T. Zimmermann, and A. Zeller, "When do changes induce fixes?" *ACM sigsoft software engineering notes*, vol. 30, no. 4, pp. 1–5, 2005.

[12] S. Kim, T. Zimmermann, K. Pan, E. James Jr *et al.*, "Automatic identification of bug-introducing changes," in *21st IEEE/ACM international conference on automated software engineering (ASE'06)*. IEEE, 2006, pp. 81–90.

[13] D. A. Da Costa, S. McIntosh, W. Shang, U. Kulesza, R. Coelho, and A. E. Hassan, "A framework for evaluating the results of the szz approach for identifying bug-introducing changes," *IEEE Transactions on Software Engineering*, vol. 43, no. 7, pp. 641–657, 2016.

[14] E. C. Neto, D. A. Da Costa, and U. Kulesza, "The impact of refactoring changes on the szz algorithm: An empirical study," in *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2018, pp. 380–390.

[15] D. Silva and M. T. Valente, "Refdiff: detecting refactorings in version histories," in *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*. IEEE, 2017, pp. 269–279.

[16] E. C. Neto, D. A. Da Costa, and U. Kulesza, "Revisiting and improving szz implementations," in *2019 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*. IEEE, 2019, pp. 1–12.

[17] N. Tsantalis, M. Mansouri, L. M. Eshkevari, D. Mazinanian, and D. Dig, "Accurate and efficient refactoring detection in commit history," in *Proceedings of the 40th international conference on software engineering*, 2018, pp. 483–494.

[18] M. Fowler, "Refactoring: Improving the design of existing code," in *11th European Conference. Jyväskylä, Finland*, 1997.

[19] G. Rosa, L. Pascarella, S. Scalabrino, R. Tufano, G. Bavota, M. Lanza, and R. Oliveto, "Evaluating szz implementations through a developer-informed oracle," in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 2021, pp. 436–447.

[20] X. Wang, H. Ji, C. Shi, B. Wang, Y. Ye, P. Cui, and P. S. Yu, "Heterogeneous graph attention network," in *The world wide web conference*, 2019, pp. 2022–2032.

[21] M. Wen, R. Wu, Y. Liu, Y. Tian, X. Xie, S.-C. Cheung, and Z. Su, "Exploring and exploiting the correlations between bug-inducing and bug-fixing commits," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2019, pp. 326–337.

[22] X. Song, Y. Lin, S. H. Ng, Y. Wu, X. Peng, J. S. Dong, and H. Mei, "Regminer: towards constructing a large regression dataset from code evolution history," in *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2022, pp. 314–326.

[23] "Replication package." [Online]. Available: https://figshare.com/s/6f2e62a954c163660328

[24] "The commit of the motivation example." [Online]. Available: https://github.com/apache/hadoop/commit/a2a5cb60b09

[25] "Tools for your java code," https://javaparser.org/, 2023-04-01.

[26] F. E. Allen, "Control flow analysis," *ACM Sigplan Notices*, vol. 5, no. 7, pp. 1–19, 1970.

[27] J. Ferrante, K. J. Ottenstein, and J. D. Warren, "The program dependence graph and its use in optimization," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 9, no. 3, pp. 319–349, 1987.

[28] B. G. Ryder, "Constructing the call graph of a program," *IEEE Transactions on Software Engineering*, no. 3, pp. 216–226, 1979.

[29] F. Yamaguchi, N. Golde, D. Arp, and K. Rieck, "Modeling and discovering vulnerabilities with code property graphs," in *2014 IEEE Symposium on Security and Privacy*. IEEE, 2014, pp. 590–604.

[30] J. Falleri, F. Morandat, X. Blanc, M. Martinez, and M. Monperrus, "Fine-grained and accurate source code differencing," in *ACM/IEEE International Conference on Automated Software Engineering, ASE '14, Vasteras, Sweden - September 15 - 19, 2014*, 2014, pp. 313–324. [Online]. Available: http://doi.acm.org/10.1145/2642937.2642982

[31] M. Defferrard, X. Bresson, and P. Vandergheynst, "Convolutional neural networks on graphs with fast localized spectral filtering," *Advances in neural information processing systems*, vol. 29, 2016.

[32] W. Hamilton, Z. Ying, and J. Leskovec, "Inductive representation learning on large graphs," *Advances in neural information processing systems*, vol. 30, 2017.

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

[34] Y. Sun and J. Han, "Mining heterogeneous information networks: a structural analysis approach," *Acm Sigkdd Explorations Newsletter*, vol. 14, no. 2, pp. 20–28, 2013.

[35] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang *et al.*, "Codebert: A pre-trained model for programming and natural languages," *arXiv preprint arXiv:2002.08155*, 2020.

[36] C. J. Burges, "From ranknet to lambdarank to lambdamart: An overview," *Learning*, vol. 11, no. 23-581, p. 81, 2010.

[37] O. Chapelle and Y. Chang, "Yahoo! learning to rank challenge overview," in *Proceedings of the learning to rank challenge*. PMLR, 2011, pp. 1–24.

[38] Y. Song, H. Wang, and X. He, "Adapting deep ranknet for personalized search," in *Proceedings of the 7th ACM international conference on Web search and data mining*, 2014, pp. 83–92.

[39] S. K. Karmaker Santu, P. Sondhi, and C. Zhai, "On application of learning to rank for e-commerce search," in *Proceedings of the 40th international ACM SIGIR conference on research and development in information retrieval*, 2017, pp. 475–484.

[40] R. Just, D. Jalali, and M. D. Ernst, "Defects4j: A database of existing faults to enable controlled testing studies for java programs," in *Proceedings of the 2014 international symposium on software testing and analysis*, 2014, pp. 437–440.

[41] S. Pan, L. Bao, X. Xia, D. Lo, and S. Li, "Fine-grained commit-level vulnerability type prediction by cwe tree structure."

[42] X. Tan, Y. Zhang, C. Mi, J. Cao, K. Sun, Y. Lin, and M. Yang, "Locating the security patches for disclosed oss vulnerabilities with vulnerability-commit correlation ranking," in *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, 2021, pp. 3282–3299.

[43] T. Fushiki, "Estimation of prediction error by using k-fold cross-validation," *Statistics and Computing*, vol. 21, pp. 137–146, 2011.

[44] J. Zhou, M. Pacheco, Z. Wan, X. Xia, D. Lo, Y. Wang, and A. E. Hassan, "Finding a needle in a haystack: Automated mining of silent vulnerability fixes," in *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2021, pp. 705–716.

[45] T. H. M. Le, D. Hin, R. Croft, and M. A. Babar, "Deepcva: Automated commit-level vulnerability assessment with deep multi-task learning," in *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2021, pp. 717–729.

[46] X. Wu, W. Zheng, X. Xia, and D. Lo, "Data quality matters: A case study on data label correctness for security bug report prediction," *IEEE Transactions on Software Engineering*, vol. 48, no. 7, pp. 2541–2556, 2021.

[47] A. Graves, A.-r. Mohamed, and G. Hinton, "Speech recognition with deep recurrent neural networks," in *2013 IEEE international conference on acoustics, speech and signal processing*. Ieee, 2013, pp. 6645–6649.

[48] J. Pennington, R. Socher, and C. D. Manning, "Glove: Global vectors for word representation," in *Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP)*, 2014, pp. 1532–1543.

[49] C. Parnin and A. Orso, "Are automated debugging techniques actually helping programmers?" in *Proceedings of the 2011 international symposium on software testing and analysis*, 2011, pp. 199–209.

[50] R. F. Woolson, "Wilcoxon signed-rank test," *Wiley encyclopedia of clinical trials*, pp. 1–3, 2007.

[51] J. H. Lau and T. Baldwin, "An empirical evaluation of doc2vec with practical insights into document embedding generation," *arXiv preprint arXiv:1607.05368*, 2016.

[52] M. U. Haque, I. Dharmadasa, Z. T. Sworna, R. N. Rajapakse, and H. Ahmad, "" i think this is the most disruptive technology": Exploring sentiments of chatgpt early adopters using twitter data," *arXiv preprint arXiv:2212.05856*, 2022.

[53] N. M. S. Surameery and M. Y. Shakor, "Use chat gpt to solve programming bugs," *International Journal of Information Technology & Computer Engineering (IJITC) ISSN: 2455-5290*, vol. 3, no. 01, pp. 17–22, 2023.

[54] S. Jalil, S. Rafi, T. D. LaToza, K. Moran, and W. Lam, "Chatgpt and software testing education: Promises & perils," *arXiv preprint arXiv:2302.03287*, 2023.

[55] A. Hindle, D. M. German, and R. Holt, "What do large commits tell us? a taxonomical study of large commits," in *Proceedings of the 2008 international working conference on Mining software repositories*, 2008, pp. 99–108.

[56] G. Rodríguez-Pérez, G. Robles, A. Serebrenik, A. Zaidman, D. M. Germán, and J. M. Gonzalez-Barahona, "How bugs are born: a model to identify how bugs are introduced in software components," *Empirical Software Engineering*, vol. 25, pp. 1294–1340, 2020.

[57] G. Bavota and B. Russo, "Four eyes are better than two: On the impact of code reviews on software quality," in *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2015, pp. 81–90.

[58] F. Palomba, G. Bavota, M. Di Penta, F. Fasano, R. Oliveto, and A. De Lucia, "On the diffuseness and the impact on maintainability of code smells: a large scale empirical investigation," in *Proceedings of the 40th International Conference on Software Engineering*, 2018, pp. 482–482.

[59] M. L. Bernardi, G. Canfora, G. A. Di Lucca, M. Di Penta, and D. Distante, "The relation between developers' communication and fix-inducing changes: An empirical study," *Journal of Systems and Software*, vol. 140, pp. 111–125, 2018.

[60] B. Çaglayan and A. B. Bener, "Effect of developer collaboration activity on software quality in two large scale projects," *Journal of Systems and Software*, vol. 118, pp. 288–296, 2016.

[61] S. Wehaibi, E. Shihab, and L. Guerrouj, "Examining the impact of self-admitted technical debt on software quality," in *2016 IEEE 23Rd international conference on software analysis, evolution, and reengineering (SANER)*, vol. 1. IEEE, 2016, pp. 179–188.

[62] H. Keshavarz and M. Nagappan, "Apachejit: a large dataset for just-in-time defect prediction," in *Proceedings of the 19th International Conference on Mining Software Repositories*, 2022, pp. 191–195.

[63] D. Kawrykow and M. P. Robillard, "Non-essential changes in version histories," in *Proceedings of the 33rd International Conference on Software Engineering*, 2011, pp. 351–360.

[64] A. Sejfia, Y. Zhao, and N. Medvidović, "Identifying casualty changes in software patches," in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2021, pp. 304–315.

[65] F. Balagtas-Fernandez and H. Hussmann, "A methodology and framework to simplify usability analysis of mobile applications," in *2009 IEEE/ACM international conference on automated software engineering*. IEEE, 2009, pp. 520–524.

[66] K. Herzig and A. Zeller, "The impact of tangled code changes," in *2013 10th Working Conference on Mining Software Repositories (MSR)*. IEEE, 2013, pp. 121–130.

[67] M. Barnett, C. Bird, J. Brunet, and S. K. Lahiri, "Helping developers help themselves: Automatic decomposition of code review changesets," in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 1. IEEE, 2015, pp. 134–144.

[68] B. Guo and M. Song, "Interactively decomposing composite changes to support code review and regression testing," in *2017 IEEE 41st annual computer software and applications conference (COMPSAC)*, vol. 1. IEEE, 2017, pp. 118–127.