Enhancing Fault Localization in Industrial Software Systems via Contrastive Learning

Chun Li[†‡], Hui Li[§], Zhong Li[†‡], Minxue Pan[†‡*], and Xuandong Li[†]
[†]State Key Laboratory for Novel Software Technology, Nanjing University, China
[‡]Software Institute, Nanjing University, China
[§]Samsung Electronics (China) R&D Centre, China
chunli@smail.nju.edu.cn, hui.li@samsung.com, {lizhong, mxp, lxd}@nju.edu.cn

***Abstract*—Engineers utilize logs as a primary resource for fault localization in large-scale software and system testing, a process that is notoriously time-consuming, costly, and labor-intensive. Despite considerable progress in automated fault localization approaches, their applicability remains limited in such settings, due to the unavailability of fine-grained features in logs essential for most existing fault localization methods. In response, we introduce FALCON, a novel log-based fault localization framework. FALCON organizes complex semantic log information into graphical representations and employs contrastive learning to capture the differences between passed and failed logs, enabling the identification of crucial fault-related features. It also incorporates a specifically designed transitive analysis-based adaptive graph augmentation to minimize the influence of fault-unrelated log information on contrastive learning. Through extensive evaluations against 34 spectrum-based and 4 learning-based fault localization methods, FALCON demonstrates superior performance by outperforming all the methods in comparison. In addition, FALCON demonstrated its practical value by successfully identifying 71 out of 90 faults with a file-level Top-1 accuracy rate during a one-month deployment within a global company's testing system.**

***Index Terms*—Industrial software debugging, fault localization, contrastive learning.**

I. INTRODUCTION

Locating the root cause of a fault is the critical first step in the debugging process for software programs. This endeavor can often be time-consuming, costly, and labor-intensive [1]. The complexity of this task is further exacerbated in an industrial context, particularly when dealing with system testing procedures for large-scale software systems. To alleviate the intricacies of debugging, engineers frequently rely on log files as a valuable source of information, as evidenced by a recent survey [2] indicating that logs are the most commonly utilized resource by industry professionals in the process of locating faults. However, system testing of large-scale software often generates logs exceeding tens of thousands of lines. The complexity of such software and its logs presents significant challenges to engineers in debugging. Consequently, there is a tremendous demand in the industry for automated fault localization tools.

Extensive research has been conducted on Fault Localization (FL) methods [3]–[14] to fully automate the process of diagnose program entities (files, methods, or statements) related to faults and facilitate the software debugging process. More specifically, fault localization techniques compute suspiciousness scores for different program entities, and then rank the program entities in descending order based on these scores. Although fault localization has evolved significantly over decades and achieved considerable success, current methods are primarily evaluated on datasets such as Defects4J [15] and unit tests, leaving their effectiveness in large-scale software and system testing yet to be determined. In particular, existing fault localization tools may encounter specific challenges in this context. These challenges primarily arise from the limitations in the availability of information used in localization. Due to the large scale of industrial software, it becomes impractical to collect fine-grained information such as line coverage [6], data/control flow dependencies [8], [16], line-level execution paths [11]–[13], or mutation-based coverage [9], [17]–[19]. The collection of these data often involves static/dynamic analysis, instrumentation, and mutation, which significantly impacts the efficiency of testing. Even if we could, given that system testing may involve thousands of methods, the fine-grained features would cause a rapid expansion in the size of logs, significantly increasing the spatial requirements of the method. Furthermore, information such as bug reports may also be missing. All these restrictions would lead to the ineffectiveness of many fault localization methods in the industrial context, including those based on slicing [20], [21], mutation [18], [19], information retrieval [22]–[24], and code change [1], [2].

Logs, being the most frequently used information by engineers when debugging in the industry [2], are generally available. Logs typically contain information about the threads, packages, files, and methods involved during the testing process. By analyzing logs, we can only obtain the methods executed during testing, their corresponding files, and packages, achieving coarse-grained method-level coverage. This limitation means that while spectrum-based [3]–[6], [25]–[27] and some learning-based [9]–[11], [17], [28]–[33] fault localization approaches (SBFL and LBFL) remain viable, their effectiveness is notably reduced under these conditions, as highlighted in Section II. Consequently, in the realm of large-scale software and system testing, existing methodologies either become impractical or their efficacy is significantly diminished due to these data constraints. There is a pressing need

* Corresponding author.

within the industry for a novel fault localization framework designed specifically for this scenario.

To address this need, we propose FALCON (**FAuLt LoCalizatiON**), a novel log-based fault localization framework specifically designed for industrial applications. The key insight underlying FALCON is the observation that faults can alter a software system's behavior, as elaborated in Section II. FALCON organized complex semantic information from logs into graphical representations and employs contrastive learning [34] to capture the differences between passed and failed logs and link these differences to the responsible faults, achieving effective fault localization. Contrastive learning is particularly suitable for this purpose, given that it can capture the differences in features between various samples effectively and accurately [34]–[37]. Another key insight is the prevalence of fault-irrelevant data within logs, which can obscure fault localization efforts. To counter this, FALCON incorporates a unique transitive analysis-based adaptive graph augmentation technique, reducing the influence of irrelevant information on the constrastive learning process and thus, enhancing localization accuracy.

Our evaluation of FALCON's performance in large-scale software and system testing involved over two thousand logs from system tests of eleven software projects, each exceeding one million lines of code, supplied by a global company. We benchmarked FALCON against 34 spectrum-based and 4 learning-based fault localization methods, demonstrating its exceptional efficacy by significantly outperforming all compared methods, including a remarkable 58.70% improvement over the top-performing method, GRACE [10]. Additionally, a deployment within the testing system of our industrial partner highlighted FALCON's practical value, where it successfully pinpointed 71 out of 90 faults at a file-level Top-1 accuracy rate over one month.

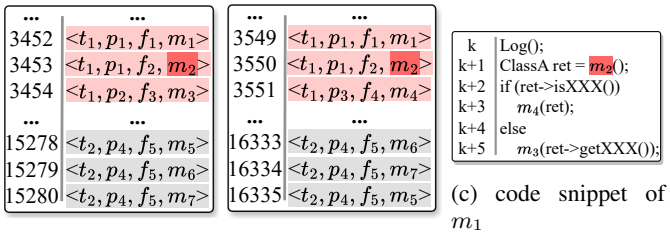
The main contributions of this paper are as follows:

- We propose a novel log-based fault localization framework, FALCON, which organizes complex semantic log information into graphical representations and employs contrastive learning to capture the differences in key features between passed and failed logs to facilitate fault localization.
- We develop a transitive analysis-based adaptive graph augmentation technique to mitigate the effects of substantial volumes of fault-irrelevant information in logs on the contrastive learning process, thereby enhancing the accuracy of fault localization.
- We conduct extensive evaluations of FALCON within an industrial context, and the results not only prove FALCON's superior performance over existing state-of-the-art approaches but also validate its practical utility and effectiveness.

II. METHODOLOGY

A. Motivating Example

In industry contexts, logs frequently act as the exclusive source for fault localization [2]. As discussed in Section I, this



(a) Bug triggered (b) No bug triggered (c) code snippet of m_1

Fig. 1: Motivating example: A simplified real bug and corresponding log.

reliance can significantly reduce the effectiveness of current fault localization methods. The core issue is the absence of necessary fine-grained information—such as data/control flow dependencies, line-level coverage, and additional resources like bug reports—that existing methods depend on but which are difficult or too costly to obtain in industrial environments.

To further illustrate the difficulties encountered by existing fault localization methods, we further present a motivating example in Figure 1. Figure 1 shows two real logs obtained from our industrial partner, which have been simplified and anonymized. More specifically, Figures 1a and 1b respectively depict partial logs resulting from whether the bug in the faulty method m_2 was triggered or not triggered, where t, p, f and m denote the thread, package, file, and method associated with each log entry. Fault-related and fault-unrelated sections are colored as red and gray, respectively, for presentation. Figure 1c provides the context of method m_1 involved in line 3453 of the bug-triggered log. Due to the triggering of the bug in m_2, the erroneous boolean value returned by the *isXXX* method called by variable *ret* at line $k+2$ leads to the invocation of m_3, ultimately resulting in differences in the log compared to the log when the bug is not triggered and m_4 is invoked instead. For this case, all SBFL and LBFL methods that involve fine-grained coverage, fail to rank the faulty method within the Top-1. This is because, under coarse-grained coverage, we can only observe that m_2 appears in both the passed and failed logs; however, we cannot observe the differences in internal method execution paths between passed and failed test cases. From a spectrum perspective alone, m_3 only appears in the failed log, while the faulty method m_2 appears in both the failed and passed log. Therefore, all SBFL formulas assign a higher suspiciousness score to m_3. This not only renders SBFL methods ineffective but also reduces the effectiveness of LBFL methods that rely on SBFL-calculated suspiciousness scores or fine-grained coverage data. Therefore, there is an urgent need for an improved fault localization method in the industry context.

B. Problem Statement

The main problem tackled in this paper is *how to effectively and efficiently analyze logs to localize faults in industrial software systems*.

A log, as depicted in Figure 1, consists of multiple lines, with each line containing information such as the running thread number, package, file, and method. Formally, let $L =$

$(l_1, l_2, \ldots, l_n)$ be a log where $l_1, l_2, \ldots, l_n$ denotes the lines of the log L ordered chronologically. Each line l_i, in general, comprises the running thread number, the package, file, and method executed. That is, $l_i = (t, p, f, m)$, with t, p, f, m respectively representing the thread, package, file, and method.

The vast and diverse content of logs significantly complicates the analysis for fault localization, presenting notable challenges. **Firstly**, these logs are not only voluminous but also entangled. Given the large scale of industrial software systems, the resulting testing logs can easily span over ten thousand lines, containing a multitude of messages related to packages, files, and thousands of methods, which complicates the analysis process. Furthermore, the involvement of multiple threads in testing introduces concurrent logging of information. This concurrency results in the intermixing of messages and states from different threads within the logs, increasing the difficulty of extracting relevant and critical information for fault localization. **Secondly**, pinpointing bug-related features presents a significant challenge. Effective fault localization requires the identification of unexpected or suspicious features within failed logs that could be indicative of bugs. The reliance on failed logs alone means we are without references of correct behavior, complicating our ability to discern which features are anomalous. Moreover, not every feature that deviates from the norm is inherently suspicious. Given the logs' extensive data volume, isolating these features becomes a formidable task.

In summary, to achieve effective and efficient fault localization in the industrial context, we need to address the following challenges:

• **Challenge I**: How to extract program semantics from complex logs that can aid in fault localization.

• **Challenge II**: How to identify fault-related suspicious program semantics to perform fault localization.

C. Our Approach

In this section, we introduce our idea to solve the aforementioned challenges.

• **Extracting the hierarchical structure and intra-thread method execution sequence as the program semantics for fault localization.** To effectively analyze the complex logs and obtain effective information for fault localization, we propose extracting the hierarchical structure and intra-thread method execution sequence from the logs as program semantics for fault localization. The hierarchical structure refers to the relationships between program entities involved in testing (package, file, and method). As shown in Figure 1a, the line 3452 of the log contains information about method m_1, which belongs to file f_1, and file f_1, in turn, belongs to package p_1. The hierarchical structure records all program entities involved in the testing and their relationships, which helps reflect the spectrum (coverage) information and trace how faults propagate among these entities, thereby providing useful information for fault localization. The intra-thread method execution sequence is the order of method executions within a thread. For instance, from the last three lines of Figure 1a, we can deduce that the sequence of method executions in thread t_2 is (m_5, m_6, m_7). The intra-thread method execution sequence reflects the trace and state changes of the thread, which helps understand program behavior and determine the context of the fault. Therefore, by extracting the hierarchical structure and intra-thread method execution sequence from the logs, we can focus on information that is useful for localizing faults.

• **Identifying fault-related suspicious program semantics by comparison.** To identify suspicious semantics related to faults for fault localization, we propose comparing the program semantics of passed and failed logs to identify fault-related suspicious program semantics. This is inspired by the observation that the triggering of bugs may lead to changes in information within the logs. For instance, in the motivating example, whether the bug is triggered results in line 3454 in Figure 1a and line 3551 in Figure 1b logging different methods, files, and packages. This difference will lead to a change in the program semantics extracted from the logs. Therefore, by comparing the differences in the program semantics extracted from a large number of passed and failed logs, the suspicious program semantics in the failed logs can be identified. To realize this idea, we employ contrastive learning to deep learn the program semantics extracted from the logs. Contrastive learning is commonly used to measure the differences in various complex features across different samples to learn and encode the features into sample representations [34]–[37]. Through contrastive learning, the model can learn how to discern these differences and reflect them in the representations of the samples. Transitively, the representations of samples obtained through contrastive learning exhibit clear separation (i.e., distinct differences). Therefore, through contrastive learning, we can learn a representation space capable of effectively distinguishing between the program semantics of passed and failed logs.

• **Mitigating the impact of fault-unrelated program semantics by augmentation.** Given that passed and failed logs may also exhibit differences in fault-unrelated program semantics, directly applying contrastive learning to compare these differences could potentially reduce the effectiveness of fault localization. For instance, in the motivating example, the log ends with the termination of a remote access module. However, slight differences in system states lead to variations in the execution path of thread t_2 at the end (m_5, m_6, m_7 and m_6, m_7, m_5). Contrasting these differences may impact our ability to identify truly suspicious program semantics. Therefore, we've enhanced our methodology by integrating transitive analysis-based adaptive graph augmentation within the contrastive learning framework. We aim to refine the learning process by selectively pruning irrelevant connections between program entities, thus sharpening the focus on fault-related semantics.

III. Design

In this section, we first provide an overview of FALCON workflow. Then, we elaborate on the technical details of each stage in FALCON.

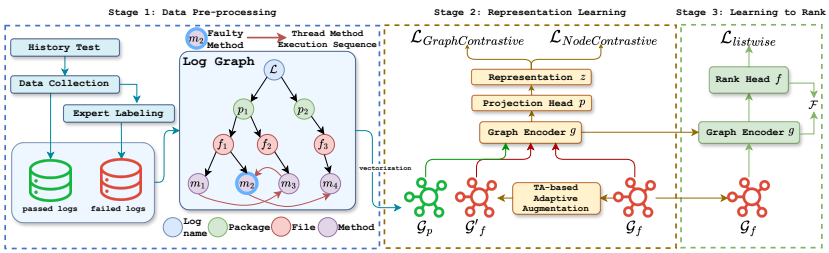


Fig. 2: The overview of FALCON.

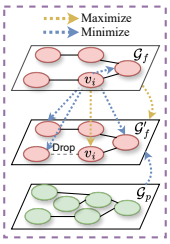


Fig. 3: Example of contrastive learning process.

Overview. Figure 2 presents the overall stage of FALCON. As shown, FALCON consists of three main stages: Data Pre-processing, Representation Learning, and Learning to Rank. In the Data Pre-processing stage, our industrial partner collects logs from historical tests and annotates the faulty methods in failed tests, ultimately forming a dataset that includes both passed and failed logs. Each log in the dataset is transformed into a graph to facilitate the modeling of two important program semantics: hierarchical structure and intra-thread method execution sequence. In the Representation Learning stage, we employ contrastive learning to understand and capture the differences in program semantics between passed and failed graphs caused by the faulty entity to identify suspicious program semantics in the new failed logs. Specifically, given the failed graph $\mathcal{G}_f$ constructed from failed logs, FALCON first generates an enhanced failed graph $\mathcal{G'}_f$ from the failed graph using adaptive graph augmentation based on transitive analysis. It then conducts node contrastive learning between $\mathcal{G}_f$ and $\mathcal{G'}_f$, aiming to reduce the influence of semantics unrelated to faults, enhance the emphasis on fault-relevant semantics, and achieve improved node embeddings. Then, given the passed graph $\mathcal{G}_p$ constructed from passed logs, we perform graph contrastive learning between the failed graph $\mathcal{G}_f$, the enhanced failed graph $\mathcal{G'}_f$, and the passed graph $\mathcal{G}_p$ to compel the model to discern the semantic discrepancies between logs indicating success and failure, and thereby capturing fault-related program semantics during the learning process. Figure 3 presents an example of our contrastive learning process. We maximize the similarity of node embeddings for the same node between $\mathcal{G}'_f$ and $\mathcal{G}_f$ while minimizing the similarity of embeddings for different nodes in node contrastive learning. We maximizing the graph embedding similarity between $\mathcal{G}'_f$ and $\mathcal{G}_p$, while minimize the graph embedding similarity between $\mathcal{G}_f$ and $\mathcal{G}'_f$ in graph contrastive learning. In the Learning to Rank stage, we train a rank head f to convert the suspicious program semantics identified by the graph encoder to suspiciousness values of program entities to rank entities and locate fault.

A. Data Pre-processing

In the data pre-processing stage, FALCON extracts the program semantics from the logs and then processes these program semantics to facilitate the subsequent representation learning stage. Specifically, the data pre-processing stage consists of two steps: program semantics extraction and vectorization.

Program Semantics Extraction. As discussed in Section II-C, FALCON employs the hierarchical structure and intra-thread method execution sequence as the program semantics for fault localization. To achieve this, we extract the relevant information from the logs and organize it using a graph structure to represent the two program semantics. The advantage of utilizing a graph structure is that the containment relationships between different program entities and the execution order of methods can be naturally represented by the same graph. As such, we first treat all program entities involved in the log, including packages, files, and methods, as nodes within the graph, and employ a dedicated node to represent the entire log, thereby constructing the set of nodes in the graph. Following this, we construct the edge set to represent the two types of relationships. We employ the containment relationship to connect package, file, and method nodes that appear on the same line, and utilize the special log node to connect all package nodes within the log, thereby forming the hierarchical structure of entities within the entire log. Then, we connect method nodes according to the order of their execution within threads to represent the intra-thread method execution sequences.

More specifically, given a log L, the graph for log L is denoted by $\mathcal{G}_L = \{\mathcal{V}_L, \mathcal{E}_L\}$, where $\mathcal{V}_L$ and $\mathcal{E}_L$ denote node set and edge set, respectively. Let n_L denote the special node representing the log itself, and T, P, F, and M respectively denote the sets of threads, packages, files and methods which are formed by extracting t, p, f, m from each line $l = (t, p, f, m)$ in L. Then, the node set of $\mathcal{G}_L$ is defined as $\mathcal{V}_L = T \cup P \cup F \cup M \cup \{n_L\}$. The attribute of each node corresponds to its respective name, such as the method name, file name, package name, or log name. Based on the node set $\mathcal{V}_L$, we further construct the edge set $\mathcal{E}_L$ to establish relationships between different nodes, thus representing the hierarchical structure and intra-thread method execution sequence. Let $u \rightarrow v$ denote node u point to node v. Then, we can represent the hierarchical structure by constructing $p \rightarrow f$ and $f \rightarrow m$ for each line l, and $n_L \rightarrow p$ for all $p \in P$. Next, let $S_t = (m_1, m_2, \ldots, m_k)$ denote the sequence of methods executed by t, we model the intra-thread method execution sequence by making $m_i \rightarrow m_{i+1}$ for each $t \in T$, where k is the length of S_t and $i \in [1, k-1]$. Finally, the edge set $\mathcal{E}_L$ is constructed by $\mathcal{E}_L = H \cup I$, where H denotes the edges represent the hierarchical structure and I denotes the edges represent the intra-thread method execution sequence.

Vectorization. In the process of representation learning, the model necessitates input in the form of vectors or matrices to facilitate efficient computation and training. Therefore, we need to further vectorize the previously constructed graph. Specifically, we use the generic Sentence Embedding tool SentenceBERT [38] to encode the attribute of a node into a corresponding vector because the packages, files, functions, and logs are typically named in a way that aligns with general reading habits. That is, we vectorize graph $\mathcal{G}_L$ into a feature matrix $\boldsymbol{X} \in \mathbb{R}^{|\mathcal{V}_L| \times D}$ and an adjacency matrix $\boldsymbol{A} \in \{0,1\}^{|\mathcal{V}_L| \times |\mathcal{V}_L|}$, where D denotes the embedding size and $\boldsymbol{A}_{ij} = 1$ iff $(v_i, v_j) \in \mathcal{E}_L$. Furthermore, we label whether a node is a fault entity to associate the learned program semantics with fault entities. Specifically, if L is a passed log, then the nodes of $\mathcal{G}_L$ do not have corresponding labels. If L is a failed log, then there is an additional label matrix $\boldsymbol{Y} \in \{0,1\}^{|\mathcal{V}_L| \times 1}$, where $y_i \in \boldsymbol{Y}$ is the label of v_i. The $y_i = 1$ indicates that node v_i is a faulty node and $y_i = 0$ indicates it is a non-faulty node.

B. Representation Learning

In the representation learning stage, FALCON utilizes contrastive learning to train a graph encoder which is capable of identifying fault-related suspicious program semantics. The insight is that the triggering of bugs may lead to changes in the information within the logs. Therefore, based on contrastive learning, the graph encoder is trained by comparing the program semantics of passed and failed logs. This training objective enables the encoder to distinguish between the program semantics of passed and failed logs and learn how to identify the fault-related suspicious program semantics during the distinguishing process. Below, we explain more details of the representation learning stage.

1) Model Architecture: To construct the graph encoder, we leverage a Graph Neural Network (GNN) in FALCON. The reason for utilizing GNNs is that the program semantics are represented by graphs (c.f., Section III-A) and GNNs have been demonstrated to be effective in processing graph data in many tasks [39], [40]. In particular, following previous work [10], FALCON employs the Gate Graph Neural Network (GGNN) [41] to construct the graph encoder. GGNN is a variant of GNN that utilizes *gated* units to preserve long-term dependencies such as Long Short Term Memory (LSTM) [42] and thus it could better capture the complex structures and long-term dependencies in the program semantics we extracted.

2) Contrastive Learning: To learn to identify fault-related suspicious program semantics, we employ contrastive learning to train the GGNN encoder. However, as discussed in Section II-C, directly training the encoder using contrastive learning is sub-optimal due to the negative impact of fault-unrelated program semantics. Thus, to mitigate the negative impacts of fault-unrelated information, we introduce transitive analysis-based adaptive graph augmentation (AGA) to help contrastive learning. AGA generates augmented graphs that are less influenced by fault-unrelated information. Then, by incorporating the augmented graphs into the contrastive learning process, we can alleviate the negative impacts of fault-unrelated information, thus enabling the model to better identify fault-related suspicious program semantics. Furthermore, to enhance the model's learning of program semantics and rank the node for fault localization, we design a two-step contrastive learning framework. The first step is *Node Contrastive Learning (NCL)* which trains the encoder by comparing and aligning the original graph with the augmented graph generated by AGA. As such, the encoder is able to focus more on learning fault-related information because augmented graphs are less influenced by fault-unrelated information. Then, the *Graph Contrastive Learning (GCL)* step derives the program semantics of the graph from node embeddings learned by NCL and trains the encoder through contrasting between passed graph and failed graph to learn identifying suspicious program semantics related to faults. In the following, we elaborate on each step in detail.

Transitive analysis based adaptive graph augmentation. The goal of adaptive graph augmentation (AGA) is to generate augmented graphs that are less influenced by fault-unrelated information. To achieve this, we first identify the fault-unrelated subgraphs and then apply the topological-level modification (i.e., selectively dropping certain edges) to the original graphs based on the fault-unrelated subgraphs. By removing edges in the fault-unrelated subgraphs, the fault-unrelated information is prevented from influencing other parts of the original graph. Therefore, the augmented graphs would be less impacted by the fault-unrelated information.

To identify the fault-unrelated subgraph, we first identify the fault-related subgraph, as the fault-related information could be more intuitively obtained in failed graph. Specifically, we identify the fault-related subgraph by including all nodes in the graph that are related to the faulty node (including the faulty node itself) and the edges associated with these nodes. Given a failed log graph $\mathcal{G}_f$, let $\mathcal{V}_r$ be the set of nodes related to the faulty node. We construct $\mathcal{V}_r$ through transitive predecessor and transitive successor analysis. Transitive predecessors include all nodes that can be reached via a directed path to the faulty node, essentially forming the predecessor closure of that node [43]. Transitive successors are similarly defined. Then, we define the fault-related edge set $\mathcal{E}_r$, which includes all edges in $\mathcal{G}_f$ connected to nodes in $\mathcal{V}_r$ without distinguishing between incoming and outgoing edges. Finally, we can obtain the fault-unrelated subgraph $\mathcal{G}_{fu}$ by $\mathcal{G}_{fu} = \{\mathcal{V}_f \backslash \mathcal{V}_r, \mathcal{E}_f \backslash \mathcal{E}_r\}$.

Based on the identified fault-unrelated subgraph, we conduct the topological-level modification on the original graph to produce the augmented graph. Specifically, following previous work [44], we calculate edge centrality using the degree of the node and use edge centrality to assess the probability of dropping edges. Let $\varphi_c(v)$ be the degree of the node v. We calculate the edge centrality of edge (u, v) as $w_{uv} = \varphi_c(v)$. Then, we sample a subset from the edge set of $\mathcal{G}_{fu}$ and remove them in the original graph with probability p_{uv}, where $(u, v) \in \mathcal{E}_{fu}$. The p_{uv} is defined based on edge centrality [44]

and computed as:

$$p_{uv} = \min\left(\frac{w_{\max} - w_{uv}}{w_{\max} - w_{avg}}, p_\tau\right) \quad (1)$$

where $p_\tau < 1$ is a cut-off probability, used to truncate the probabilities since extremely high removal probabilities will lead to overly corrupted graph structures, $w_{\max}$ and w_{avg} is the maximum and average of w_{uv}. The Equation 1 is a normalization step that transforms edge centrality into probabilities. Finally, by sampling a subset from the edge set of the fault-unrelated graph with a probability p and removing its edges from the original graph, we obtain the adaptively augmented failed graph $\mathcal{G}'_f$.

Node Contrastive Learning. Based on the augmented graphs, we first conduct node contrastive learning between failed graph $\mathcal{G}_f$ and augmented failed graph $\mathcal{G}'_f$ to learn program semantics at the node level. Since the augmented failed graph $\mathcal{G}'_f$ is less influence by the fault-unrelated information, by comparing and aligning $\mathcal{G}_f$ and $\mathcal{G}'_f$, we can make the encoder more focus on the fault-related information, thus learning better program semantics for fault localization. More specifically, we achieve this by designing a node contrastive loss function based on the message passing mechanism.

Message passing is a universal mechanism of GNNs, where the embedding of a node is computed based on the embeddings of its neighbors. Intuitively, since the edges in the fault-unrelated subgraph have been removed from $\mathcal{G}'_f$, the fault-unrelated information cannot be transmitted to other nodes. At this point, the embeddings of nodes in $\mathcal{G}'_f$ do not contain information from the fault-unrelated subgraph. Therefore, by aligning the embeddings of identical nodes between $\mathcal{G}_f$ and $\mathcal{G}'_f$, we can compel the model to ignore fault-unrelated information when outputting embeddings for nodes in the $\mathcal{G}_f$.

Based on the message passing mechanism, we design a node contrastive loss function to align $\mathcal{G}_f$ and $\mathcal{G}'_f$. In this function, we increase the similarity of embeddings of identical nodes in these two graphs while minimizing the similarity of embeddings of distinct nodes. This training approach yields better node representations [45]. It can also reduce the impact of fault-unrelated features during the process of increasing the similarity of embeddings of the same nodes, i.e., aligning the embeddings of the same nodes. More specifically, we first obtain the dimensionality-reduce node embeddings $\boldsymbol{Z}_f$ and $\boldsymbol{Z}'_f$ for graphs $\mathcal{G}_f$ and $\mathcal{G}'_f$ through $p(g(\mathcal{G}))$, where g is graph encoder to obtain node embeddings and p is projection head to perform dimensionality reduction. Note that we introduce a projection head p here to reduce the dimensionality of the embeddings output by the graph encoder g, thereby decreasing computational complexity and enhancing the performance of the encoder [35], [46]. Then, we employ the node contrastive loss function. For a node v_i, its corresponding vectors are $\boldsymbol{z_i} \in \boldsymbol{Z}_f$ and $\boldsymbol{z}'_i \in \boldsymbol{Z}'_f$. We consider $(\boldsymbol{z}_i, \boldsymbol{z}'_i)$ as a positive pair, while $\boldsymbol{z}_i$, $\boldsymbol{z}'_i$, and embeddings of other nodes form negative pairs. Let $\theta(\cdot,\cdot)$ is cosine similarity. Then the node contrastive loss function of v_i is defined by:

$$\ell_i = -\log \frac{e^{\theta(\boldsymbol{z}_i, \boldsymbol{z}'_i)/\tau}}{e^{\theta(\boldsymbol{z}_i, \boldsymbol{z}'_i)/\tau} + \sum_{k\neq i} e^{\theta(\boldsymbol{z}_i, \boldsymbol{z}_k)/\tau} + \sum_{k\neq i} e^{\theta(\boldsymbol{z}_i, \boldsymbol{z}'_k)/\tau}} \quad (2)$$

where τ is a temperature parameter [47]. Taking Figure 3 as an example, through Equation 2, we enforce each positive pair to converge while ensuring each negative pair diverges. Since edges in the fault-unrelated subgraph are removed, $\boldsymbol{v}_i$ does not contain part of the information from the fault-unrelated subgraph. Therefore, in the process of aligning $\boldsymbol{z}_i$ closer to $\boldsymbol{z}'_i$, the model will learn to ignore part of fault-unrelated features and enhance the influence of fault-related features on $\boldsymbol{z}_i$. Overall, through adaptive graph augmentation, *message passing*, and Equation 2, we can efficiently learn fault-related features from a local perspective, while minimizing the impact of fault-unrelated features, and finally obtain improved node representations. Finally, our whole node contrastive loss function is as shown in Equation 3.

$$\mathcal{L}_{NodeContrastive} = \frac{1}{|\mathcal{V}_f|}\sum_{i=1}^{|\mathcal{V}_f|} \ell_i \quad (3)$$

Graph Contrastive Learning. After completing node contrastive learning, we conduct graph contrastive learning to contrast the difference in global program semantics between passed graphs and failed graphs, thereby training the model to identify suspicious program semantics. Through graph contrastive learning, the model is able to encode the local suspicious program semantics into node embeddings to rank nodes. Specifically, inspired by FaceNet [48], we use the $\mathcal{G}'_f$ as the anchor, aiming to increase the similarity between the $\mathcal{G}_f$ and it, while decreasing the similarity between the $\mathcal{G}_p$ and it. The insights behind this design can be summarized in two folds. First, choosing $\mathcal{G}'_f$ as the anchor is because, by removing edges in the fault-unrelated subgraph, its representation more prominently highlights the features of the fault-related parts. Second, we optimize the model by reducing the similarity between $\mathcal{G}_p$ and $\mathcal{G}'_f$ and increasing the similarity between $\mathcal{G}_f$ and $\mathcal{G}'_f$. This enables us to compel the model to focus more on the differences in fault-related features between $\mathcal{G}_p$ and $\mathcal{G}'_f$ and minimize its focus on fault-unrelated information when encountering real failed logs.

More specifically, We first employ a widely used *average* readout function to obtain the graph embedding $\boldsymbol{s} = \frac{1}{N}\sum_{i=1}^{N} \boldsymbol{Z}_i$ from the dimensionality-reduced node embeddings, where N is the number of nodes. Let $\boldsymbol{s}_p, \boldsymbol{s}_f, \boldsymbol{s}'_f$ respectively represent the graph embedding of passed graph $\mathcal{G}_p$, failed graph $\mathcal{G}_f$, and augmented failed graph $\mathcal{G}'_f$. We perform the graph contrastive loss function to achieve our optimized objective as Equation 4.

$$\mathcal{L}_{GraphContrastive} = \sum \left[||\boldsymbol{s}_f - \boldsymbol{s}'_f||_2^2 - ||\boldsymbol{s}_p - \boldsymbol{s}'_f||_2^2 + \alpha\right] \quad (4)$$

where α is a margin that is enforced between positive and negative pairs [48].

TABLE I: Statistics about the dataset.

Attribute	Value	Attribute	Value
Total #logs	2524	Average #Files	352.48
Average #Threads	90.05	Average #Methods	1060.90

C. Learning to Rank

After representation learning, FALCON needs to convert the embeddings of nodes into suspiciousness values for the corresponding program entities to rank these entities and perform fault localization in this stage. Specifically, we will train a rank head f from scratch and fine-tune the graph encoder g using the failed logs to achieve our goal. Following previous work [10], we also use the *listwise* loss function to optimize the model as it is suitable for ranking nodes in a graph. That is, we view the final ranking model $\mathcal{F}$ in FALCON as $\mathcal{F}(\mathcal{G}) = f(g(\mathcal{G}))$. Specifically, the rank head f will linearly transformed the node embeddings into $\boldsymbol{Y}' \in R^{|\mathcal{V}_f|\times 1}$. Then, we utilize the *softmax* function to normalize the suspiciousness scores of each node as $p(v_i) = \exp\{y_i'\}/\sum_{j=1}^{n}\exp\{y_j'\}$, where $p(v_i)$ denotes the probability of node v_i being faulty, n is the number of target nodes. If conducting method-level localization, then the target nodes are methods set M; if conducting file-level localization, then the target nodes are files set F. Finally, the *listwise* loss function is denoted by $\mathcal{L}_{list} = -\sum_{i=1}^{n} y_i \log(p(v_i))$, where y_i denotes the ground truth label for node v_i. FALCON will train rank head f and fine-tune encoder g using $\mathcal{L}_{list}$ and rank the target node based on $p(v_i)$.

IV. EXPERIMENT DESIGN

We evaluate FALCON on the following research questions:

- RQ1: How does FALCON's effectiveness compare to that of state-of-the-art fault localization techniques?
- RQ2: How does FALCON perform in the cross-project prediction scenario?
- RQ3: How do different components within FALCON affect its overall effectiveness?
- RQ4: How does the training efficiency of FALCON compare to that of LBFL techniques?

A. Industrial Subject Systems

To evaluate FALCON within the backdrop of large-scale software and system testing, we collaborated with our industrial partner which operates multiple digital product lines worldwide. They provided us with log files generated during system testing of eleven software that cover diverse product lines and platforms. Each software exceeds one million lines of code, averaging around 170 packages, 2000 files, and 50,000 methods. Software Engineers from our industrial partner initially collected logs generated from failed tests, followed by logs from retesting after these bugs were fixed. Each failed log contains only one fault. Table I reports key statistics about the dataset. We have a total of 2524 logs, with an average of 90.05 threads, 114.07 packages, 352.48 files, and 1060.90 methods involved in each log.

B. Compared Approaches

We compare FALCON against 38 state-of-the-art fault localization approaches, including 34 *spectrum-based fault localization approaches* and 4 *learning-based fault localization approaches*. Note that there are also many other types of fault localization approaches like mutation-based [18], [19], slicing-based [20], [21], IR-based fault localization [22]–[24]. However, as discussed in Section II, these approaches are not applicable for comparison because the information they require is unavailable in the logs. Thus, we omit them in our experiments. For *spectrum-based fault localization approaches*, we follow prior work [9], [17] to choose 34 representative SBFL formulae. For detailed information on these SBFL formulae, please refer to Table 7 in [17]. Regarding *learning-based fault localization approaches*, we consider 4 recent SOTA approaches, which are:

•**MULTRIC** [30] combines different SBFL formulae using learning-to-rank for fault localization.

•**FLUCCS** [31] combines different SBFL formulae and code features (e.g., code churn) to rank faulty entities.

•**DeepFL** [9] employs a deep learning-based technique to locate faults by learning existing/latent features from various aspects of test cases and programs

•**GRACE** [10] represents the method's AST and test cases using a graph structure and employs graph neural networks to learn and rank fault entities.

C. Evaluation Setup

Evaluation Metric. Following previous work [9]–[11], [17], [31], we adopt Recall at Top-N (N=1, 3, 5) and MFR (Mean First Rank) as our evaluation metrics. Note that MFR and MAR (Mean Average Rank) are numerically equal in the context of single-fault localization. Thus, we replace MAR with MRR (Mean Reciprocal Rank), a more suitable metric widely adopted in information retrieval systems [49], [50], for assessment. Let r denote the rank of the faulty entity in the ranked list, $\mathcal{N}$ denote the number of failed logs, and $\mathbb{I}(\cdot)$ denote indicator function. We formally define Recall at Top-N as $(\sum_{i=1}^{\mathcal{N}}\mathbb{I}(r_i \leq N))/\mathcal{N}$, define MFR as $(\sum_{i=1}^{\mathcal{N}} r_i)/\mathcal{N}$, and define MRR as $(\sum_{i=1}^{\mathcal{N}}(r_i)^{-1})/\mathcal{N}$. Higher Top-N and MRR, along with lower MFR, indicate better localization performance. Following previous work [9], [10], [17], we use the *worst* ranking for the tied elements that have the same suspiciousness scores. Due to the large volume of data and the cost of model training, we employed ten-fold cross-validation in subsequent experiments instead of the leave-one-out validation method used in previous work [9], [10].

Localization Level. We perform fault localization at the file and method levels because we are unable to collect line coverage data when only the logs are available. For the compared approaches, because their original implementations do not account for file-level localization, we follow the setup used in FLUCCS [31] to select the highest suspiciousness value among all methods in a file to represent the overall suspiciousness of that file.

Within-project and Cross-Project Setting. RQ1 studies FALCON's performance in a within-project setting, a scenario where a single project's data populates both training and testing sets. This setting mirrors prevalent industry practices, exemplified by our industrial partner, where projects are continuously developed over years or decades, accumulating a vast array of logs and enabling the training of tailored project-specific models. In the within-project setting, the dataset is carefully curated to ensure that for any failed log in the test set, neither it nor its corresponding successful log is in the training set.

RQ2 explores FALCON's effectiveness in a cross-project setting, seeking to evaluate its generality across projects. Consistent with existing literature [11], this setting adopts the leave-one-out validation method: testing each failed log in a project while training a model on the logs from all the remaining projects.

Implementation. We build FALCON based on PyTorch Geometric [51], one of the most widely used deep learning frameworks on graphs built upon PyTorch. For the parameters in FALCON, we determine them through grid search to ensure the best performance of FALCON. The result values of the parameters in FALCON are shown in Table II. p_τ is cuf-off probability in augmentation. $lr_{contrastive}$ denotes the learning rate of the optimizer on contrastive learning, lr_{rank} denotes the learning rate of the optimizer on learning to rank, and weight decay is L2-Regularization. h^g, h^p represent the hidden vector of the graph encoder and projection head. τ is a temperature parameter in $\mathcal{L}_{NodeContrastive}$ and α is the margin in $\mathcal{L}_{GraphContrastive}$. $epoch_n, epoch_g, epoch_r$ represent the epoch of node contrastive learning, graph contrastive learning, and learning to rank, respectively.

Regarding the compared approaches, we directly adopt their open-source implementations and employ the default configurations of the original papers to ensure the accuracy of experimental repetition. We provide method-level coverage for the studied comparison approaches by collecting all executed methods recorded in the logs because the method-level is the finest granularity obtainable from logs. Furthermore, because these approaches require input from a test suite containing at least one failed test to perform fault localization, we enlist the software engineers from our industrial partner to organize the existing dataset into test suites, ensuring that each suite contained at least one failing test.

For the studied LBFL techniques, there are constraints in accessing all the necessary features. In the case of FLUCCS, we cannot obtain information about code changes because the software we used have more than three years of development history. Querying the age and churn of all faulty methods in the code change system would incur significant manual collection costs. In the case of DeepFL, we cannot obtain mutation-based features or certain information used for calculating textual similarity (types of exceptions, messages, and stack traces). With an average of thousands of functions involved in each test, performing mutations would significantly increase the cost of testing. Information such as exception types and stack traces is inherently missing. In the case of GRACE, we can only connect the Test Node with the root node of the AST of the covered method since we cannot obtain line coverage information due to the logs not recording line numbers. The limited information in the logs further demonstrates the importance of FALCON, which can effectively locate faults in industrial scenarios.

TABLE II: Hyper-parameters of FALCON.

Hyper-parameter	Value	Hyper-parameter	Value	Hyper-parameter	Value
p_τ	0.3	optimizer	Adam	$lr_{contrastive}$	1e-3
lr_{rank}	1e-5	weight decay	1e-4	embedd size	384
h^g size	768	h^p size	128	α	0.2
τ	0.4	$epoch_n$	30	$epoch_g$	10
$epoch_r$	10	batch size	12	graph layer	6

TABLE III: Comparison with state-of-the-art at method level.

Techniques	Top-1↑	Top-3↑	Top-5↑	MFR↓	MRR↑
RussellRao	22.13	23.20	23.20	289.37	0.23
Hamann	18.89	20.00	20.00	301.39	0.20
SϕrensenDice	16.57	17.71	17.71	309.97	0.18
MULTRIC	30.86	32.10	37.04	134.47	0.33
FLUCCS	27.16	44.44	45.68	63.74	0.36
DeepFL	2.47	8.64	18.52	302.33	0.09
GRACE	40.00	50.43	58.26	47.50	0.48
FALCON	**63.48**	**82.61**	**85.22**	**8.23**	**0.74**

All experiments are conducted on a workstation with AMD Ryzen 9 3900XT, 32GB memory, and two RTX 4090 GPUs, running Ubuntu 20.04.

V. RESULT ANALYSIS

A. RQ1: Effectiveness of FALCON

In this RQ, the primary objective is to evaluate the effectiveness of FALCON in locating faults. We conduct a comparative analysis of FALCON against 38 baseline approaches outlined in Section IV-B, examining performance at both file and method levels. Results are detailed in Table III and Table IV. Due to spatial constraints, only the three SBFL formulae with top performance–RussellRao [52], Hamann [53], and SϕrensenDice [54], [55]–are highlighted in the tables, while comprehensive results are accessible on our project website [56].

Analysis of Tables III and IV yields notable insights: First, FALCON consistently surpasses competing models across all five metrics, at both file and method levels. Notably, at the method level, FALCON achieves Top-1 localization accuracy of 63.48%, representing significant improvements of 186.85%, 105.70%, 133.72% and 58.70% over the RussellRao, MULTRIC, FLUCCS, and GRACE baselines, respectively. Additionally, MFR and MRR metrics exhibit substantial enhancements, with FALCON showing an 82.67% improvement in MFR and a 54.16% improvement in MRR compared to the best-performing baseline, GRACE. Similar trends are observed at the file level, underscoring FALCON's effectiveness in fault detection within industrial software contexts.

Second, the comparison between FALCON and GRACE—both of which leverage graph structures to capture

TABLE IV: Comparison with state-of-the-art at file level.

Techniques	Top-1↑	Top-3↑	Top-5↑	MFR↓	MRR↑
RussellRao	25.87	32.53	37.87	72.64	0.32
Hamann	24.44	34.89	41.11	62.15	0.33
SϕrensenDice	21.52	30.67	36.76	70.06	0.30
MULTRIC	45.45	51.52	57.58	53.36	0.52
FLUCCS	43.21	69.14	70.37	23.00	0.56
DeepFL	23.46	35.80	40.74	37.63	0.32
GRACE	55.65	58.26	63.48	13.30	0.60
FALCON	**75.65**	**86.09**	**93.91**	**2.17**	**0.83**

program semantics and apply GNNs for learning—reveals that FALCON markedly outperforms GRACE at both analysis levels. This indicates that FALCON's consideration of program semantics is more adept for fault localization in industrial applications. Unlike GRACE, which solely employs static Abstract Syntax Trees, FALCON integrates both static and dynamic information, including a hierarchical structure of static inter-entity relationships and intra-thread dynamic execution sequences of methods. The improvement also suggests that our contrastive learning is more effective in identifying the suspicious program semantics in failed logs than supervised learning employed by GRACE. This comprehensive semantic approach and contrastive learning enable FALCON to more accurately mirror program behavior, enhancing fault localization efficacy.

Finally, DeepFL's method-level performance is observed to be relatively low, which we attribute to its struggle with the extreme class imbalance prevalent in industrial software system fault localization—where failed tests may involve thousands of methods, but only a few are faulty. This imbalance can bias models relying on binary cross-entropy loss functions [57], [58] towards non-fault methods. In contrast, FALCON would not suffer from such an issue since it employs the *listwise* loss function that always ranks the faulty node ahead of non-faulty nodes.

To further confirm the observations above, we have followed previous works [9], [10] to perform the Wilcoxon signed-rank test [59] with Bonferroni corrections [60] to investigate the statistical significance between FALCON and other baselines. The results show that FALCON is significantly better than all studied techniques at the significance level of 0.05, with p-values from 1.87e-138 to 1.45e-63 at the file level and from 1.81e-141 to 4.56e-68 at the method level.

B. RQ2: Cross-project Effectiveness of FALCON

In RQ2, we further evaluate the method-level effectiveness of FALCON in a cross-project scenario. Table V presents the comparison results between FALCON and GRACE in the cross-project scenario. Note that, due to the space limit, we only present the results of GRACE which achieves the best performance among all the compared approaches. From Table III and Table V, we find that the performance of FALCON experiences a certain degree of decline in the cross-project scenario. For example, there is a 22.32% decrease in the Top-1 accuracy of FALCON. This is reasonable because the cross-project scenario is more challenging than the within-project scenario. In the within-project scenario, various methods are likely to exhibit similarities because they originate from the same projects, thus facilitating the learning of fault-related features. However, in the cross-project scenario, methods may vary significantly due to differences between projects. This diversity presents a challenge in learning fault-related features. On the other hand, as shown in Table V, despite the performance degradation of FALCON in the cross-project scenario, it still achieves superior performance compared to GRACE (the best compared approach). Specifically, the improvements of FALCON compared with GRACE achieve 61.51%, 79.78%, 60.99%, 28.06%, and 56.41%, in terms of Top-1, Top-3, Top-5, MFR and MRR, respectively. Furthermore, we can observe from Table III and Table V that the performance drop of FALCON in the cross-project scenario is also smaller than that of GRACE. For example, in terms of top-1 accuracy, FALCON decreased by 22.32%, which is less than the 23.67% decrease experienced by GRACE. These results demonstrate the superiority of FALCON in the cross-project scenario. Compared to existing LBFL methods, which learn fault localization solely from the features of failed tests, FALCON adopts contrastive learning to train the model by comparing passed logs and failed logs. This combines the features of both types of logs, thus resulting in higher performance in the cross-project scenario.

TABLE V: Cross-project effectiveness at method level.

Techniques	Top-1↑	Top-3↑	Top-5↑	MFR↓	MRR↑
GRACE	30.53	34.92	39.46	124.83	0.39
FALCON	**49.31**	**62.78**	**63.53**	**89.80**	**0.61**

TABLE VI: Method level fault localization ablation study.

Variant	Top-1↑	Top-3↑	Top-5↑	MFR↓	MRR↑
w/o NCL (GCL + AGA)	55.65	71.30	75.65	41.36	0.65
w/o GCL (NCL + AGA)	59.13	70.43	77.39	20.36	0.66
w/o AGA (NCL + GCL)	53.04	65.22	69.57	38.75	0.6
FALCON (NCL + GCL + AGA)	**63.48**	**82.61**	**85.22**	**8.23**	**0.74**

C. RQ3: Ablation Study

In this RQ, we conduct a series of ablation studies to further analyze the impact of each component in FALCON. In particular, we consider three variants of FALCON: without transitive analysis-based adaptive graph augmentation (AGA), without node contrastive learning (NCL), and without graph contrastive learning (GCL). Table VI summarizes the study results. We can see from Table VI that, all the variants of FALCON lead to a decrease in Top-N, MFR, and MRR, indicating that both our AGA and contrastive learning contribute to enhancing the fault localization capability of FALCON. Beyond the performance decline, we can also observe from Table VI that the model performs worse when only contrastive learning (without AGA) is conducted, as compared to the FALCON. Specifically, in method-level localization, the Top-1 accuracy with *w/o AGA* decreased by 16.44% compared to FALCON. This suggests that pure contrastive learning may be influenced by fault-unrelated program semantics in logs.

TABLE VII: Training time cost of studied LBFL techniques.

Techniques	MULTRIC	FLUCCS	DeepFL	GRACE	FALCON
Training Time	2m12s	4h23m10s	12m6s	6m19s	19m14s

Our AGA effectively addresses this issue, thus improving the effectiveness of contrastive learning and fault localization. From Table VI, we further find that NCL plays a more important role in FALCON than GCL, as removing NCL results in a larger performance loss. The reason is that GCL relies on the node embeddings learned from NCL. When the NCL is removed, the node embeddings would be less effective due to the impact of fault-unrelated information, consequently diminishing the effectiveness of GCL.

D. RQ4: Efficiency of FALCON

This RQ empirically analyzes the efficiency of FALCON. Table VII presents the time cost of training a model using FALCON and the studied LBFL approaches. As shown in this table, compared to the studied LBFL approaches, FALCON requires more time to train the model because FALCON requires transitive analysis-based adaptive graph augmentation, node contrastive learning, and graph contrastive learning. Nevertheless, it is important to highlight that FALCON achieves significantly better performance in locating faults than the studied LBFL approaches, as demonstrated in Section V-A Therefore, we believe such a time cost of FALCON is worthwhile for achieving better models. Furthermore, considering that the training process is offline, the training time of FALCON (i.e., only takes 20 minutes) is also acceptable in practice.

E. Threats to Validity.

The main threat to **internal** validity lies in the technical implementation of FALCON, the compared approaches, and experimental scripts. To mitigate this threat, we developed FALCON based on widely used libraries and employed the original implementations of the compared approaches. We carefully check the source code of FALCON and experimental scripts. The main threat to **external** validity lies in the benchmark used in our study. To reduce this threat, we perform experiments on an industrial dataset provided by our industrial partner, which contains 2524 logs from 11 large-scale software each exceeding one million lines of code. Furthermore, we compare FALCON with 34 SBFL formulae and 4 LBFL techniques in our experiments. The main threat to **construct** validity lies in the parameters in FALCON and metrics used in experiments. To mitigate this threat, we present the detailed parameter settings in Section IV-C. To reduce the threat from metrics employed, we employed various metrics that are widely used in fault localization and information retrieval research.

VI. PRACTICAL EVALUATION

We have successfully deployed FALCON onto the development pipeline of product T from our industrial partner. A series of large-scale software related to product T is continually being developed and tested. The logs generated during system testing often exceed ten thousand lines and involve thousands of functions, presenting significant challenges to manual debugging. FALCON is capable of automatically modeling key features in logs and localizing faults without manual intervention, admirably meeting their demands. Indeed, they largely appreciated the performance of FALCON on their systems. Here, we report practical evaluation results based on the first month of usage since deploying FALCON. Specifically, developers in the company primarily use file-level localization. When a test fails, they assign the failed test to the engineer responsible for the file with the highest suspiciousness value. During a month of deployment, the tests conducted were to determine whether a series of software related to product T would perform as expected on new hardware. Consequently, although the software had appeared in FALCON's training data, the distribution of program semantics in the logs varied due to the differences in hardware. Table VIII shows the effectiveness of FALCON. From the table, we can see that FALCON achieved promising results in its actual deployment. Specifically, among a total of 90 system tests, FALCON reached a Top-1 accuracy of 78.88%. Although the software involved in the tests was present in the training set, the tests were for compatibility of this software on new hardware, resulting in different log distributions. Therefore, this result further demonstrates the generality and practicability of FALCON. Through an informal interview, developers indeed confirmed the usefulness of FALCON in practice.

TABLE VIII: Practical evaluation results of FALCON over a one-month deployment period.

Method	Top-1 Accuracy	Top-1 Failed	Number of Tests
FALCON	71	19	90

VII. RELATED WORKS

Spectrum-based Fault Localization. Spectrum-based fault localization (SBFL) [3]–[6], [25]–[27] is a crucial technique for the precise localization of faults within software systems. By leveraging coverage information generated from passed/-failed tests, SBFL assigns suspiciousness scores (probability of being faulty) to distinct program entities (such as a statement or a method). At the core of SBFL lies the assumption that the likelihood of program entities being faulty increases if they are covered by more failed tests and less passed tests. More specifically, given a buggy software, a test suite (containing at least one failed test case), and coverage information, SBFL extracts the following tuple information for each program entity e: (e_p, e_f, n_p, n_f): e_p and e_f represent the number of passed/failed tests covering the program entity, respectively. Similarly, n_p and n_f represent the number of passed/failed tests that do not cover the program entity, respectively. Employing the tuple, the SBFL is capable of calculating the suspiciousness score of each program entity using a variety of ranking formulae.

Learning-based Fault Localization. In addition to the traditional SBFL approaches, deep learning technologies have found extensive application in fault localization [9]–[11],

[13], [14], [17], [28]–[33]. Learning-based fault localization (LBFL) can be broadly divided into two categories: one is *learning-to-represent* [10], [11], [14], [28], [29], and the other is *learning-to-combine* [9], [17], [30]–[32]. The *learning-to-represent* techniques focus on how to better represent finer-grained coverage and employ learning algorithms to process this information for the calculation of suspiciousness scores. The *learning-to-combine* techniques are concerned with how to synergize information that encapsulates various feature dimensions using learning processes. FALCON falls under the learning-to-represent category that utilizes contrastive learning to represent the suspicious program semantics for fault localization.

GNNs and Contrastive Learning. GNNs and contrastive learning have demonstrated their potential and utility across multiple domains. GNNs [61]–[64] update each node's representation by aggregating information from neighboring nodes, enabling the model to capture complex graph structural features. Contrastive learning acquires powerful feature representations by learning the differences between data points and popularity in the fields of computer vision [35], [36], [65] and natural language processing [66], [67]. Recently, some works utilize contrastive learning or GNNs for different software engineering tasks such as clone detection [39], vulnerabilities detection [40], and type inference [68]. FALCON further combines and extends their application into fault localization.

VIII. CONCLUSION

In this paper, we propose a novel fault localization framework, named FALCON, designed to effectively locate faults within the context of industrial software systems. FALCON organizes complex semantic log information into graphical representations and employs contrastive learning to capture the differences between passed and failed logs, enabling the identification of crucial fault-related features. It also incorporates a specifically designed transitive analysis-based adaptive graph augmentation to minimize the influence of fault-unrelated log information on contrastive learning. The experiment results and practical evaluation demonstrate the effectiveness of FALCON in fault localization within industrial software systems. We believe FALCON makes a significant contribution to advancing the practice of fault localization.

ACKNOWLEDGMENT

We thank the anonymous reviewers for their valuable feedback. This research was supported by the National Natural Science Foundation of China under Grant Nos. 62372227 and 62032010. This research was also supported by the Fund of State Key Laboratory for Novel Software Technology in Nanjing University under Grant No.ZZKT2024B09.

REFERENCES

[1] A. R. Chen, T. P. Chen, and J. Chen, "How useful is code change information for fault localization in continuous integration?" in *ASE*. ACM, 2022, pp. 52:1–52:12.

[2] M. Wen, J. Chen, Y. Tian, R. Wu, D. Hao, S. Han, and S. Cheung, "Historical spectrum based fault localization," *IEEE Trans. Software Eng.*, vol. 47, no. 11, pp. 2348–2368, 2021.

[3] R. Abreu, P. Zoeteweij, and A. J. C. van Gemund, "An evaluation of similarity coefficients for software fault localization," in *PRDC*. IEEE Computer Society, 2006, pp. 39–46.

[4] M. Y. Chen, E. Kiciman, E. Fratkin, A. Fox, and E. A. Brewer, "Pinpoint: Problem determination in large, dynamic internet services," in *DSN*. IEEE Computer Society, 2002, pp. 595–604.

[5] S. Reis, R. Abreu, and M. d'Amorim, "Demystifying the combination of dynamic slicing and spectrum-based fault localization," in *IJCAI*. ijcai.org, 2019, pp. 4760–4766.

[6] X. Xie, T. Y. Chen, F. 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, pp. 31:1–31:40, 2013. [Online]. Available: https://doi.org/10.1145/2522920.2522924

[7] X. Xie, Z. Liu, S. Song, Z. Chen, J. Xuan, and B. Xu, "Revisit of automatic debugging via human focus-tracking analysis," in *ICSE*. ACM, 2016, pp. 808–819.

[8] M. Zeng, Y. Wu, Z. Ye, Y. Xiong, X. Zhang, and L. Zhang, "Fault localization via efficient probabilistic modeling of program semantics," in *44th IEEE/ACM 44th International Conference on Software Engineering, ICSE 2022, Pittsburgh, PA, USA, May 25-27, 2022*. ACM, 2022, pp. 958–969. [Online]. Available: https://doi.org/10.1145/3510003.3510073

[9] X. Li, W. Li, Y. Zhang, and L. Zhang, "Deepfl: integrating multiple fault diagnosis dimensions for deep fault localization," in *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2019, Beijing, China, July 15-19, 2019*, D. Zhang and A. Møller, Eds. ACM, 2019, pp. 169–180. [Online]. Available: https://doi.org/10.1145/3293882.3330574

[10] Y. Lou, Q. Zhu, J. Dong, X. Li, Z. Sun, D. Hao, L. Zhang, and L. Zhang, "Boosting coverage-based fault localization via graph-based representation learning," in *ESEC/FSE '21: 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Athens, Greece, August 23-28, 2021*, D. Spinellis, G. Gousios, M. Chechik, and M. D. Penta, Eds. ACM, 2021, pp. 664–676. [Online]. Available: https://doi.org/10.1145/3468264.3468580

[11] Y. Li, S. Wang, and T. N. Nguyen, "Fault localization with code coverage representation learning," in *ICSE*. IEEE, 2021, pp. 661–673.

[12] ——, "Fault localization to detect co-change fixing locations," in *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2022, Singapore, Singapore, November 14-18, 2022*, A. Roychoudhury, C. Cadar, and M. Kim, Eds. ACM, 2022, pp. 659–671. [Online]. Available: https://doi.org/10.1145/3540250.3549137

[13] W. Chen, W. Chen, J. Liu, K. Zhao, and M. Zhang, "Supconfl: Fault localization with supervised contrastive learning," in *Internetware*. ACM, 2023, pp. 44–54.

[14] T. B. Le, D. Lo, C. L. Goues, and L. Grunske, "A learning-to-rank based fault localization approach using likely invariants," in *ISSTA*. ACM, 2016, pp. 177–188.

[15] R. Just, D. Jalali, and M. D. Ernst, "Defects4j: a database of existing faults to enable controlled testing studies for java programs," in *ISSTA*. ACM, 2014, pp. 437–440.

[16] Z. Zhang, Y. Lei, X. Mao, M. Yan, X. Xia, and D. Lo, "Context-aware neural fault localization," *IEEE Trans. Software Eng.*, vol. 49, no. 7, pp. 3939–3954, 2023.

[17] X. Li and L. Zhang, "Transforming programs and tests in tandem for fault localization," *Proc. ACM Program. Lang.*, vol. 1, no. OOPSLA, pp. 92:1–92:30, 2017. [Online]. Available: https://doi.org/10.1145/3133916

[18] S. Moon, Y. Kim, M. Kim, and S. Yoo, "Ask the mutants: Mutating faulty programs for fault localization," in *ICST*. IEEE Computer Society, 2014, pp. 153–162.

[19] M. Papadakis and Y. L. Traon, "Metallaxis-fl: mutation-based fault localization," *Softw. Test. Verification Reliab.*, vol. 25, no. 5-7, pp. 605–628, 2015.

[20] H. Agrawal, J. R. Horgan, S. London, and W. E. Wong, "Fault localization using execution slices and dataflow tests," in *ISSRE*. IEEE Computer Society, 1995, pp. 143–151.

[21] M. Renieris and S. P. Reiss, "Fault localization with nearest neighbor queries," in *ASE*. IEEE Computer Society, 2003, pp. 30–39.

[22] R. K. Saha, M. Lease, S. Khurshid, and D. E. Perry, "Improving bug localization using structured information retrieval," in *ASE*. IEEE, 2013, pp. 345–355.

[23] Q. Wang, C. Parnin, and A. Orso, "Evaluating the usefulness of ir-based fault localization techniques," in *ISSTA*. ACM, 2015, pp. 1–11.

[24] V. Murali, L. Gross, R. Qian, and S. Chandra, "Industry-scale ir-based bug localization: A perspective from facebook," in *ICSE (SEIP)*. IEEE, 2021, pp. 188–197.

[25] W. E. Wong, V. Debroy, R. Gao, and Y. Li, "The dstar method for effective software fault localization," *IEEE Trans. Reliab.*, vol. 63, no. 1, pp. 290–308, 2014.

[26] L. Zhang, M. Kim, and S. Khurshid, "Localizing failure-inducing program edits based on spectrum information," in *ICSM*. IEEE Computer Society, 2011, pp. 23–32.

[27] J. A. Jones, M. J. Harrold, and J. T. Stasko, "Visualization of test information to assist fault localization," in *ICSE*. ACM, 2002, pp. 467–477.

[28] W. Zheng, D. Hu, and J. Wang, "Fault localization analysis based on deep neural network," *Mathematical Problems in Engineering*, vol. 2016, pp. 1–11, 01 2016.

[29] Z. Zhang, Y. Lei, X. Mao, and P. Li, "CNN-FL: an effective approach for localizing faults using convolutional neural networks," in *SANER*. IEEE, 2019, pp. 445–455.

[30] J. Xuan and M. Monperrus, "Learning to combine multiple ranking metrics for fault localization," in *30th IEEE International Conference on Software Maintenance and Evolution, Victoria, BC, Canada, September 29 - October 3, 2014*. IEEE Computer Society, 2014, pp. 191–200. [Online]. Available: https://doi.org/10.1109/ICSME.2014.41

[31] J. Sohn and S. Yoo, "FLUCCS: using code and change metrics to improve fault localization," in *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis, Santa Barbara, CA, USA, July 10 - 14, 2017*, T. Bultan and K. Sen, Eds. ACM, 2017, pp. 273–283. [Online]. Available: https://doi.org/10.1145/3092703.3092717

[32] D. Zou, J. Liang, Y. Xiong, M. D. Ernst, and L. Zhang, "An empirical study of fault localization families and their combinations," *IEEE Trans. Software Eng.*, vol. 47, no. 2, pp. 332–347, 2021. [Online]. Available: https://doi.org/10.1109/TSE.2019.2892102

[33] X. Meng, X. Wang, H. Zhang, H. Sun, and X. Liu, "Improving fault localization and program repair with deep semantic features and transferred knowledge," in *44th IEEE/ACM 44th International Conference on Software Engineering, ICSE 2022, Pittsburgh, PA, USA, May 25-27, 2022*. ACM, 2022, pp. 1169–1180. [Online]. Available: https://doi.org/10.1145/3510003.3510147

[34] A. Jaiswal, A. R. Babu, M. Z. Zadeh, D. Banerjee, and F. Makedon, "A survey on contrastive self-supervised learning," *CoRR*, vol. abs/2011.00362, 2020. [Online]. Available: https://arxiv.org/abs/2011.00362

[35] T. Chen, S. Kornblith, M. Norouzi, and G. E. Hinton, "A simple framework for contrastive learning of visual representations," in *Proceedings of the 37th International Conference on Machine Learning, ICML 2020, 13-18 July 2020, Virtual Event*, ser. Proceedings of Machine Learning Research, vol. 119. PMLR, 2020, pp. 1597–1607. [Online]. Available: http://proceedings.mlr.press/v119/chen20j.html

[36] K. He, H. Fan, Y. Wu, S. Xie, and R. B. Girshick, "Momentum contrast for unsupervised visual representation learning," in *2020 IEEE/CVF Conference on Computer Vision and Pattern Recognition, CVPR 2020, Seattle, WA, USA, June 13-19, 2020*. Computer Vision Foundation / IEEE, 2020, pp. 9726–9735. [Online]. Available: https://doi.org/10.1109/CVPR42600.2020.00975

[37] Y. Chen, Z. Ding, and D. A. Wagner, "Continuous learning for android malware detection," in *USENIX Security Symposium*. USENIX Association, 2023, pp. 1127–1144.

[38] N. Reimers and I. Gurevych, "Sentence-bert: Sentence embeddings using siamese bert-networks," in *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing, EMNLP-IJCNLP 2019, Hong Kong, China, November 3-7, 2019*, K. Inui, J. Jiang, V. Ng, and X. Wan, Eds. Association for Computational Linguistics, 2019, pp. 3980–3990. [Online]. Available: https://doi.org/10.18653/v1/D19-1410

[39] N. Mehrotra, N. Agarwal, P. Gupta, S. Anand, D. Lo, and R. Purandare, "Modeling functional similarity in source code with graph-based siamese networks," *IEEE Trans. Software Eng.*, vol. 48, no. 10, pp. 3771–3789, 2022. [Online]. Available: https://doi.org/10.1109/TSE.2021.3105556

[40] X. Cheng, H. Wang, J. Hua, G. Xu, and Y. Sui, "Deepwukong: Statically detecting software vulnerabilities using deep graph neural network," *ACM Trans. Softw. Eng. Methodol.*, vol. 30, no. 3, pp. 38:1–38:33, 2021. [Online]. Available: https://doi.org/10.1145/3436877

[41] Y. Li, D. Tarlow, M. Brockschmidt, and R. S. Zemel, "Gated graph sequence neural networks," in *4th International Conference on Learning Representations, ICLR 2016, San Juan, Puerto Rico, May 2-4, 2016, Conference Track Proceedings*, Y. Bengio and Y. LeCun, Eds., 2016. [Online]. Available: http://arxiv.org/abs/1511.05493

[42] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural Comput.*, vol. 9, no. 8, pp. 1735–1780, 1997.

[43] M. E. J. Newman, *Networks: An Introduction*. Oxford University Press, 2010. [Online]. Available: https://doi.org/10.1093/ACPROF:OSO/9780199206650.001.0001

[44] Y. Zhu, Y. Xu, F. Yu, Q. Liu, S. Wu, and L. Wang, "Graph contrastive learning with adaptive augmentation," in *WWW '21: The Web Conference 2021, Virtual Event / Ljubljana, Slovenia, April 19-23, 2021*, J. Leskovec, M. Grobelnik, M. Najork, J. Tang, and L. Zia, Eds. ACM / IW3C2, 2021, pp. 2069–2080. [Online]. Available: https://doi.org/10.1145/3442381.3449802

[45] T. Wang and P. Isola, "Understanding contrastive representation learning through alignment and uniformity on the hypersphere," in *Proceedings of the 37th International Conference on Machine Learning, ICML 2020, 13-18 July 2020, Virtual Event*, ser. Proceedings of Machine Learning Research, vol. 119. PMLR, 2020, pp. 9929–9939. [Online]. Available: http://proceedings.mlr.press/v119/wang20k.html

[46] K. Gupta, T. Ajanthan, A. van den Hengel, and S. Gould, "Understanding and improving the role of projection head in self-supervised learning," *CoRR*, vol. abs/2212.11491, 2022. [Online]. Available: https://doi.org/10.48550/arXiv.2212.11491

[47] Z. Wu, Y. Xiong, S. X. Yu, and D. Lin, "Unsupervised feature learning via non-parametric instance discrimination," in *2018 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2018, Salt Lake City, UT, USA, June 18-22, 2018*. Computer Vision Foundation / IEEE Computer Society, 2018, pp. 3733–3742.

[48] F. Schroff, D. Kalenichenko, and J. Philbin, "Facenet: A unified embedding for face recognition and clustering," in *IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2015, Boston, MA, USA, June 7-12, 2015*. IEEE Computer Society, 2015, pp. 815–823. [Online]. Available: https://doi.org/10.1109/CVPR.2015.7298682

[49] A. R. Chen, T. P. Chen, and J. Chen, "How useful is code change information for fault localization in continuous integration?" in *37th IEEE/ACM International Conference on Automated Software Engineering, ASE 2022, Rochester, MI, USA, October 10-14, 2022*. ACM, 2022, pp. 52:1–52:12. [Online]. Available: https://doi.org/10.1145/3551349.3556931

[50] J. Wang and J. Zhu, "Portfolio theory of information retrieval," in *Proceedings of the 32nd Annual International ACM SIGIR Conference on Research and Development in Information Retrieval, SIGIR 2009, Boston, MA, USA, July 19-23, 2009*, J. Allan, J. A. Aslam, M. Sanderson, C. Zhai, and J. Zobel, Eds. ACM, 2009, pp. 115–122. [Online]. Available: https://doi.org/10.1145/1571941.1571963

[51] M. Fey and J. E. Lenssen, "Fast graph representation learning with PyTorch Geometric," in *ICLR Workshop on Representation Learning on Graphs and Manifolds*, 2019.

[52] P. F. Russell, T. R. Rao *et al.*, "On habitat and association of species of anopheline larvae in south-eastern madras." *Journal of the Malaria Institute of India*, vol. 3, no. 1, 1940.

[53] U. Hamann, "Merkmalsbestand und verwandtschaftsbeziehungen der farinosae: ein beitrag zum system der monokotyledonen," *Willdenowia*, pp. 639–768, 1961.

[54] T. Sorenson, "A method of establishing groups of equal amplitude in plant sociology based on similarity of species content, and its application to analysis of vegetation on danish commons," *Kong Dan Vidensk Selsk Biol Skr*, vol. 5, pp. 1–5, 1948.

[55] L. R. Dice, "Measures of the amount of ecologic association between species," *Ecology*, vol. 26, no. 3, pp. 297–302, 1945.

[56] "Falcon website," 2024. [Online]. Available: https://github.com/pppppkun/falcon

[57] J. M. Johnson and T. M. Khoshgoftaar, "Survey on deep learning with class imbalance," *J. Big Data*, vol. 6, p. 27, 2019.

[58] K. Cao, C. Wei, A. Gaidon, N. Aréchiga, and T. Ma, "Learning imbalanced datasets with label-distribution-aware margin loss," in *NeurIPS*, 2019, pp. 1565–1576.

[59] F. Wilcoxon, "Individual comparisons by ranking methods," in *Breakthroughs in statistics: Methodology and distribution*. Springer, 1992, pp. 196–202.

[60] O. J. Dunn, "Multiple comparisons among means," *Journal of the American statistical association*, vol. 56, no. 293, pp. 52–64, 1961.

[61] T. N. Kipf and M. Welling, "Semi-supervised classification with graph convolutional networks," in *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*. OpenReview.net, 2017. [Online]. Available: https://openreview.net/forum?id=SJU4ayYgl

[62] W. L. Hamilton, Z. Ying, and J. Leskovec, "Inductive representation learning on large graphs," in *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA*, I. Guyon, U. von Luxburg, S. Bengio, H. M. Wallach, R. Fergus, S. V. N. Vishwanathan, and R. Garnett, Eds., 2017, pp. 1024–1034.

[63] P. Velickovic, G. Cucurull, A. Casanova, A. Romero, P. Liò, and Y. Bengio, "Graph attention networks," in *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings*. OpenReview.net, 2018. [Online]. Available: https://openreview.net/forum?id=rJXMpikCZ

[64] Z. Wu, S. Pan, F. Chen, G. Long, C. Zhang, and P. S. Yu, "A comprehensive survey on graph neural networks," *IEEE Trans. Neural Networks Learn. Syst.*, vol. 32, no. 1, pp. 4–24, 2021. [Online]. Available: https://doi.org/10.1109/TNNLS.2020.2978386

[65] A. Radford, J. W. Kim, C. Hallacy, A. Ramesh, G. Goh, S. Agarwal, G. Sastry, A. Askell, P. Mishkin, J. Clark, G. Krueger, and I. Sutskever, "Learning transferable visual models from natural language supervision," in *Proceedings of the 38th International Conference on Machine Learning, ICML 2021, 18-24 July 2021, Virtual Event*, ser. Proceedings of Machine Learning Research, M. Meila and T. Zhang, Eds., vol. 139. PMLR, 2021, pp. 8748–8763. [Online]. Available: http://proceedings.mlr.press/v139/radford21a.html

[66] Z. Yang, Y. Cheng, Y. Liu, and M. Sun, "Reducing word omission errors in neural machine translation: A contrastive learning approach," in *Proceedings of the 57th Conference of the Association for Computational Linguistics, ACL 2019, Florence, Italy, July 28- August 2, 2019, Volume 1: Long Papers*, A. Korhonen, D. R. Traum, and L. Màrquez, Eds. Association for Computational Linguistics, 2019, pp. 6191–6196. [Online]. Available: https://doi.org/10.18653/v1/p19-1623

[67] T. Gao, X. Yao, and D. Chen, "Simcse: Simple contrastive learning of sentence embeddings," in *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing, EMNLP 2021, Virtual Event / Punta Cana, Dominican Republic, 7-11 November, 2021*, M. Moens, X. Huang, L. Specia, and S. W. Yih, Eds. Association for Computational Linguistics, 2021, pp. 6894–6910. [Online]. Available: https://doi.org/10.18653/v1/2021.emnlp-main.552

[68] P. Jain, A. Jain, T. Zhang, P. Abbeel, J. Gonzalez, and I. Stoica, "Contrastive code representation learning," in *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing, EMNLP 2021, Virtual Event / Punta Cana, Dominican Republic, 7-11 November, 2021*, M. Moens, X. Huang, L. Specia, and S. W. Yih, Eds. Association for Computational Linguistics, 2021, pp. 5954–5971. [Online]. Available: https://doi.org/10.18653/v1/2021.emnlp-main.482