Pre-training Code Representation with Semantic Flow Graph for Effective Bug Localization

Yali Du
Shandong University, China
duyali2000@gmail.com

Zhongxing Yu*
Shandong University, China
zhongxing.yu@sdu.edu.cn

ABSTRACT

Enlightened by the big success of pre-training in natural language processing, pre-trained models for programming languages have been widely used to promote code intelligence in recent years. In particular, BERT has been used for bug localization tasks and impressive results have been obtained. However, these BERT-based bug localization techniques suffer from two issues. First, the pre-trained BERT model on source code does not adequately capture the deep semantics of program code. Second, the overall bug localization models neglect the necessity of large-scale negative samples in contrastive learning for representations of changesets and ignore the lexical similarity between bug reports and changesets during similarity estimation. We address these two issues by 1) proposing a novel directed, multiple-label code graph representation named Semantic Flow Graph (SFG), which compactly and adequately captures code semantics, 2) designing and training SemanticCodeBERT based on SFG, and 3) designing a novel Hierarchical Momentum Contrastive Bug Localization technique (HMCBL). Evaluation results show that our method achieves state-of-the-art performance in bug localization.

CCS CONCEPTS

• **Software and its engineering** → **Software testing and debugging**; **Maintaining software**.

KEYWORDS

bug localization, semantic flow graph, type, computation role, pre-trained model, contrastive learning

ACM Reference Format:
Yali Du and Zhongxing Yu. 2023. Pre-training Code Representation with Semantic Flow Graph for Effective Bug Localization. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '23), December 3–9, 2023, San Francisco, CA, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3611643.3616338

*Zhongxing Yu is the corresponding author.

1 INTRODUCTION

While modern software engineering recognizes a broad range of methods (e.g., model checking, symbolic execution, type checking) for helping ensure that the software meets the specification of its desirable behavior, the software (even deployed ones) is still unfortunately plagued with heterogeneous bugs for reasons such as programming errors made by developers and immature development process. The process of resolving the resultant bugs termed debugging, is an indispensable yet frustrating activity that can easily account for a significant part of software development and maintenance costs [76]. To tackle the ever-growing high costs involved in debugging, a variety of automatic techniques have been proposed as debugging aids for developers over the past decades [88]. In particular, numerous methods have been developed to facilitate fault localization, which aims to identify the exact locations of program bugs and is one of the most expensive, tedious, and time-consuming activities in debugging [80, 85].

The literature on fault localization is rich and is abundant with methods stemming from ideas that originate from several different disciplines, notably including statistical analysis [37, 48, 84, 86], program transformation [60], information retrieval [68, 71]. Among them, information retrieval-based methods typically proceed by establishing the relevance between bug reports and related software artifacts on the ground of information retrieval techniques, and this category of methods is appealing as it is amenable to the mainstream development practice which features continuous integration (CI), versioning with Git, and collaboration within platforms like GitHub [75]. In line with existing literature, information retrieval-based fault localization hereafter is simply referred to as bug localization.

The matched software artifact at the early phase of bug localization research focuses on code elements such as classes and methods [41, 72], but recent years have witnessed a growing interest in changesets [18, 68, 79, 81]. The key advantage of changesets is that they contain simultaneously changed parts of the code that are related, facilitating bug fixing. With regard to information retrieval techniques, the major shift is that the dominating techniques have changed from Vector Space Model (VSM) to deep learning techniques, both for code elements and changesets. To precisely locate the bug, bug localization techniques essentially need to accurately relate the natural language used to describe the bug (in the bug report) and identifier naming practices adopted by developers (in the software artifacts). However, it is quite common that there exists a significant lexical gap between them, and consequently, the retrieval quality of bug localization techniques is not always satisfactory [96]. To overcome the issue, bug localization techniques necessarily need to go beyond exact term matching and establish the semantic relatedness between bug reports and software artifacts.

Given that deep learning architectures are capable of leveraging contextual information and have achieved impressive progress in natural language processing, a number of bug localization techniques based on the neural network have been proposed in recent years [17, 32, 44, 55, 57, 64, 83, 94, 95]. In particular, state-of-the-art transformer-based architecture BERT [21] (bidirectional encoder representation from the transformer) has been widely employed [17, 47]. Based on the naturalness hypothesis which states that "*software corpora have similar statistical properties to natural language corpora*" [29], these BERT-based techniques first pre-train a BERT model on a massive corpus of source code using certain pre-training tasks such as masked language modeling, and then fine-tune the trained BERT model for bug localization task. Experimental evaluations have shown that reasonable accuracy improvements can be obtained by these BERT-based techniques.

Despite the progress made, one drawback of these BERT-based techniques is that the pre-trained BERT model on source code does not adequately capture the deep semantics of program code. Unlike natural language, the programming language has a formal structure, which provides important code semantics that is unambiguous in general [2]. However, the existing pre-trained BERT model either totally ignores the code structure by treating code snippet as a sequence of tokens same as natural language or considers only the shallow structure of the code by using graph code representations such as data flow graph [25]. Consequently, the formal code structure has not been fully exploited, resulting in an under-optimal BERT model. To overcome this issue, we in this paper present a novel code graph representation termed Semantic Flow Graph (SFG), which compactly and adequately captures code semantics. SFG is a directed, multiple-label graph that captures not only the data flow and control flow between program elements but also the type of program element and the specific role that a certain program element plays in computation. On the ground of SFG, we further propose SemanticCodeBERT, a pre-training model with BERT-like architecture to learn code representation that considers deep code structure. SemanticCodeBERT features novel pre-training tasks besides the ordinary masked language modeling task.

In addition, the overall models of existing BERT-based bug localization techniques ignore several points which are beneficial for further improving performance. First, the batch size is typically limited to save model space because of the huge scale of BERT parameters, and the number of negative samples coupled to batch size is thus limited. A variety of existed methods [9, 11, 13–16, 22, 28, 39, 45, 65, 66, 74, 78, 82, 89] emphasizes the necessity of large-scale negative samples in contrastive representation learning. In the bug localization context, it implies the importance of considering the large-scale negative sample interactions for representation learning of bug reports and changesets. Nevertheless, existing techniques like Ciborowska *et. al.* [17] only select one irrelevant changeset in training as the negative sample for a bug report, which causes inefficient mining of negative samples and poor representation of the programming language. To alleviate this issue, we propose to use a memory bank [82] to store rich changesets obtained from different batches for later contrast. In particular, due to the constant parameter update by back-propagation, we utilize the momentum contrastive method [28] to account for the inconsistency of negative vectors obtained by different models (in different mini-batches). Second, existing BERT-based bug localization techniques only account for the semantic level similarity between bug reports and changesets, totally ignoring the lexical similarity (*e.g.*, same identifier) which is also of vital importance for retrieval if exists. To alleviate this issue, we propose to use a hierarchical contrastive loss to leverage similarities at different levels. On the whole, we design a novel Hierarchical Momentum Contrastive Bug Localization (HMCBL) technique to address the two limitations.

We implement the analyzer for obtaining SFG for Java code and use the Java corpus (including 450,000 functions) of the CodeSearchNet dataset [33] to pre-train SemanticCodeBERT. On top of SemanticCodeBERT, we apply the hierarchical momentum contrastive method to facilitate the retrieval of bug-inducing changesets given a bug report on the widely used dataset established in [79], which includes six Java projects. Results show that we achieve state-of-the-art performance on bug localization. Ablation studies justify that the newly designed SFG improves the BERT model and the new bug localization architecture is better than the existing ones.

Our contributions can be summarized as follows:

- We present a novel directed, multiple-label code graph representation termed Semantic Flow Graph (SFG), which compactly and adequately captures code semantics.
- We employ SFG to train SemanticCodeBERT, which can be applied to obtain code representations for various code-related downstream tasks.
- We design a novel Hierarchical Momentum Contrastive Bug Localization technique (HMCBL), which overcomes two important issues of existing techniques.
- We conduct a large-scale experimental evaluation, and the results show that our method outperforms state-of-the-art techniques in bug localization performance.

2 RELATED WORKS

This section reviews work closely related to this paper. Bug localization techniques proceed by making a query about the relevance between bug reports and related software artifacts on top of information retrieval techniques. The investigated software artifacts can be majorly divided into two categories: code elements such as classes and methods [31, 41, 42, 46, 50–53, 58, 62, 67, 71–73, 90, 94] and changesets [18, 68, 79, 81]. Given changesets contain simultaneously changed parts of the code that are related and can thus facilitate bug fixing, the use of changesets is gradually dominating.

With regard to information retrieval techniques, the Vector Space Model (VSM) is widely used for its simplicity and effectiveness, especially in the early phase of bug localization research. For instance, BugLocator [91] makes use of the revised Vector Space Model (rVSM) to establish the textual similarity between the bug report and the source code and then ranks all source code files based on the calculated similarity. For another example, Locus [79] represents one of the earliest works on changeset-based bug localization, and it proceeds by matching bug reports to hunks.

As VSM basically performs exact term matching, the effectiveness will be compromised in the common case where there exists a significant lexical gap between the descriptions in the bug report and naming practices adopted by developers in the software artifacts. To overcome this issue, bug localization techniques essentially need to establish the semantic relatedness between bug reports and software artifacts. Given the impressive progress in leveraging contextual information by deep learning architectures in natural language processing, deep neural networks have been widely used by researchers to learn representations for bug localization in recent years [32, 44, 57, 93]. For instance, Huo *et. al.* [32] present the Deep Transfer Bug Localization task, and propose the TRANP-CNN as the first solution for the cold-start problem which combines cross-project transfer learning and convolutional neural networks for file-level bug localization. Zhu *et. al.* [93] focus on transferring knowledge (while filtering out irrelevant noise) from the source project to the target project, and propose the COOBA to leverage adversarial transfer learning for cross-project bug localization. Murali *et. al.* [57] propose Bug2Commit, which is an unsupervised model leveraging multiple dimensions of data associated with bug reports and commits.

In particular, enlightened by the impressive achievements made by BERT in natural language processing, BERT has been used for bug localization tasks. Lin *et. al.* [47] study the tradeoffs between different BERT architectures for the purpose of changeset retrieval. Based on the Colbert developed by Khattab *et. al.* [40], Ciborowska *et. al.* [17] propose the FBL-BERT model towards changeset-based bug localization. Evaluation results show that FBL-BERT can speed up the retrieval and several design decisions have also been explored, including granularities of input changesets and the utilization of special tokens for capturing changesets' semantic representation. While impressive retrieval results of changesets have been achieved, the Colbert used by FBL-BERT does not adequately capture the deep semantics of program code and the overall models of FBL-BERT suffer from two important limitations as described in Section 1 (Introduction).

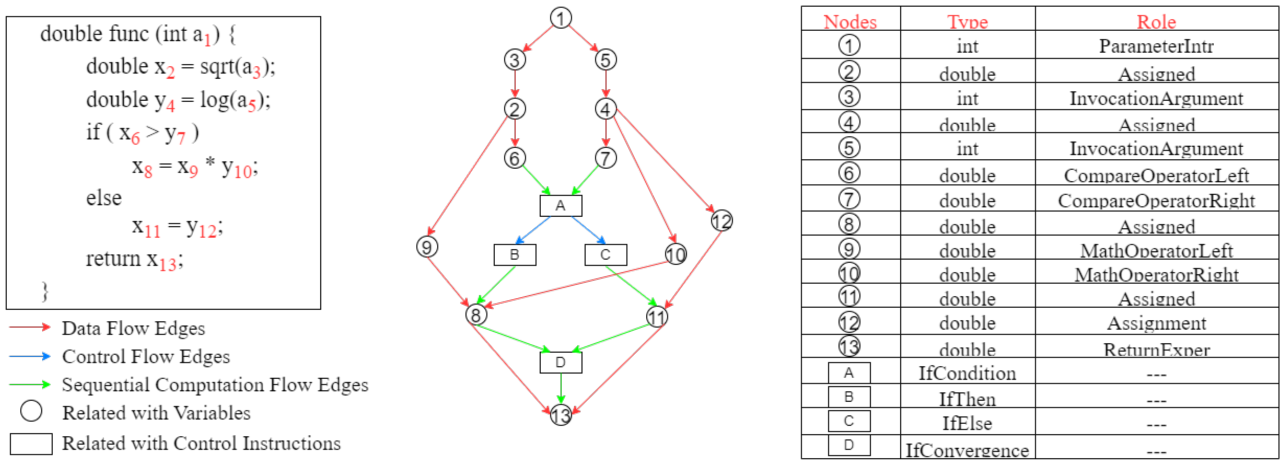
Furthermore, inspired by the success of pre-training models in natural language processing, a number of pre-trained models for programming languages have been proposed to promote the development of code representation (which is vital for a variety of code-based tasks in the field of SE). For instance, CodeBERT is a pre-trained model proposed by Feng *et. al.* [23], which provides generic representations for natural and programming language downstream applications. GraphCodeBERT [25] imports structural information to enhance the code representation by adding the data flow graph as an auxiliary of input tokens and improves the performance of code representation compared to CodeBERT. Kanade *et. al.* [38] propose CuBERT, which is pre-trained on a massive Python source corpus with two pre-training tasks of Masked Language Modeling (MLM) and Next Sentence Prediction (NSP). Buratti *et. al.* [10] propose C-BERT, a transformer-based language model that is pre-trained on the C language corpus for code analysis tasks. Xue *et. al.* [35] propose TreeBERT, which proposes a hybrid target for AST to learn syntactic and semantic knowledge with tree-masked language modeling (TMLM) and node order prediction (NOP) pre-training tasks. More recently, the UniXcoder [24] is proposed to leverage cross-modal information like Abstract Syntax Tree and comments written in natural language to enhance code representation. While these pre-trained models on source code have made progress towards code representation, one drawback of them is that they not adequately capture the deep semantics of program code as they either treat code snippets as token sequences or consider only shallow code structure by using graph code representations such as data flow graph. Hence, we give a novel code graph representation termed Semantic Flow Graph (SFG) to more compactly and adequately capture code semantics in this paper. On top of SFG, we further design and train SemanticCodeBERT with novel pre-training tasks.

3 SEMANTIC FLOW GRAPH

This section introduces the Semantic Flow Graph (SFG), a novel code graph representation designed for compactly representing deep code semantics. On top of the naturalness hypothesis "*Software is a form of human communication, software corpora have similar statistical properties to natural language corpora, and these properties can be exploited to build better software engineering tools*" [29], recent years have witnessed many innovations in using machine learning (particularly deep learning) techniques to help make the software more reliable and maintainable. To achieve successful learning, one important ingredient lies in suitable code representation. The representation, on the one hand, should capture enough code semantics, and on the other hand, should be learnable across code written by different developers or even different programming languages [2].

There are majorly three categories of code representation ways within the literature: token-based ways that represent code as a sequence of tokens [1, 19, 26, 27], syntactic-based ways that represent code as trees [4–6, 30, 56, 63, 87], and semantic-based ways that represent code as graph structures [3, 7, 8, 12, 20, 25, 43, 92]. For token-based representation, while its simplicity facilities learning, the representation ignores the structural nature of code and thus captures quite limited semantics. For syntactic-based representation, despite the tree representation in principle can contain rich semantic information, the learnability is unfortunately confined as the tree typically has an unusually deep hierarchy and there, in general, will involve significant refinement efforts of the raw tree representation to enable successful learning in practice. Semantic-based representation aims to encode semantics in a way that facilitates learning, and a variety of graphs have been employed for code model learning, including for example data flow graph [12, 25], control flow graph [20], program dependence graph [8], contextual flow graph [7].

While these graph-based representations have facilitated the learning of code semantics embodied in data dependency and control dependency, certain other code semantics are overlooked. In particular, the information of *what* kinds of program elements are related by data dependency or control dependency and through *which* operations they are related to is neglected. We argue that this information is crucial for accurately learning code semantics. For instance, given a code snippet "`a = m (b, c)`" where `a`, `b`, and `c` are `Boolean`, `Integer`, and User-defined type variables respectively, and `m` is a certain function call, there will be two data flow edges `b` $\longrightarrow$ `a` and `c` $\longrightarrow$ `a` considering the data flow graph, and the code snippet will read "*the values of two variables have flown into another variable*". Under this circumstance, as the corresponding data flow graph coincides, the meaning of the code snippet has no



```
double func (int a1) {
        double x2 = sqrt(a3);
        double y4 = log(a5);
        if ( x6 > y7 )
                x8 = x9 * y10;
        else
                x11 = y12;
        return x13;
}
```

Nodes	Type	Role
1	int	ParameterIntr
2	double	Assigned
3	int	InvocationArgument
4	double	Assigned
5	int	InvocationArgument
6	double	CompareOperatorLeft
7	double	CompareOperatorRight
8	double	Assigned
9	double	MathOperatorLeft
10	double	MathOperatorRight
11	double	Assigned
12	double	Assignment
13	double	ReturnExper
A	IfCondition	---
B	IfThen	---
C	IfElse	---
D	IfConvergence	---

Figure 1: An example of the semantic flow graph.

difference with a variety of other code snippets such as "`a = (b && m (c))`" where a, b, and c are `Boolean`, `Boolean`, and arbitrary type variables respectively, and `m` is a certain function call that returns a boolean value. But if the additional information of *what* kinds of program elements and *which* operations are taken into account, the code snippet "`a = m (b, c)`" will read "*the value of an Integer type variable and the value of a User-defined type variable have flown into another Boolean type variable through a function call*", which is more precise. To compactly integrate these two pieces of information into graphs, we design a novel directed, multiple-label code graph representation termed Semantic Flow Graph (SFG).

Definition 3.1. (Semantic Flow Graph). The Semantic Flow Graph (SFG) for a code snippet is a tuple $< N, E, T, R >$ where N is a set of nodes, E is a set of directed edges between nodes in N, and T and R are mappings from nodes to their types and their roles in computation respectively.

A number of points deserve comment. First, the node set N can be further divided into node sets N_V and N_C, which contain nodes corresponding to variables and control instructions in the code respectively. While the variable has a one-to-one mapping with a certain node from N_V, there may be one or multiple nodes in N_C for a certain control instruction. Essentially, if a control instruction has an associated condition and n different branches (i.e., straight-line code blocks) to go depending on the condition evaluation result, there will be a node in N_C for the condition, a node in N_C for the convergence of the different branches, and n different nodes in N_C for the n branches respectively.

Second, a directed edge $n_a \longrightarrow n_b$ ($n_a \in N, n_b \in N$) in E can be of 3 kinds. The first kind E_D represents a data flow between two variables if $n_a \in N_V \wedge n_b \in N_V$ holds, the second kind E_C embodies the control flow between two straight-line basic blocks if $n_a \in N_C \wedge n_b \in N_C$ holds, and finally the third kind E_S denotes the natural sequential computation flow inside or between basic blocks in case $n_a \in N_V \wedge n_b \in N_C$ or $n_a \in N_C \wedge n_b \in N_V$ holds. In particular, the edge set E is established as follows:

(1) Establish E_D among nodes from set N_V according to Intra-block and Inter-block data dependencies between variables.

(2) Establish E_C among nodes from set N_C according to the specific control flow of the control instruction.

(3) Establish E_S following these rules: there will be an edge $n_a \longrightarrow n_b$ (i) if $n_b \in N_C$ is for the control instruction condition and $n_a \in N_V$ is for a certain variable involved in the condition; (ii) if $n_a \in N_C$ is for the control instruction branch and $n_b \in N_V$ is for the left-most variable of the first statement inside the branch; (iii) if $n_a \in N_V$ is for the left-most variable of the last statement inside a control instruction branch and $n_b \in N_C$ is for the control instruction convergence; (IV) if $n_a \in N_C$ is for the control instruction convergence and $n_b \in N_V$ is for the left-most variable of the first statement inside the basic block directly following the control instruction.

Third, mapping T maps each node in N to its type, encoding the needed information of "*what* kinds of program elements are related". For each node in N_V, T maps it to the corresponding type of the variable. For each node in N_C, T maps the node to the specific part of the control instruction it refers to. Take the control instruction `If-Then-Else` as an example, T maps the associated 4 nodes in N_C for it to type `IfCondition`, `IfThen`, `IfElse`, and `IfCONVERGE` respectively.

Finally, mapping R maps each node in N_V to its role in the computation, encoding the needed information of "through *which* operations program elements are related". Basically, R considers the associated operation and control structure for the variable to determine its computation role. From an implementation perspective, for each node in N_V, R checks the direct parent of the corresponding variable in the abstract syntax tree (AST) and the position relationship between it and the direct parent to establish the role. For instance, given a code snippet, "`a = b`" where a and b are variables, R maps the roles of `a` and `b` to `Assigned` and `Assignement` respectively. For another example, given a code snippet "`a.m(b)`" where a and b are variables, and `m` is a certain function call, R maps the roles of `a` and `b` to `InvocationTarget` and `InvocationArgument` respectively. For nodes in N_C, we do not consider their roles as they are implicit in their types.

Note it is difficult to simply augment classical graph program representations with information of type and computation role. Existing representations like program dependence graph typically work at the statement granularity (i.e., each graph node represents a statement), making it hard to encode detailed type and computation role information of multiple program elements in the statement. The proposed SFG works at a finer granularity with two types of nodes that have a one-to-one mapping with program variables and a one-to-one (or many-to-one) mapping with program control ingredients respectively. This kind of node representation is proposed for two reasons. On the one hand, it is convenient to analyze the types and computation roles of variables (through how they are connected with other program elements) and program control ingredients. On the other hand, data flow and control flow information are established respectively by analyzing variable uses and program control ingredients. With SFG (built on such a node representation), data flow and control flow can be encoded through the edges between nodes, and the type and computation role information can be encoded through node labels. SFG does not have nodes for additional program elements (like invocation etc.), thus it is compact but contains adequate semantic information.

Overall, SFG is a directed, multiple-label graph that captures not only the data flow and control flow between program elements, but also the type of program element and the specific role that a certain program element plays in computation. Moreover, SFG represents this information in a compact way, facilitating learning across programs.

Example 3.1. *Figure 1 gives an example of a Semantic Flow Graph for a simple method.*

Implementation: We fully implement an analyzer to get Semantic Flow Graph (SFG) for a Java method on top of Spoon [61], which is an open-source library to analyze, rewrite, transform, and transpile Java source code. Our analyzer supports modern Java versions up to Java 16. For nodes in N_V, the analyzer considers different kinds of primitive types and common JDK types, and a special type named `user-defined type`. In total, the analyzer considers 20 types for nodes in N_V. For nodes in N_C, the analyzer takes all the control instruction kinds (up to Java 16) into account and considers 35 types in total. With regard to role, the analyzer considers 43 different roles in total for nodes in N_V.

4 SEMANTICCODEBERT

In this section, we describe first the architecture of SemanticCodeBERT (shown in Figure 3), then the graph-guided masked attention based on the semantic flow graph, and finally the pre-training tasks. Overall, the SemanticCodeBERT network architecture adapts the architecture of GraphCodeBERT for the proposed novel SFG program representation and SemanticCodeBERT also features tailored pre-training tasks for the SFG representation.

4.1 Model Architecture

The SemanticCodeBERT follows BERT (Bidirectional Encoder Representation from Transformers) (Devlin *et. al.,* [21]) as the backbone.

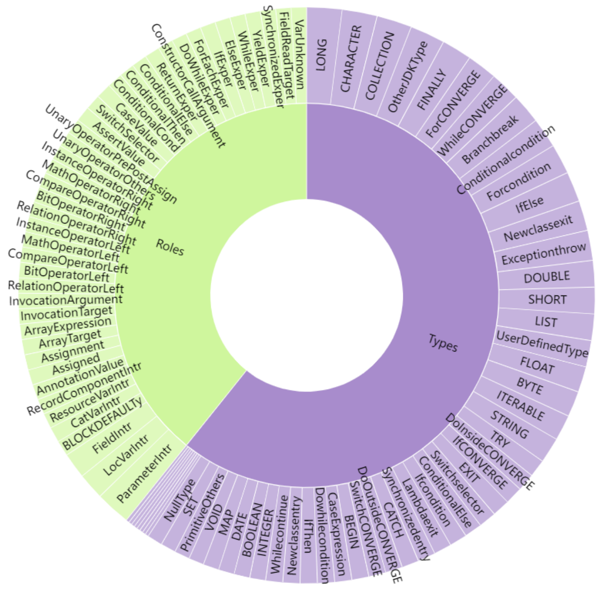


Figure 2: All defined types and roles.

Comment Input Sequence: We import comments as a supplement for the model to understand the semantic information of programming code. $[CLS]$ is the special classification token at the beginning of the comment sequence W.

Source Code Input Sequence: We cleanse the source code and remove erroneous characters, and add the special classification token $[SEP]$ at the end of the source code and input sequence. To represent the start-of-code, we import a pre-appended token $[C]$ to split the comment and source code. The source code sequence can be represented as S.

Node Input Sequence: With the procedure discussed in Section 3, we generate a semantic flow graph (SFG) for each code snippet. At the beginning of the node list N, a pre-appended token $[N]$ is added to represent the start-of-node.

Type Input Sequence: To answer the question of "*what kinds of program elements are related*", we have identified 55 possible types for the code element. $T = \{t_1, \ldots, t_{55}\}$ represents the set of all 55 possible types, and $[T]$ is pre-appended as the start-of-type. The complete list of types is shown in Figure 2.

Role Input Sequence: To answer the question of "*through which operations program elements are related*", we have defined 43 roles to mark the role of each program element in the computation, taking into account the associated operation and control structure. $R = \{r_1, \ldots, r_{43}\}$ is the set of all 43 possible roles, and the pre-appended token $[R]$ represents the start-of-role. The complete list of roles is shown in Figure 2.

As intuitively shown in Figure 3, we concatenate the comment, source code, nodes, types, and roles as the input sequence:

$$X = Concat[[CLS], W, [C], S, [SEP], [N], N, [T], T, [R], R, [SEP]]. \tag{1}$$

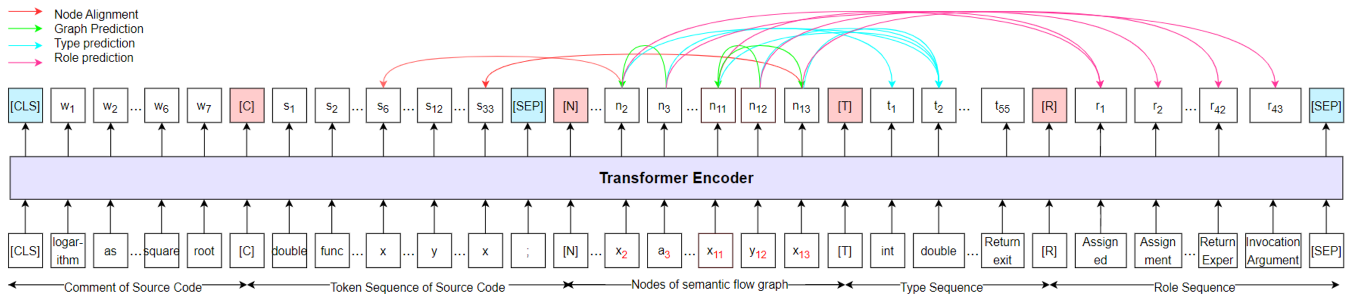


Figure 3: The SemanticCodeBERT takes to comment, source code, nodes of SFG, types, and roles as the input, and is pre-trained by standard masked language modeling [21], node alignment (marked with red lines), graph prediction (marked with green lines), type prediction (marked with blue lines) and role prediction (marked with purple lines).

4.2 Masked Attention

We resort to the graph-guided masked attention function described in [25] to filter irrelevant signals in Transformer.

- The set E^1 indicates the alignment relation between S and N, where $(s_i, n_j)/(n_j, s_i) \in E^1$ if the node n_j is identified from the source code token s_i.
- The set E^2 indicates the dependency relation in N, where $(n_i, n_j) \in E^2$ if there is a direct edge from the node n_i to the node n_j.
- The set E^3 incorporates the type information of the nodes, where $(n_i, t_j) \in E^3$ if the type of the node n_i is t_j.
- The set E^4 incorporates the role information of the nodes, where $(n_i, r_j) \in E^4$ if the role of the node n_i is r_j.

The masked attention matrix is formulated as M:

$$M_{ij} = \begin{cases} 0 & x_i \in [CLS], [SEP]; \\ & w_i, s_j \in W \cup S; \\ & (s_i, n_j)/(n_j, s_i) \in E^1; \\ & (n_i, n_j) \in E^2; \\ & (n_i, t_j) \in E^3; \\ & (n_i, r_j) \in E^4; \\ -\infty & otherwise. \end{cases} \tag{2}$$

Specifically, the masked attention function blocks the transmission of unrelated tokens by setting the attention score to an infinitely negative value.

4.3 Pre-Training Tasks

The pre-training tasks of SemanticCodeBERT are described in this section. Besides masked language modeling, node alignment, and edge prediction pre-training tasks proposed by Guo *et. al.*, [25], we define two novel pre-training tasks–type and role prediction. These two novel pre-training tasks represent the first attempt to leverage the attribute information of nodes for learning code representation.

Masked Language Modeling: The masked language modeling pre-training task is proposed by Devlin *et. al.*, [21]. We replace 15% of the source code with [MASK] 80% of the time, a random token 10% of the time or itself 10% of the time. The comment context contributes to inferring the masked code tokens [25].

Node Alignment: The motivation of node alignment is to align representation between source code and nodes of semantic flow graph [25]. We randomly mask 20% edges between the source code and nodes, and then predict where the nodes are identified from (i.e., predict these masked edges E^1_{mask}). As shown in the Figure 3, the model should distinguish that n_2 comes from s_6 and n_{13} comes from s_{33}. We formulate the loss function as Equation 3. Let E^1 be $S \times N$, $\delta(e_{ij} \in E^1_{mask})$ is one if $(s_i, n_j) \in E^1$, and zero otherwise. $p_{e_{ij}}$ is the probability of the edge from i-th code token to j-th node, which is calculated by dot product following a sigmoid function using the representations of s_i and n_j outputted from SemanticCodeBERT.

$$\mathcal{L}_{NA} = -\sum_{e_{ij} \in E^1_{mask}} [\delta(e_{ij}) log p_{e_{ij}} + (1 - \delta(e_{ij})) log(1 - p_{e_{ij}})]. \tag{3}$$

Edge Prediction: The motivation of edge prediction is to encourage the model to learn structural relationships from semantic flow graphs for better programming code representation. Like node alignment, we randomly mask 20% edges between nodes in the mask matrix, encouraging the model to predict these masked edges E^2_{mask} (*e.g.,* the edges (n_3, n_2) and (n_{12}, n_{11})). We formulate the loss function as Equation 4. Let E^2 be $N \times N$, $\delta(e_{ij} \in E^2_{mask})$ is one if $(n_i, n_j) \in E^2$, and zero otherwise. $p_{e_{ij}}$ is the probability of the edge from i-th node to j-th node.

$$\mathcal{L}_{GP} = -\sum_{e_{ij} \in E^2_{mask}} [\delta(e_{ij}) log p_{e_{ij}} + (1 - \delta(e_{ij})) log(1 - p_{e_{ij}})]. \tag{4}$$

Type Prediction: The motivation of type prediction is to guide the model to comprehend the types (*e.g.,* "int", "double", "IfCondition") of nodes for better programming code representation. We pre-append the full set of types T to the input nodes. Let E^3 be $N \times T$, if the type of node n_i is t_j (*i.e.,* $(n_i, t_j) \in E^3$), $\delta(e_{ij} \in E^3_{mask})$ is one, otherwise it is zero. We randomly mask 20% edges between nodes and types and formulate the loss function as Equation 5, where E^3_{mask} are masked edges and $p_{e_{ij}}$ is the probability of the edge from i-th node to j-th type.

$$\mathcal{L}_{TP} = -\sum_{e_{ij} \in E^3_{mask}} [\delta(e_{ij}) log p_{e_{ij}} + (1 - \delta(e_{ij})) log(1 - p_{e_{ij}})]. \tag{5}$$

Role Prediction: "Role" indicates the computation role of the node in the semantic flow graph (*e.g.,* "InvocationArgument", "Assigned", "Assignment"). Role prediction can feed the model with a more informative signal to understand the correlation among different nodes. We pre-append the full set of roles R to the input nodes. Let E^4 be $N \times R$, if the role of node n_i is r_j (*i.e.,* $(n_i, r_j) \in E^4$), $\delta(e_{ij} \in E^4_{mask})$ is one, otherwise it is zero. We randomly mask 20% edges between nodes and roles and formulate the loss function as Equation 6, where E^4_{mask} are masked edges and $p_{e_{ij}}$ is the probability of the edge from i-th node to j-th role.

$$\mathcal{L}_{RP} = -\sum_{e_{ij} \in E^4_{mask}} [\delta(e_{ij}) log p_{e_{ij}} + (1 - \delta(e_{ij})) log(1 - p_{e_{ij}})]. \quad (6)$$

5 CHANGESET-BASED BUG LOCALIZATION

In this section, we illustrate the utilization of the SemanticCodeBERT towards bug localization with changesets. The proposed bug localization model is shown in Figure 4. The model aims to address the two important limitations (as described in Section 1) of the overall models of existing BERT-based bug localization techniques.

5.1 Problem Definition

Given a set $Q = \{q_1, q_2, \ldots, q_M\}$ of M bug reports, the bug localization task aims to discover more relevant changesets from $\mathcal{K} = \{k_1, k_2, \ldots, k_N\}$, a set including N changesets. More specifically, for a bug report $q \in Q$, a bug-inducing changeset $p \in \mathcal{K}$ and a not bug-inducing changeset $n \in \mathcal{K}$ are selected to form a triplet (q, p, n). All bug-inducing changesets and not bug-inducing changesets are non-overlapping. The goal of learned similarity function s is to provide a high value for $s(q, p)$ (between the anchor q and the positive sample p) and a low value for $s(q, n)$ (between the anchor q and the negative sample n). Section 5.2 focuses on producing accurate representations of bug reports and changesets, and Section 5.3 describes the estimation of similarities and the loss function for training the model.

5.2 Representation Learning

The proposed model consists of three parts, an encoder network, projector network, and momentum update mechanism with a memory bank that stores rich representations of changesets.

Encoder Network: As mentioned before, bug reports consist of natural language descriptions and project changesets consist of programming language code. Hence, we introduce BERT [21] as the backbone to the encoder bug report as $\mathbf{q}_{feature}$, and SemanticCodeBERT as the backbone to encoder relevant changeset as $\mathbf{p}_{feature}$ and irrelevant changeset as $\mathbf{n}_{feature}$.

$$\begin{cases} \mathbf{q}_{feature} = \textbf{BERT}(q_{tok}), \\ \mathbf{p}_{feature} = \textbf{SemanticCodeBERT}(p_{tok}), \\ \mathbf{n}_{feature} = \textbf{SemanticCodeBERT}(n_{tok}), \end{cases} \quad (7)$$

where **BERT** and **SemanticCodeBERT** are the trainable parameters of BERT and SemanticCodeBERT, q_{tok}, p_{tok}, and n_{tok} are the input tokens obtained by tokenizers, $\mathbf{q}_{feature} \in \mathbb{R}^d$, $\mathbf{p}_{feature} \in \mathbb{R}^d$, and $\mathbf{n}_{feature} \in \mathbb{R}^d$ are the refined vectors (d is the dimension of the mapped spaces).

Projector Network: After the feature vectors are extracted, we use a multi-layer perception neural network as a projector to compress the vectors of bug reports and changesets into a compact shared embedding space. We replace Dropout with Batch Normalization for regularization, which can be trained with saturating nonlinearities and are more tolerant to increased training rates [34].

$$\begin{cases} \mathbf{q}_{model} = \mathbf{W}_b^2 norm(\phi(\mathbf{W}_b^1 \mathbf{q}_{feature})), \\ \mathbf{p}_{model} = \mathbf{W}_c^2 norm(\phi(\mathbf{W}_c^1 \mathbf{p}_{feature})), \\ \mathbf{n}_{model} = \mathbf{W}_c^2 norm(\phi(\mathbf{W}_c^1 \mathbf{n}_{feature})), \end{cases} \quad (8)$$

where $\mathbf{q}_{model} \in \mathbb{R}^{d'}$, $\mathbf{p}_{model} \in \mathbb{R}^{d'}$, and $\mathbf{n}_{model} \in \mathbb{R}^{d'}$ are the projected vectors (d' is the dimension of the output of projector), $\mathbf{W}_b^{\cdot}$ and $\mathbf{W}_c^{\cdot}$ are the trainable weight matrices, $norm(\cdot)$ denotes batch normalization [34], and $\phi(\cdot)$ is the $leaky\_relu$ function [54].

Momentum Update Mechanism with Memory Bank: As mentioned in Section 1, it is important to consider large-scale negative samples in contrastive learning for representations of changesets. To account for this, we use memory bank [82] to store rich changesets obtained from different batches for later contrast. In particular, we build the key model for encoder and projector networks of changesets based on the momentum contrastive learning mechanism proposed by He *et. al.* [28]. The parameters of the query model θ^q, are updated by back-propagation, while the parameters of the key model θ^k are momentum updated as follows:

$$\theta^k \leftarrow m\theta^k + (1 - m)\theta^q, \quad (9)$$

where $m \in [0, 1)$ is a pre-defined momentum coefficient, which is set as 0.999 in our experiment. As proved in the previous study [28], a relatively large momentum works much better than a smaller value suggesting that a slowly evolving key model is core to making use of the memory bank. For per mini-batch, we use average pooling and enqueue the latest negative samples into the memory bank and dequeue the oldest negative samples.

5.3 Similarity Estimation

As mentioned before, the lexical similarity between bug reports and program changesets like the same application programming interfaces is also crucial for retrieval besides semantic similarity. In this paper, we use the hierarchical contrastive loss to leverage the lower feature-level similarity, higher model-level similarity, and broader bank-level similarity for matching the bug report with relevant changesets. We get the positive feature-level similarity s^{f+} by calculating cosine similarity between $q_{feature}$ and $p_{feature}$, the negative feature-level similarity s^{f-} by calculating cosine similarity between $q_{feature}$ and $n_{feature}$, the positive model-level similarity s^{m+} by calculating cosine similarity between q_{model} and p_{model}, and the negative model-level similarity s^{m-} by calculating cosine similarity between q_{model} and n_{model}. Specifically, we calculate the positive bank-level similarity s^{b+} as cosine similarity between $\mathbf{q}_{query}$ and $\mathbf{p}_{key}$, and the negative bank-level similarity s_i^{b-} ($i \in \{1, 2, \ldots, K\}$) as cosine similarity between $\mathbf{q}_{query}$ and i-th negative sample $\mathbf{n}^i_{key}$ of the memory bank (K is the size of memory bank).

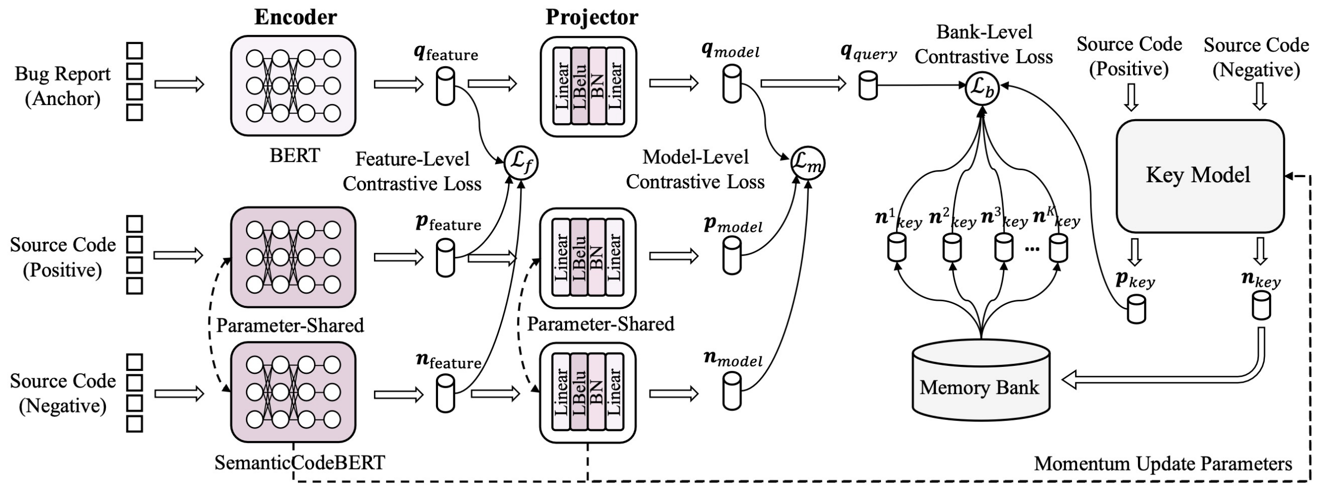


Figure 4: An overview of the Hierarchical Momentum Contrastive Bug Localization technique (HMCBL).

We adopt InfoNCE [59], a form of contrastive loss functions, as our objective function for contrastive matching. The feature-level contrastive loss is formulated as follows:

$$\mathcal{L}_f = -log\frac{exp(s^{f+}/\gamma)}{exp(s^{f+}/\gamma) + exp(s^{f-}/\gamma)}. \tag{10}$$

The model-level contrastive loss is formulated as follows:

$$\mathcal{L}_m = -log\frac{exp(s^{m+}/\gamma)}{exp(s^{m+}/\gamma) + exp(s^{m-}/\gamma)}. \tag{11}$$

The bank-level contrastive loss is formulated as follows:

$$\mathcal{L}_b = -log\frac{exp(s^{b+}/\gamma)}{exp(s^{b+}/\gamma) + \sum_{k=1}^{K} exp(s_i^{b-}/\gamma)}, \tag{12}$$

where K is the size of the memory bank and γ is a temperature hyper-parameter that is set to be 0.07 in our experiment. Thus, the overall objective function is $\mathcal{L}$:

$$\mathcal{L} = \alpha_f\mathcal{L}_f + \alpha_m\mathcal{L}_m + \alpha_b\mathcal{L}_b, \tag{13}$$

where α_f, α_m, and α_b are three hyper-parameters to balance the feature-level, model-level, and bank-level contrasts.

5.4 Offline Indexing and Retrieval

After fine-tuning the model on a project-specific dataset, we resort to the offline indexing and retrieval methods proposed by Ciborowska *et. al.* [17]. All encoded changesets are stored in IVFPQ (InVert File with Product Quantization) index. The IVFPQ index is implemented using the Faiss library [36], which uses the k-means algorithm to partition the embedding space into programmed partitions and assign each embedding to its nearest cluster. In the retrieval process, the query bug report is first located to the nearest partition's centroid, and then the nearest instance within the partition is discovered. For each query bug report, we can identify the N' most similar changesets across all N changesets stored in the IVFPQ index. Therefore, we only re-rank the top-N' subset as the candidate changesets to produce the final ranking.

Table 1: Six projects used for evaluation.

Dataset	Bugs	Changesets		
		Commits	Files	Hunks
AspectJ	200	2,939	14,030	23,446
JDT	94	13,860	58,619	150,630
PDE	60	9,419	42,303	100,373
SWT	90	10,206	25,666	69,833
Tomcat	193	10,034	30,866	72,134
ZXing	20	843	2,846	6,165

6 EXPERIMENTAL EVALUATION

6.1 Dataset

The SemanticCodeBERT is trained using all the Java corpus in CodeSearchNet [33], and we provide the weights and the guidance to fine-tune the pre-trained model for downstream tasks. To evaluate our bug localization technique, we use the dataset separated by Ciborowska *et. al.* [17] from the manually validated dataset by Ming *et. al.* [79]. The dataset includes six software projects, termed AspectJ, JDT, PDE, SWT, Tomcat, and ZXing, as shown in Table 1. To explore the impact of the granularity of changeset data, the bug-inducing changeset is further divided into file-level and hunk-level code changes. Thus, one bug report can have multiple pairs with files or hunks from the original inducing changes. In total, we consider three different granularities: commits, files, and hunks.

6.2 Evaluation Metrics

A set of metrics commonly used to evaluate the performance of information retrieval systems are applied to evaluate the performance of different models.

Precision@K (*P@K*): *P@K* evaluates how many of the top-*K* changesets in a ranking are relevant to the bug report, which is equal to the number of the relevant changesets $|Rel_{B_i}|$ located in the top-K position in the ranking averaged across B bug reports:

$$P@K = \frac{1}{|B|}\sum_{i=1}^{|B|}\frac{|Rel_{B_i}|}{K}. \tag{14}$$

Mean Average Precision (*MAP*): *MAP* quantifies the ability of a model to locate all changesets relevant to a bug report. *MAP* is calculated as the mean of *AvgP* (average precision) of B bug reports.

$$AvgP = \sum_{j=1}^{M}\frac{P@j \times pos(j)}{N}. \tag{15}$$

$$MAP = \frac{1}{|B|}\sum_{i=1}^{|B|}\frac{1}{AvgP_{B_i}}, \tag{16}$$

where j is the rank, M is the number of retrieved changesets, $pos(j)$ denotes whether j-th changeset is relevant to the bug report, N is the total number of bug reports relevant to changesets, $P@j$ is the precision of top-j position in the ranking of this retrieval, and B_i is the i-th bug report.

Mean Reciprocal Rank (*MRR*): *MRR* quantifies the ability of a model to locate the first relevant changeset to a bug report, and is calculated as the average of reciprocal ranks across B bug reports. $1stRank_{B_i}$ is the reciprocal rank of i-th bug report, which is the inverted rank of the first relevant changeset in the ranking:

$$MRR = \frac{1}{|B|}\sum_{i=1}^{|B|}\frac{1}{1stRank_{B_i}}. \tag{17}$$

6.3 Experimental Setup

Configurations of Pre-Training Tasks: The SemanticCodeBERT is pre-trained on NVIDIA Tesla A100 with 128GB RAM on the Ubuntu system. The Adam optimizer [49] is used to update model parameters with batch size 80 and learning rate 1E-04. To accelerate the training process, the parameters of GraphCodeBERT [25] are used to initialize the pre-training model. The model is trained with 600K batches and costs about 156 hours.

Configurations of Bug Localization: The first half of the project's pairs of bug reports and bug-inducing changesets, ordered by bug opening date, are selected as the training dataset, and the remaining half is left as the test dataset. The experiments are implemented with GPU support. The Adam optimizer [49] is used to update model parameters with learning rate 3E-05. All bug reports and changesets are truncated or padded to their respective length limit. According to the experimental verification, we set the trade-off hyper-parameters α_f, α_m, and α_b as 1, 1, and 1, respectively.

Changeset Encoding Strategies: Changesets are time-ordered sequences recording the software's evolution over time. We build upon the three changeset encoding strategies (D-encoding, ARC-encoding, and ARC_L-encoding) proposed by Ciborowska *et. al.* [17] to encode changesets. D-encoding does not utilize specific characteristics of changeset lines. ARC-encoding divides the lines into three groups with three unique tokens. ARC_L-encoding instead does not group the lines and maintains the ordering of lines within a changeset. These three strategies are based on the output of the git *diff* command, which divides changeset lines into three kinds: added lines, removed lines, and unchanged lines. All code sequences are preprocessed by filtering the intrusive characters (*e.g.,* docstrings, comments) from the original code tokens.

6.4 Retrieval Performance

We compare the performance of our proposed model with the traditional bug localization tool, state-of-the-art changeset-based bug localization approach, and two recent state-of-the-art pre-trained models with the HMCBL framework.

- **BLUiR** [71]: A structured IR-based fault localization tool, which builds AST to extract the program constructs of each source code file and utilizes Okapi BM25 [69] to calculate the similarity between the bug report and the candidate changesets.
- **FBL-BERT** [17]: The state-of-the-art approach for automatically retrieving bug-inducing changesets given a bug report, which uses the popular BERT model to more accurately match the semantics in the bug report text with the bug-inducing changesets.
- **GraphCodeBERT** [25]: A pre-trained model that considers data flow to better encode the relation between variables.
- **UniXcoder** [24]: An unified cross-modal pre-trained model, which leverages cross-modal information like Abstract Syntax Tree and comments to enhance code representation.

For BLUiR, we fully follow the original technical description in [71] (as no open-source implementation is available) to get the results for the evaluation metrics. For FBL-BERT, we use the experimental results provided in [17]. For GraphCodeBERT and UniXcoder, we get the results by replacing the pre-trained model SemanticCodeBERT within the HMCBL framework respectively with GraphCodeBERT and UniXcoder (keeping other configurations the same). Table 2 shows the retrieval performances of different models with different changeset encoding strategies (*i.e., D-*, *ARC-* and ARC_L- encoding) and three granularities (*i.e. Commits*−, *Files*− and *Hunks*− level) on six projects. Limited by space, the best result of the three encoding strategies is shown for each configuration.

The following observations can be obtained from the figure. First, compared with the traditional bug localization method which relies on more direct term matching between a bug report and a changeset, the neural network methods perform better by obtaining semantic representations for the calculation of similarity. Second, our proposed method outperforms the state-of-the-art method (FBL-BERT) by a clear margin. In particular, our proposed bug localization technique improves FBL-BERT by 140.78% to 188.79% in terms of MRR on six projects with *Commits*− level granularity. Third, compared with GraphCodeBERT and UniXcoder, our model using SemanticCodeBERT as a changeset encoder consistently achieves better performance in almost all experimental configurations. This suggests that the proposed Semantic Flow Graph (SFG) captures good code semantics, and the proposed framework contributes to changeset-based bug localization.

The Student's t-test is conducted between our technique and other baselines, and the results show that the improvements are significant with $p < 0.01$. We additionally observe that with the *Commits*−level granularity, the obtained improvement is more significant than the other two granularities (*Files*−level and *Hunks*−

Table 2: Retrieval performance of different models.

Projects	Technique	*Commits–*					*Files–*					*Hunks–*				
		MRR	**MAP**	**P@1**	**P@3**	**P@5**	**MRR**	**MAP**	**P@1**	**P@3**	**P@5**	**MRR**	**MAP**	**P@1**	**P@3**	**P@5**
ZXing	*BLUiR*	0.077	0.016	0.071	0.024	0.014	0.073	0.023	0.000	0.024	0.014	0.056	0.035	0.000	0.071	0.086
	FBL-BERT	0.155	0.061	0.100	0.133	0.120	0.212	0.163	0.100	0.133	0.220	0.328	0.210	0.200	0.233	0.240
	GraphCodeBERT	0.189	0.118	0.143	0.143	0.118	0.280	0.155	0.214	0.143	0.214	0.346	0.118	0.225	0.111	0.067
	UniXcoder	0.354	0.167	0.414	0.171	0.120	0.359	0.143	0.333	0.224	0.200	0.331	0.164	0.214	0.261	0.282
	Ours	**0.439**	**0.226**	**0.429**	**0.250**	**0.225**	**0.421**	**0.185**	**0.357**	**0.226**	**0.271**	**0.422**	**0.212**	**0.333**	**0.444**	**0.400**
PDE	*BLUiR*	0.009	0.001	0.000	0.000	0.000	0.018	0.003	0.000	0.008	0.005	0.024	0.005	0.000	0.008	0.010
	FBL-BERT	0.103	0.013	0.067	0.033	0.027	0.260	0.079	0.167	0.128	0.151	0.288	0.093	0.200	0.144	0.127
	GraphCodeBERT	0.180	0.042	0.142	0.087	0.058	0.264	0.094	0.167	0.129	0.148	0.284	0.074	0.206	0.124	0.129
	UniXcoder	0.178	0.029	0.095	0.063	0.072	0.267	0.090	0.167	0.135	0.129	0.289	0.102	0.212	0.144	0.129
	Ours	**0.248**	**0.045**	**0.190**	**0.103**	**0.076**	**0.274**	**0.095**	**0.214**	**0.137**	**0.160**	**0.294**	**0.134**	**0.286**	**0.182**	**0.160**
AspectJ	*BLUiR*	0.016	0.013	0.007	0.014	0.015	0.098	0.065	0.028	0.076	0.108	0.086	0.048	0.007	0.017	0.159
	FBL-BERT	0.107	0.061	0.058	0.080	0.083	0.176	0.085	0.154	0.095	0.097	0.183	0.093	0.173	0.111	0.099
	GraphCodeBERT	0.172	0.065	0.167	0.065	0.060	0.178	0.071	0.167	0.065	0.060	0.188	0.086	0.167	0.120	0.116
	UniXcoder	0.270	0.148	0.245	0.160	0.158	0.209	0.119	0.167	0.140	**0.152**	0.250	0.134	0.250	0.150	0.138
	Ours	**0.309**	**0.169**	**0.278**	**0.198**	**0.196**	**0.272**	**0.148**	**0.250**	**0.157**	0.146	**0.262**	**0.143**	**0.250**	**0.161**	**0.163**
JDT	*BLUiR*	0.019	0.001	0.015	0.005	0.003	0.027	0.003	0.000	0.010	0.012	0.033	0.005	0.000	0.005	0.009
	FBL-BERT	0.118	0.016	0.064	0.043	0.030	0.403	0.060	0.319	0.184	0.128	0.429	0.062	0.319	0.195	0.167
	GraphCodeBERT	0.125	0.022	0.061	0.035	0.030	0.423	0.058	0.308	0.179	0.118	0.385	0.041	0.231	0.179	0.118
	UniXcoder	0.182	0.018	0.182	0.061	0.038	0.434	0.062	0.379	0.166	0.131	0.364	0.045	0.288	0.182	0.123
	Ours	**0.306**	**0.026**	**0.288**	**0.096**	**0.064**	**0.489**	**0.080**	**0.462**	**0.195**	**0.167**	**0.443**	**0.088**	**0.322**	**0.206**	**0.167**
SWT	*BLUiR*	0.005	0.001	0.000	0.000	0.000	0.020	0.003	0.016	0.005	0.006	0.014	0.001	0.000	0.000	0.013
	FBL-BERT	0.067	0.015	0.023	0.027	0.026	0.555	0.131	0.535	0.233	0.173	0.526	0.131	0.488	0.217	0.164
	GraphCodeBERT	0.105	0.018	0.048	0.026	0.022	0.535	0.137	0.525	0.220	0.175	0.536	0.132	0.516	0.220	0.159
	UniXcoder	0.129	0.035	0.107	0.106	0.063	0.548	0.149	0.524	0.233	0.183	0.535	0.143	0.535	0.205	0.179
	Ours	**0.283**	**0.085**	**0.159**	**0.177**	**0.170**	**0.560**	**0.153**	**0.540**	**0.249**	**0.192**	**0.540**	**0.147**	**0.540**	**0.228**	**0.179**
Tomcat	*BLUiR*	0.007	0.002	0.000	0.002	0.002	0.014	0.003	0.000	0.010	0.007	0.014	0.005	0.000	0.012	0.013
	FBL-BERT	0.141	0.055	0.062	0.077	0.088	0.463	0.114	0.381	0.222	0.183	0.482	0.129	0.412	0.216	0.182
	GraphCodeBERT	0.253	0.062	0.188	0.104	0.084	0.287	0.067	0.271	0.104	0.080	0.395	0.118	0.363	0.216	0.211
	UniXcoder	0.328	0.057	0.338	0.120	0.084	0.364	0.065	0.353	0.125	0.085	0.396	0.097	0.378	0.139	0.118
	Ours	**0.386**	**0.073**	**0.360**	**0.135**	**0.107**	**0.487**	**0.122**	**0.406**	**0.247**	**0.232**	**0.484**	**0.132**	**0.423**	**0.225**	**0.211**

Table 3: Ablation study of pre-training tasks of SemanticCodeBERT with Semantic Flow Graph (SFG).

Dataset	Pre-training Tasks	MRR	MAP	P@1	P@3	P@5
ZXing	-w/	0.189	0.118	0.143	0.143	0.118
	-w/ N.& E.	0.372	0.102	0.333	0.111	0.067
	-w/ N.& E.& T.& R.	**0.439**	**0.226**	**0.429**	**0.250**	**0.225**
PDE	-w/	0.180	0.042	0.142	0.087	0.058
	-w/ N.& E.	0.219	0.032	0.143	0.076	0.072
	-w/ N.& E.& T.& R.	**0.248**	**0.045**	**0.190**	**0.103**	**0.076**
AspectJ	-w/	0.172	0.065	0.167	0.065	0.060
	-w/ N.& E.	0.289	0.158	0.250	0.184	0.170
	-w/ N.& E.& T.& R.	**0.309**	**0.169**	**0.278**	**0.198**	**0.196**
JDT	-w/	0.125	0.022	0.061	0.035	0.030
	-w/ N.& E.	0.139	0.021	0.095	0.044	0.048
	-w/ N.& E.& T.& R.	**0.306**	**0.026**	**0.288**	**0.096**	**0.064**
SWT	-w/	0.105	0.018	0.048	0.026	0.022
	-w/ N.& E.	0.197	0.058	0.063	0.085	0.141
	-w/ N.& E.& T.& R.	**0.283**	**0.085**	**0.159**	**0.177**	**0.170**
Tomcat	-w/	0.253	0.062	0.188	0.104	0.084
	-w/ N.& E.	0.300	0.048	0.346	0.113	0.077
	-w/ N.& E.& T.& R.	**0.386**	**0.073**	**0.360**	**0.135**	**0.107**

level). It can be attributed that the undivided bug-inducing changeset carries enriched semantic information which can be captured by SemanticCodeBERT. This again confirms the effectiveness of the SemanticCodeBERT-based bug localization technique.

6.5 Ablation Study

To evaluate the design choices in the proposed model, we conduct several ablation studies. To begin with, as shown in Table 3, we analyze the contributions of node alignment, edge prediction, type prediction, and role prediction pre-training tasks on the six projects with commits granularity. N., E., T., and R. denote the Node Alignment, Edge Prediction, Type Prediction, and Role Prediction pre-training tasks, respectively. With all of these pre-training tasks, we train SemanticCodeBert according to the proposed new code representation SFG. According to the results, after adding Type and Role Prediction pre-training tasks, the obtained performance has universally improved. This result suggests that leveraging the node attributes (type and role) is vital to learn code representation.

Furthermore, we evaluate the effectiveness of the Hierarchical Momentum Contrastive Bug Localization (HMCBL) technique on the six projects with commits granularity. As illustrated in Table 4, for -w/o HMCBL, the memory bank and hierarchical contrastive loss which leverages similarities at different levels do not exist, and only the representation obtained by the encoder is utilized to calculate similarity.

To demonstrate the generality, the technique is evaluated with different pre-training models as the encoder of the changeset, including BERT, GraphCodeBERT, and SemanticCodeBERT. It is observed that overall much better performance will be obtained with hierarchical momentum contrastive learning, which provides large-scale negative sample interactions for representation learning and

Table 4: Ablation study of Hierarchical Momentum Contrastive Bug Localization (HMCBL) technique, where GCBERT and SCBERT are short of GraphCodeBERT and SemanticCodeBERT.

Technique	Dataset	MRR	MAP	P@1	P@3	P@5
BERT -w/o *HMCBL* (*FBL-BERT*)	ZXing	0.155	0.061	0.100	**0.133**	**0.120**
	PDE	0.103	0.013	0.067	0.033	0.027
	AspectJ	0.107	0.061	0.058	0.080	0.083
	JDT	0.118	0.016	**0.064**	0.043	0.030
	SWT	0.067	**0.015**	0.023	**0.027**	**0.026**
	Tomcat	0.141	0.055	0.062	0.077	0.088
GCBERT -w/o *HMCBL*	ZXing	0.162	0.106	0.143	0.095	0.086
	PDE	0.167	0.018	0.119	0.071	0.045
	AspectJ	0.123	0.067	0.076	**0.073**	**0.084**
	JDT	0.120	0.022	0.061	0.035	**0.036**
	SWT	0.090	**0.019**	0.048	0.021	0.022
	Tomcat	0.151	0.035	0.059	0.064	0.063
SCBERT -w/o *HMCBL*	ZXing	0.222	0.112	0.143	0.190	0.150
	PDE	0.230	**0.049**	0.142	0.095	0.069
	AspectJ	0.271	0.148	0.250	0.161	0.165
	JDT	0.217	**0.051**	0.136	**0.111**	**0.091**
	SWT	0.250	0.062	0.095	0.167	**0.185**
	Tomcat	0.285	0.053	0.265	0.092	0.069
BERT -w/ *HMCBL*	ZXing	**0.179**	**0.040**	**0.143**	0.095	0.061
	PDE	**0.156**	**0.032**	**0.119**	**0.063**	**0.051**
	AspectJ	**0.162**	**0.097**	**0.118**	**0.141**	**0.149**
	JDT	**0.128**	**0.017**	0.030	**0.070**	**0.100**
	SWT	**0.082**	0.013	**0.048**	0.024	0.021
	Tomcat	**0.235**	**0.055**	**0.169**	**0.098**	**0.096**
GCBERT -w/ *HMCBL*	ZXing	**0.189**	**0.118**	**0.143**	**0.143**	**0.118**
	PDE	**0.180**	**0.042**	**0.142**	**0.087**	**0.058**
	AspectJ	**0.172**	**0.065**	**0.167**	0.065	0.060
	JDT	**0.125**	**0.022**	**0.061**	**0.035**	0.030
	SWT	**0.105**	0.018	**0.048**	**0.026**	**0.022**
	Tomcat	**0.253**	**0.062**	**0.188**	**0.104**	**0.084**
SCBERT -w/ *HMCBL*	ZXing	**0.439**	**0.226**	**0.429**	**0.250**	**0.225**
	PDE	**0.248**	0.045	**0.190**	**0.103**	**0.076**
	AspectJ	**0.309**	**0.169**	**0.278**	**0.198**	**0.196**
	JDT	**0.306**	0.026	**0.288**	0.096	0.064
	SWT	**0.283**	**0.085**	**0.159**	**0.177**	0.170
	Tomcat	**0.386**	**0.073**	**0.360**	**0.135**	**0.107**

increases retrieval accuracy. For instance, compared with BERT -w/o HMCBL, which is the FBL-BERT exactly, BERT -w/ HMCBL improves the performance in terms of MRR scores for more than 80% projects by 15.48% to 66.67%. It is indicative of the observation that the hierarchical momentum contrastive bug localization technique can be extended as a general and effective framework with different advanced pre-training models.

6.6 Threats to Validity

Our results should be interpreted with several threats to validity in mind. As bug-inducing changes are identified using the SZZ algorithm [70], one threat to the internal validity of the results is possible noise introduced by SZZ may make the mapping between bug reports and bug-inducing changesets not very precise. However, the dataset used in the study has been validated manually [79], so this threat is minimized. Another threat to internal validity is the dataset may contain tangled changes [77]. While we do believe tangled changes can affect our results, the dataset has been widely used for changeset-based bug localization studies [17, 79], and removing tangled changes completely is extraordinarily difficult.

With regard to threats to external validity, one potential issue is that the evaluation is conducted on a limited number of bugs from several open-source projects. However, these projects feature various purposes and development styles. Also, the dataset can be considered as the de-facto evaluation target for changeset-based bug localization studies and prior studies have widely used it [17, 79].

7 CONCLUSION

We aim to advance the state-of-the-art BERT-based bug localization techniques in this paper, which currently suffer from two issues: the pre-trained BERT models on source code are not robust enough to capture code semantics and the overall bug localization models neglect the necessity of large-scale negative samples in contrastive learning and ignore the lexical similarity between bug reports and changesets. To address these two issues, we 1) propose a novel directed, multiple-label Semantic Flow Graph (SFG), which compactly and adequately captures code semantics, 2) design and train SemanticCodeBERT on the basis of SFG, and 3) design a novel Hierarchical Momentum Contrastive Bug Localization technique (HMCBL). Evaluation results confirm that our method achieves state-of-the-art performance.

8 DATA AVAILABILITY

Our replication package (including code, model, etc.) is publicly available at https://github.com/duyali2000/SemanticFlowGraph.

ACKNOWLEDGMENTS

We are very grateful to the anonymous ESEC/FSE reviewers for their valuable feedback on this work. This work was partially supported by National Natural Science Foundation of China (Grant No. 62102233), Shandong Province Overseas Outstanding Youth Fund (Grant No. 2022HWYQ-043), and Qilu Young Scholar Program of Shandong University.

REFERENCES

[1] Miltiadis Allamanis, Earl T. Barr, Christian Bird, and Charles Sutton. 2014. Learning Natural Coding Conventions. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. Association for Computing Machinery. https://doi.org/10.1145/2635868.2635883

[2] Miltiadis Allamanis, Earl T. Barr, Premkumar Devanbu, and Charles Sutton. 2018. A Survey of Machine Learning for Big Code and Naturalness. *ACM Comput. Surv.* (2018). https://doi.org/10.1145/3212695

[3] Miltiadis Allamanis, Marc Brockschmidt, and Mahmoud Khademi. 2018. Learning to Represent Programs with Graphs. In *International Conference on Learning Representations*.

[4] Miltiadis Allamanis and Charles Sutton. 2014. Mining Idioms from Source Code. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. Association for Computing Machinery. https://doi.org/10.1145/2635868.2635901

[5] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. 2018. A General Path-Based Representation for Predicting Program Properties. *SIGPLAN Not.* (2018). https://doi.org/10.1145/3192366.3192412

[6] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. 2019. Code2vec: Learning Distributed Representations of Code. *Proc. ACM Program. Lang.* POPL (2019). https://doi.org/10.1145/3290353

[7] Tal Ben-Nun, Alice Shoshana Jakobovits, and Torsten Hoefler. 2018. Neural Code Comprehension: A Learnable Representation of Code Semantics. In *Proceedings of the 32nd International Conference on Neural Information Processing Systems*. Curran Associates Inc.

[8] Benjamin Bichsel, Veselin Raychev, Petar Tsankov, and Martin Vechev. 2016. Statistical Deobfuscation of Android Applications. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. Association for Computing Machinery. https://doi.org/10.1145/2976749.2978422

[9] Nghi D. Q. Bui, Yijun Yu, and Lingxiao Jiang. 2021. Self-Supervised Contrastive Learning for Code Retrieval and Summarization via Semantic-Preserving Transformations. In *The 44th International ACM SIGIR Conference on Research and Development in Information Retrieval, Virtual Event*. ACM.

[10] Luca Buratti, Saurabh Pujar, Mihaela Bornea, Scott McCarley, Yunhui Zheng, Gaetano Rossiello, Alessandro Morari, Jim Laredo, Veronika Thost, Yufan Zhuang, et al. 2020. Exploring software naturalness through neural language models. *arXiv preprint arXiv:2006.12641* (2020).

[11] Xianshuai Cao, Yuliang Shi, Jihu Wang, Han Yu, Xinjun Wang, and Zhongmin Yan. 2022. Cross-modal Knowledge Graph Contrastive Learning for Machine Learning Method Recommendation. In *MM '22: The 30th ACM International Conference on Multimedia*. ACM.

[12] Kwonsoo Chae, Hakjoo Oh, Kihong Heo, and Hongseok Yang. 2017. Automatically Generating Features for Learning Program Analysis Heuristics for C-like Languages. *Proc. ACM Program. Lang.* (2017). https://doi.org/10.1145/3133925

[13] Qibin Chen, Jeremy Lacomis, Edward J. Schwartz, Graham Neubig, Bogdan Vasilescu, and Claire Le Goues. 2022. VarCLR: Variable Semantic Representation Pre-training via Contrastive Learning. In *44th IEEE/ACM 44th International Conference on Software Engineering*. ACM.

[14] Ting Chen, Simon Kornblith, Mohammad Norouzi, and Geoffrey Hinton. 2020. A simple framework for contrastive learning of visual representations. In *International conference on machine learning*. PMLR.

[15] Xinlei Chen, Haoqi Fan, Ross B. Girshick, and Kaiming He. 2020. Improved Baselines with Momentum Contrastive Learning. *CoRR* (2020).

[16] Hyunsoo Cho, Jinseok Seol, and Sang-goo Lee. 2021. Masked Contrastive Learning for Anomaly Detection. In *Proceedings of the Thirtieth International Joint Conference on Artificial Intelligence, Virtual Event / Montreal*. ijcai.org.

[17] Agnieszka Ciborowska and Kostadin Damevski. 2022. Fast changeset-based bug localization with BERT. In *2022 IEEE/ACM 44th International Conference on Software Engineering (ICSE)*. https://doi.org/10.1145/3510003.3510042

[18] Agnieszka Ciborowska, Michael J Decker, and Kostadin Damevski. 2022. Online Adaptable Bug Localization for Rapidly Evolving Software. *arXiv preprint arXiv:2203.03544* (2022).

[19] Chris Cummins, Pavlos Petoumenos, Zheng Wang, and Hugh Leather. 2017. End-to-End Deep Learning of Optimization Heuristics. In *2017 26th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. https://doi.org/10.1109/PACT.2017.24

[20] Yaniv David, Uri Alon, and Eran Yahav. 2020. Neural reverse engineering of stripped binaries using augmented control flow graphs. *Proceedings of the ACM on Programming Languages* (2020). https://doi.org/10.1145/3428293

[21] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. https://doi.org/10.18653/V1/N19-1423

[22] Yali Du, Yinwei Wei, Wei Ji, Fan Liu, Xin Luo, and Liqiang Nie. 2023. Multi-queue Momentum Contrast for Microvideo-Product Retrieval. In *Proceedings of the Sixteenth ACM International Conference on Web Search and Data Mining, WSDM 2023, Singapore, 27 February 2023 - 3 March 2023*. ACM. https://doi.org/10.1145/3539597.3570405

[23] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. 2020. Codebert: A pre-trained model for programming and natural languages. (2020). https://doi.org/10.18653/V1/2020.FINDINGS-EMNLP.139

[24] Daya Guo, Shuai Lu, Nan Duan, Yanlin Wang, Ming Zhou, and Jian Yin. [n. d.]. UniXcoder: Unified Cross-Modal Pre-training for Code Representation. ([n. d.]).

[25] Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, Michele Tufano, Shao Kun Deng, Colin B. Clement, Dawn Drain, Neel Sundaresan, Jian Yin, Daxin Jiang, and Ming Zhou. 2021. Graphcodebert: Pre-training code representations with data flow. (2021).

[26] Jin Guo, Jinghui Cheng, and Jane Cleland-Huang. 2017. Semantically Enhanced Software Traceability Using Deep Learning Techniques. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. https://doi.org/10.1109/ICSE.2017.9

[27] Rahul Gupta, Soham Pal, Aditya Kanade, and Shirish Shevade. 2017. DeepFix: Fixing Common C Language Errors by Deep Learning. In *Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence*. AAAI Press. https://doi.org/10.1609/AAAI.V31I1.10742

[28] Kaiming He, Haoqi Fan, Yuxin Wu, Saining Xie, and Ross Girshick. 2020. Momentum contrast for unsupervised visual representation learning. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*. https://doi.org/10.1109/CVPR42600.2020.00975

[29] Abram Hindle, Earl T. Barr, Zhendong Su, Mark Gabel, and Premkumar Devanbu. 2012. On the Naturalness of Software. In *Proceedings of the 34th International Conference on Software Engineering*. https://doi.org/10.1145/2902362

[30] Xing Hu, Yuhan Wei, Ge Li, and Zhi Jin. 2017. CodeSum: Translate Program Language to Natural Language.

[31] Xuan Huo, Ming Li, and Zhi-Hua Zhou. 2020. Control Flow Graph Embedding Based on Multi-Instance Decomposition for Bug Localization. In *The Thirty-Fourth AAAI Conference on Artificial Intelligence, AAAI 2020, The Thirty-Second Innovative Applications of Artificial Intelligence Conference, IAAI 2020, The Tenth AAAI Symposium on Educational Advances in Artificial Intelligence, EAAI 2020, New York, NY, USA, February 7-12, 2020*. AAAI Press.

[32] Xuan Huo, Ferdian Thung, Ming Li, David Lo, and Shu-Ting Shi. 2019. Deep transfer bug localization. *IEEE Transactions on software engineering* (2019). https://doi.org/10.1109/TSE.2019.2920771

[33] Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. 2019. Codesearchnet challenge: Evaluating the state of semantic code search. *arXiv preprint arXiv:1909.09436* (2019).

[34] Sergey Ioffe and Christian Szegedy. 2015. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *International conference on machine learning*.

[35] Xue Jiang, Zhuoran Zheng, Chen Lyu, Liang Li, and Lei Lyu. 2021. Treebert: A tree-based pre-trained model for programming language. In *Uncertainty in Artificial Intelligence*. PMLR.

[36] Jeff Johnson, Matthijs Douze, and Hervé Jégou. 2019. Billion-scale similarity search with gpus. *IEEE Transactions on Big Data* (2019). https://doi.org/10.1109/TBDATA.2019.2921572

[37] James A Jones and Mary Jean Harrold. 2005. Empirical evaluation of the tarantula automatic fault-localization technique. In *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*. https://doi.org/10.1145/1101908.1101949

[38] Aditya Kanade, Petros Maniatis, Gogul Balakrishnan, and Kensen Shi. 2019. Pre-trained contextual embedding of source code. (2019).

[39] Minguk Kang and Jaesik Park. 2020. ContraGAN: Contrastive Learning for Conditional Image Generation. In *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020*.

[40] Omar Khattab and Matei Zaharia. 2020. ColBERT: Efficient and Effective Passage Search via Contextualized Late Interaction over BERT. In *Proceedings of the 43rd International ACM SIGIR conference on research and development in Information Retrieval*. ACM. https://doi.org/10.1145/3397271.3401075

[41] Dongsun Kim, Yida Tao, Sunghun Kim, and Andreas Zeller. 2013. Where should we fix this bug? a two-phase recommendation model. *IEEE transactions on software Engineering* (2013). https://doi.org/10.1109/TSE.2013.24

[42] Kisub Kim, Sankalp Ghatpande, Kui Liu, Anil Koyuncu, Dongsun Kim, Tegawendé F Bissyandé, Jacques Klein, and Yves Le Traon. 2022. DigBug—Pre/post-processing operator selection for accurate bug localization. *Journal of Systems and Software* (2022).

[43] Ted Kremenek, Andrew Y. Ng, and Dawson Engler. 2007. A Factor Graph Model for Software Bug Finding. In *Proceedings of the 20th International Joint Conference on Artifical Intelligence*. Morgan Kaufmann Publishers Inc.

[44] An Ngoc Lam, Anh Tuan Nguyen, Hoan Anh Nguyen, and Tien N Nguyen. 2017. Bug localization with combination of deep learning and information retrieval. In *2017 IEEE/ACM 25th International Conference on Program Comprehension (ICPC)*. https://doi.org/10.1109/ICPC.2017.24

[45] Hao Lan, Li Chen, and Baochun Li. 2021. Accelerated Device Placement Optimization with Contrastive Learning. In *ICPP 2021: 50th International Conference on Parallel Processing*. ACM.

[46] Zhengliang Li, Zhiwei Jiang, Xiang Chen, Kaibo Cao, and Qing Gu. 2021. Laprob: a label propagation-based software bug localization method. *Information and Software Technology* (2021). https://doi.org/10.1016/J.INFSOF.2020.106410

[47] Jinfeng Lin, Yalin Liu, Qingkai Zeng, Meng Jiang, and Jane Cleland-Huang. 2021. Traceability Transformed: Generating More Accurate Links with Pre-Trained BERT Models. In *Proceedings of the 43rd International Conference on Software Engineering*. https://doi.org/10.1109/ICSE43902.2021.00040

[48] Chao Liu, Xifeng Yan, Long Fei, Jiawei Han, and Samuel P Midkiff. 2005. SOBER: statistical model-based bug localization. *ACM SIGSOFT Software Engineering Notes* (2005). https://doi.org/10.1145/1081706.1081753

[49] Ilya Loshchilov and Frank Hutter. 2019. Decoupled Weight Decay Regularization. In *7th International Conference on Learning Representations*. OpenReview.net.

[50] Yiling Lou, Qihao Zhu, Jinhao Dong, Xia Li, Zeyu Sun, Dan Hao, Lu Zhang, and Lingming Zhang. 2021. Boosting coverage-based fault localization via graph-based representation learning. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*.

[51] Yi-Fan Ma, Yali Du, and Ming Li. 2023. Capturing the Long-Distance Dependency in the Control Flow Graph via Structural-Guided Attention for Bug Localization. In *Proceedings of the Thirty-Second International Joint Conference on Artificial Intelligence, IJCAI 2023, 19th-25th August 2023, Macao, SAR, China*. ijcai.org. https://doi.org/10.24963/IJCAI.2023/249

[52] Yi-Fan Ma and Ming Li. 2022. The flowing nature matters: feature learning from the control flow graph of source code for bug localization. *Mach. Learn.* (2022). https://doi.org/10.1007/S10994-021-06078-4

[53] Yi-Fan Ma and Ming Li. 2022. Learning from the Multi-Level Abstraction of the Control Flow Graph via Alternating Propagation for Bug Localization. In *IEEE International Conference on Data Mining*. IEEE.

[54] Andrew L Maas, Awni Y Hannun, Andrew Y Ng, et al. 2013. Rectifier nonlinearities improve neural network acoustic models. In *Proc. icml*.

[55] Ginika Mahajan and Neha Chaudhary. 2022. Design and development of novel hybrid optimization-based convolutional neural network for software bug localization. *Soft Computing* (2022). https://doi.org/10.1007/S00500-022-07341-Z

[56] Lili Mou, Ge Li, Lu Zhang, Tao Wang, and Zhi Jin. 2016. Convolutional Neural Networks over Tree Structures for Programming Language Processing. In *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence*. AAAI Press. https://doi.org/10.1609/AAAI.V30I1.10139

[57] Vijayaraghavan Murali, Lee Gross, Rebecca Qian, and Satish Chandra. 2021. Industry-scale IR-based bug localization: a perspective from Facebook. In *2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice*. https://doi.org/10.1109/ICSE-SEIP52600.2021.00028

[58] Chao Ni, Wei Wang, Kaiwen Yang, Xin Xia, Kui Liu, and David Lo. 2022. The best of both worlds: integrating semantic features with expert features for defect prediction and localization. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*.

[59] Aaron van den Oord, Yazhe Li, and Oriol Vinyals. 2018. Representation learning with contrastive predictive coding. *arXiv preprint arXiv:1807.03748* (2018).

[60] Mike Papadakis and Yves Le Traon. 2015. Metallaxis-FL: mutation-based fault localization. *Software Testing, Verification and Reliability* (2015). https://doi.org/10.1002/STVR.1509

[61] Renaud Pawlak, Martin Monperrus, Nicolas Petitprez, Carlos Noguera, and Lionel Seinturier. 2015. Spoon: A Library for Implementing Analyses and Transformations of Java Source Code. *Software: Practice and Experience* (2015).

[62] Michael Pradel, Vijayaraghavan Murali, Rebecca Qian, Mateusz Machalica, Erik Meijer, and Satish Chandra. 2020. Scaffle: bug localization on millions of files. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*.

[63] Michael Pradel and Koushik Sen. 2018. DeepBugs: A Learning Approach to Name-Based Bug Detection. *Proc. ACM Program. Lang.* (2018). https://doi.org/10.1145/3276517

[64] Binhang Qi, Hailong Sun, Wei Yuan, Hongyu Zhang, and Xiangxin Meng. 2022. DreamLoc: A Deep Relevance Matching-Based Framework for bug Localization. *IEEE Transactions on Reliability* (2022). https://doi.org/10.1109/TR.2021.3104728

[65] Shibo Qi, Rize Jin, and Joon-Young Paik. 2022. eMoCo: Sentence Representation Learning With Enhanced Momentum Contrast. In *Proceedings of the 5th International Conference on Computer Science and Software Engineering*.

[66] Yiyue Qian, Yiming Zhang, Qianlong Wen, Yanfang Ye, and Chuxu Zhang. 2022. Rep2Vec: Repository Embedding via Heterogeneous Graph Adversarial Contrastive Learning. In *KDD '22: The 28th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*. ACM.

[67] Fangcheng Qiu, Meng Yan, Xin Xia, Xinyu Wang, Yuanrui Fan, Ahmed E Hassan, and David Lo. 2020. JITO: a tool for just-in-time defect identification and localization. In *Proceedings of the 28th ACM joint meeting on european software engineering conference and symposium on the foundations of software engineering*.

[68] Mohammad Masudur Rahman and Chanchal K Roy. 2018. Improving IR-based bug localization with context-aware query reformulation. In *Proceedings of the 2018 26th ACM joint meeting on European software engineering conference and symposium on the foundations of software engineering*. https://doi.org/10.1145/3236024.3236065

[69] Stephen E. Robertson, Steve Walker, and Micheline Hancock-Beaulieu. 2000. Experimentation as a way of life: Okapi at TREC. *Inf. Process. Manag.* (2000).

[70] Giovanni Rosa, Luca Pascarella, Simone Scalabrino, Rosalia Tufano, Gabriele Bavota, Michele Lanza, and Rocco Oliveto. 2021. Evaluating SZZ Implementations Through a Developer-Informed Oracle. In *Proceedings of the 43rd International Conference on Software Engineering*.

[71] Ripon K Saha, Matthew Lease, Sarfraz Khurshid, and Dewayne E Perry. 2013. Improving bug localization using structured information retrieval. In *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. https://doi.org/10.1109/ASE.2013.6693093

[72] Shivkumar Shivaji, E James Whitehead, Ram Akella, and Sunghun Kim. 2012. Reducing features to improve code change-based bug prediction. *IEEE Transactions on Software Engineering* (2012). https://doi.org/10.1109/TSE.2012.43

[73] Marius Smytzek and Andreas Zeller. 2022. SFLKit: a workbench for statistical fault localization. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*.

[74] Chenning Tao, Qi Zhan, Xing Hu, and Xin Xia. 2022. C4: contrastive cross-language code clone detection. In *Proceedings of the 30th IEEE/ACM International Conference on Program Comprehension*. ACM.

[75] Simon Urli, Zhongxing Yu, Lionel Seinturier, and Martin Monperrus. 2018. How to design a program repair bot? insights from the repairnator project. In *2018 IEEE/ACM 40th international conference on software engineering: software engineering in practice Track (ICSE-SEIP)*. https://doi.org/10.1145/3183519.3183540

[76] Iris Vessey. 1985. Expertise in debugging computer programs: A process analysis. *International Journal of Man-Machine Studies* (1985). https://doi.org/10.1016/S0020-7373(85)80054-7

[77] Min Wang, Zeqi Lin, Yanzhen Zou, and Bing Xie. 2020. CoRA: Decomposing and Describing Tangled Code Changes for Reviewer. In *Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering*.

[78] Xuanrun Wang, Kanglin Yin, Qianyu Ouyang, Xidao Wen, Shenglin Zhang, Wenchi Zhang, Li Cao, Jiuxue Han, Xing Jin, and Dan Pei. 2022. Identifying Erroneous Software Changes through Self-Supervised Contrastive Learning on Time Series Data. In *IEEE 33rd International Symposium on Software Reliability Engineering, ISSRE 2022, Charlotte, NC, USA, October 31 - Nov. 3, 2022*. IEEE.

[79] Ming Wen, Rongxin Wu, and Shing-Chi Cheung. 2016. Locus: Locating Bugs from Software Changes. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*. Association for Computing Machinery. https://doi.org/10.1145/2970276.2970359

[80] W Eric Wong, Ruizhi Gao, Yihao Li, Rui Abreu, and Franz Wotawa. 2016. A survey on software fault localization. *IEEE Transactions on Software Engineering* (2016). https://doi.org/10.1109/TSE.2016.2521368

[81] Rongxin Wu, Ming Wen, Shing-Chi Cheung, and Hongyu Zhang. 2018. Changelocator: locate crash-inducing changes based on crash reports. *Empirical Software Engineering* (2018). https://doi.org/10.1007/S10664-017-9567-4

[82] Zhirong Wu, Yuanjun Xiong, Stella X Yu, and Dahua Lin. 2018. Unsupervised feature learning via non-parametric instance discrimination. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. https://doi.org/10.1109/CVPR.2018.00393

[83] Huan Xie, Yan Lei, Meng Yan, Yue Yu, Xin Xia, and Xiaoguang Mao. 2022. A Universal Data Augmentation Approach for Fault Localization. In *Proceedings of the 44th International Conference on Software Engineering*. Association for Computing Machinery. https://doi.org/10.1145/3510003.3510136

[84] Zhongxing Yu, Chenggang Bai, and Kai-Yuan Cai. 2013. Mutation-Oriented Test Data Augmentation for GUI Software Fault Localization. *Inf. Softw. Technol.* (2013). https://doi.org/10.1016/j.infsof.2013.07.004

[85] Zhongxing Yu, Chenggang Bai, and Kai-Yuan Cai. 2015. Does the Failing Test Execute a Single or Multiple Faults? An Approach to Classifying Failing Tests. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1*. IEEE Press.

[86] Zhongxing Yu, Hai Hu, Chenggang Bai, Kai-Yuan Cai, and W. Eric Wong. 2011. GUI Software Fault Localization Using N-gram Analysis. In *2011 IEEE 13th International Symposium on High-Assurance Systems Engineering*. https://doi.org/10.1109/HASE.2011.29

[87] Zhongxing Yu, Matias Martinez, Zimin Chen, Tegawendé F. Bissyandé, and Martin Monperrus. 2023. Learning the Relation Between Code Features and Code Transforms With Structured Prediction. *IEEE Transactions on Software Engineering* (2023). https://doi.org/10.1109/TSE.2023.3275380

[88] Zhongxing Yu, Matias Martinez, Benjamin Danglot, Thomas Durieux, and Martin Monperrus. 2019. Alleviating Patch Overfitting with Automatic Test Generation: A Study of Feasibility and Effectiveness for the Nopol Repair System. *Empirical Softw. Engg.* (2019). https://doi.org/10.1007/s10664-018-9619-4

[89] Chenxi Zhang, Xin Peng, Tong Zhou, Chaofeng Sha, Zhenghui Yan, Yiru Chen, and Hong Yang. [n. d.]. TraceCRL: contrastive representation learning for microservice trace analysis. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, publisher = ACM, year = 2022,*.

[90] Jinglei Zhang, Rui Xie, Wei Ye, Yuhan Zhang, and Shikun Zhang. 2020. Exploiting code knowledge graph for bug localization via bi-directional attention. In *Proceedings of the 28th International Conference on Program Comprehension*.

[91] Jian Zhou, Hongyu Zhang, and David Lo. 2012. Where Should the Bugs Be Fixed? - More Accurate Information Retrieval-Based Bug Localization Based on Bug Reports. In *Proceedings of the 34th International Conference on Software Engineering*. IEEE Press.

[92] Yaqin Zhou, Shangqing Liu, Jingkai Siow, Xiaoning Du, and Yang Liu. 2019. *Devign: Effective Vulnerability Identification by Learning Comprehensive Program Semantics via Graph Neural Networks*. Curran Associates Inc.

[93] Ziye Zhu, Yun Li, Hanghang Tong, and Yu Wang. 2020. Cooba: Cross-project bug localization via adversarial transfer learning. In *IJCAI*. https://doi.org/10.24963/IJCAI.2020/493

[94] Ziye Zhu, Yun Li, Yu Wang, Yaojing Wang, and Hanghang Tong. 2021. A deep multimodal model for bug localization. *Data Mining and Knowledge Discovery* (2021).

[95] Ziye Zhu, Hanghang Tong, Yu Wang, and Yun Li. 2022. Enhancing bug localization with bug report decomposition and code hierarchical network. *Knowledge-Based Systems* (2022). https://doi.org/10.1016/J.KNOSYS.2022.108741

[96] Weiqin Zou, David Lo, Zhenyu Chen, Xin Xia, Yang Feng, and Baowen Xu. 2020. How Practitioners Perceive Automated Bug Report Management Techniques. *IEEE Transactions on Software Engineering* (2020). https://doi.org/10.1109/TSE.2018.2870414

Received 2023-02-02; accepted 2023-07-27