Received: 28 June 2023 | Revised: 12 November 2023 | Accepted: 5 March 2024
DOI: 10.1002/smr.2669

SPECIAL ISSUE - TECHNOLOGY PAPER

Journal of Software: Evolution and Process
WILEY

WELL: Applying bug detectors to bug localization via weakly supervised learning

Huangzhao Zhang | **Zhuo Li** | **Jia Li** | **Zhi Jin** | **Ge Li**

Key Laboratory of High Confidence Software Technologies, Peking University, Beijing, China

Correspondence
Ge Li, Key Laboratory of High Confidence Software Technologies, Peking University, No. 1542, No. 1 Science Building. No. 5 Yiheyuan Road, Haidian District, Beijing 100871, China.
Email: lige@pku.edu.cn

Abstract

Bug localization, which is used to help programmers identify the location of bugs in source code, is an essential task in software development. Researchers have already made efforts to harness the powerful deep learning (DL) techniques to automate it. However, training bug localization model is usually challenging because it requires a large quantity of data labeled with the bug's exact location, which is difficult and time-consuming to collect. By contrast, obtaining bug detection data with binary labels of whether there is a bug in the source code is much simpler. This paper proposes a WEakly supervised bug LocaLization (WELL) method, which only uses the bug detection data with binary labels to train a bug localization model. With CodeBERT finetuned on the buggy-or-not binary labeled data, WELL can address bug localization in a weakly supervised manner. The evaluations on three method-level synthetic datasets and one file-level real-world dataset show that WELL is significantly better than the existing state-of-the-art model in typical bug localization tasks such as variable misuse and other bugs.

KEYWORDS
bug detection, bug localization, weakly supervised learning

1 | INTRODUCTION

Bug localization is one of the key activities in software engineering (SE), where the practitioners are supposed to position the erroneous part of the code. Effectively automating bug localization is essential to the software developers as it can improve productivity and software quality greatly.

In the past decade, deep learning (DL) has demonstrated its great powerfulness in many SE tasks, and has achieved state-of-the-art performance in functionality classification,[1,2] code clone detection,[3,4] method naming,[5,6] code completion,[7–9] and code summarization.[10–12]

These may show the feasibility of harnessing the DL techniques to facilitate automated bug localization. Researchers have already tried to apply DL models to bug localization[13–16]. GREAT[15] and CuBERT[16] are among the state-of-the-art DL models for bug localization and further fixing. Taking variable misuse (VarMisuse),[13] for instance, which is one of the most thoroughly studied DL-based bug localization tasks, the DL models are supposed to locate the erroneously used variable in the given buggy code. Existing approaches, including GREAT and CuBERT, are trained in the end-to-end style, that is, the buggy locations in the code are fine-grained annotated in the training set.

One major challenge in existing DL solutions for bug localization is that obtaining models such as GREAT and CuBERT requires a large quantity of buggy-location-annotated training data. This kind of data provides *strong supervision* as the annotations are very fine-grained and highly related to the bug localization task. However, such dataset with reasonable annotation quality and sufficient examples is difficult to collect or annotate, due to the huge expense of manpower and resources in real-world scenarios.

 © 2024 John Wiley & Sons Ltd.

According to Benton et al,[17] there exists only few large and publicly available bug datasets with high quality for research purpose. The bug localization datasets are usually obtained in two major ways: manual annotation and automatic collection. ① As mentioned before, it consumes a lot of manpower and resources to annotate high-quality bug localization data pairs. The annotators must be experienced software developers, and they have to take enough time to read and comprehend the code along with the bug report to locate the buggy position for every example to be annotated.

② Automatic dataset collection, on the other hand, is much more efficient. It often utilizes web crawlers and rule-based filters to find bug fixing commits in the open source projects to locate the bugs. However, the annotation quality of automatic approaches is not guaranteed. For example, Lutellier et al[18] recently propose a large (million-level) program repair dataset collected from commit history of open source projects, but up to 7 of the 100 random samples are not actually bug-related commits in their manual investigations.

On the other hand, bug detection is usually a binary classification task, and utilizes the coarse-grained annotated data to train DL models. The buggy-or-not annotated data provides *weak supervision* compared with bug localization, as the annotation granularity is much coarser. Data for bug detection is much easier to collect or annotate. One may automatically run tests over the projects to determine which function or file is buggy, without much effort to dive into the project nor the bug report. Hence, in short words, bug localization data is scarce and difficult to collect or annotate, while data for bug detection is more easily accessible.

Thence, we introduce the idea of *weakly supervised learning* into bug localization. The methodology is to train the strong model (bug localization) with weak supervision signals (bug detection dataset), as illustrated in Figure 1. Heuristically, a bug detection model internally learns the dependency of buggy labels on the buggy portions in the code. Retrieving such knowledge embedded in the bug detection model to achieve bug localization is very desirable and feasible. Based on such intuition, we propose ***WE***akly supervised bug ***L***oca***L***ization (WELL), which transforms the bug detection model into a bug locator without any additional trainable weights nor bug localization data.

WELL makes full usage of the easily accessible bug detection data to tackle the lack-of-data challenge in bug localization.

To summarize the technical part, WELL harnesses the powerfulness of the pre-trained CodeBERT model,[19] and finetunes CodeBERT for bug localization in the weakly supervised manner. Concretely, WELL finetunes CodeBERT on bug detection datasets, during the training stage.

When locating bugs, WELL acquires attention score from the finetuned CodeBERT and extracts the critical part from the input source code based on the score. By intuition, if CodeBERT classifies a piece of code as buggy, the buggy fragment is likely to be included in the key portion of the input, which draws the model's most attention. In this way, the weakly supervised bug localization is achieved in WELL.

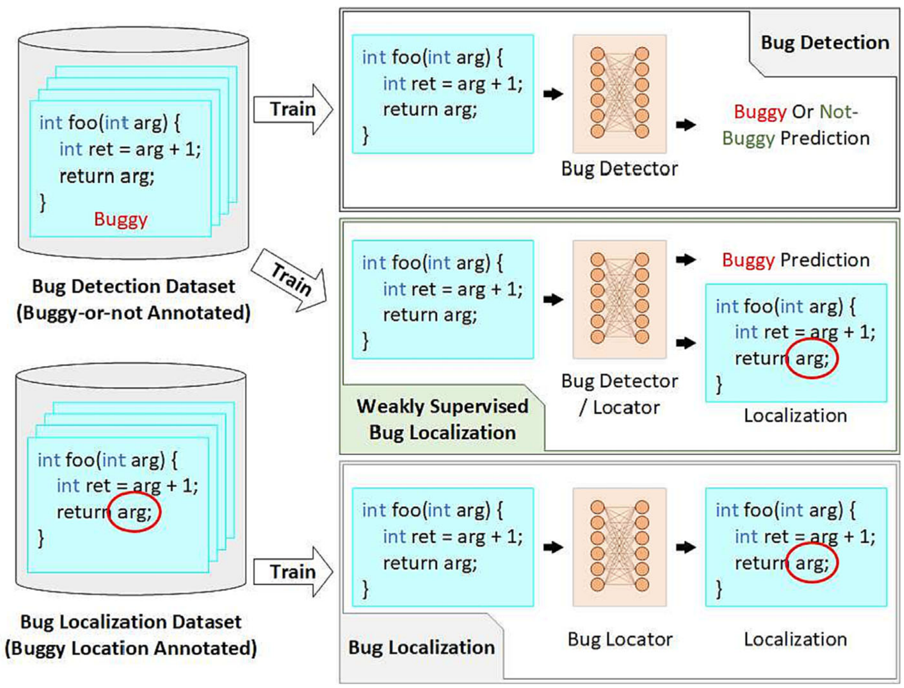


FIGURE 1 An demonstrative example of bug detection, bug localization, and weakly supervised bug localization. The bug detection dataset (upper left) consists of source code pieces and the corresponding binary buggy-or-not labels, which is easily accessible. While the bug localization dataset (lower left) is annotated by buggy locations in the code, which is hard to collect. Models for weakly supervised bug localization (mid right) are trained upon the detection dataset but are able to carry out both bug detection and localization.

20477481, 2024, 9, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/smr.2669 by Peking University Health, Wiley Online Library on [05/01/2025]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

At last, to demonstrate the effectiveness and capacity of WELL, we carry out in-depth evaluations on three different synthetic token-level bug localization datasets and a real-world bug localization dataset of student programs. The three synthetic datasets include VarMisuse, bi-operator misuse (BiOpMisuse) and boundary condition error (BoundError). They are the most studied synthetic bug localization datasets using DL approaches. The student program bug localization dataset is called StuBug. On this dataset, we train our detection model with binary test result labels ("passed" or "wrong answer") and localize the buggy lines without any other annotations. For the three synthetic datasets, on average, WELL correctly detects and accurately locates 79.74% of bugs, and the extended version (WELL-1) even locates 87.57% of bugs. Specifically, WELL improves the localization accuracy of VarMisuse to 92.28% by over 4% compared with the state-of-the-art CuBERT baseline. For StuBug, WELL can locate at least 1 bug for over 29%/85% programs when reporting the top-1/top-10 suspicious line(s) per program, outperforming the baselines significantly. Ablation study further demonstrates that it is feasible to apply weak supervision to other backbones, such as LSTM. The code of this project is open sourced on Github [*].

The contributions of this paper are summarized as follows:

- We introduce the methodology of weak supervision into bug localization. This methodology tackles the lack-of-data problem and makes full usage of the easily accessible bug detection data.
- We propose WELL, which turns bug detectors into bug locators without training data of localization nor additional trainable parameters. WELL learns to locate bugs with only bug detection datasets.
- We carry out in-depth evaluations to demonstrate the effectiveness of our proposed WELL against existing state-of-the-art DL methods for bug localization. Compared with the baseline models jointly obtained with strong supervision signals, WELL trained with weak supervision produces competitive or even better performance.
- We demonstrate the capability of applying weak supervision to LSTM backbone through ablation study, suggesting the capacity and portability of the methodology.

2 | RELATED WORK

In this section, we discuss the most relevant work to this paper, including the subject tasks of DL for bug detection and bug localization (Section 2.1), the methodology of weakly supervised learning (Section 2.2) and the techniques for DL model visualization and explanation (Section 2.3).

2.1 | DL for bug detection and localization

By far, quite a lot of efforts have been made in source code processing by adopting DL techniques in the SE community.[2,4,6,8,9,12] To leverage DL for bug detection, Wang et al[20] propose AST-based deep belief network for defect prediction. Choi et al[21] utilize memory neural network to predict buffer overrun. Li et al[22] propose VulDeePecker to detect several types of vulnerabilities in source code. Pradel and Sen[23] propose DeepBugs for bugs in function call statements and binary expressions, with a feed-forward network taking variable types and names as inputs.

As for DL-based bug localization, studies are conducted mostly on artificial synthetic datasets, where certain types of bugs are injected into the clean code to formulate buggy-location-annotated data pairs, due to the aforementioned lack-of-data problem. Allamanis et al[13] first propose the VarMisuse task, which is one of the most thoroughly studied tasks in DL-based bug localization at present. Vasic et al[14] employ the sequence-to-pointer (Seq2Ptr) architecture to detect, locate and fix VarMisuse bugs jointly.

More recently, Hellendoorn et al[15] propose two architectures to generate distributed representations for source code in order to locate bugs, namely, Graph-Sandwich and GREAT.

Kanade et al[16] propose the pre-trained CuBERT for VarMisuse and multiple other bug detection and localization tasks.

Although some of the aforementioned existing work train the model for bug detection and localization jointly,[14–16] that is, the DL models are trained to detect and locate bugs simultaneously upon bug localization datasets, they do not seek to facilitate bug localization via the weak supervision signals from the bug detection (binary classification) data. As a result, the lack-of-data problem is often challenging and inevitable in traditional DL for bug localization in the real world. In this paper, on the contrary, WELL adopts the methodology of weakly supervised learning, and leverages the easily accessible and abundant buggy-or-not data to finetune CodeBERT as a bug detector for token-level fine-grained bug localization.

[*]https://github.com/Lizhmq/CodeBertRationalizer.

20477481, 2024, 9, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/smr.2669 by Peking University Health, Wiley Online Library on [05/01/2025]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

2.2 | Weakly supervised learning

Weak supervision can be categorized as incomplete supervision, inexact supervision and inaccurate supervision. In this paper, we focus on inexact supervision, where the annotation of the training data is only coarse-grained labels (e.g., buggy-or-not data provides weak supervision to the bug localization task). Please refer to the survey[24] for more detailed discussions of other types of weak supervision.

The methodology of weakly supervised learning has been demonstrated to be valid in many DL tasks. In computer vision (CV), researchers facilitate image semantic segmentation via coarse-grained annotated classification datasets. The weak annotations include bounding boxes,[25,26] scribbles,[27] and points.[28] More recently, pixel-level segmentation by image-level annotation has been achieved through the technique of CAM.[29–32] As for natural language processing (NLP), researchers also leverage weak supervision in sequence labeling tasks, such as named entity recognition (NER), to ease the burden of data annotation.[33–36] The most relevant approach to WELL is Token Tagger,[34] which employs attention-based architecture for weakly supervised NER with sentence-level annotations. Token Tagger is trained to classify whether a named entity is in the sentence, and during NER, it selects the most important tokens based on the attention score.

In this paper, we adopt the idea of weakly supervised learning to train bug locators with buggy-or-not annotated data. The inner logic to leverage attention for weakly supervised learning of WELL is inspired by Token Tagger.

2.3 | DL model visualization and explanation

Interpreting the DL models could aid the researchers to understand and explain the inner mechanism of neural networks, and such techniques may technically promote weakly supervised learning. In CV, at present, the CAM family[29–32] are the most widely-applied and mature techniques to visualize and explain the image classifiers. It generates a heat map, where the value reflects the contribution and the importance of the corresponding pixel to the final prediction.

As for NLP, which is more relevant to our subject tasks, selective rationalization[37–41] is one of the most effective techniques to explain sentence classifiers. It identifies the rationale, which is a subsequence of the input sentence, to best explain or support the prediction from the DL model. The recent proposed FRESH[41] utilizes a two-model framework for rationalization. It generates heuristic feature scores (e.g., attention score) from the subject BERT to derive pseudo binary tags on words, and finetunes another BERT in a sequence labeling manner as the rationale tagger. This work demonstrates the powerfulness of pre-trained models in comprehending and explaining the NLP models.

Inspired by FRESH, we employ a simplified framework in WELL, which utilizes one single finetuned CodeBERT[19] bug detector to generate attention scores for bug localization. As demonstrated in selective rationalization, attention in CodeBERT is supposed to mine the portions of the code information for bug detection, and such portions are supposed to be bugs.

3 | PROBLEM DEFINITION & PRELIMINARY

In this section, we first provide our formal definition of the object tasks (Section 3.1). Then we explain the multihead attention mechanism in the transformer model (Section 3.2), based on which we facilitate WELL. And at last, we introduce the large CodeBERT model pre-trained for source code (Section 3.3), which acts as the backbone of WELL. The symbols we employ in this paper are summarized in Table 1.

TABLE 1 Summary of notations and symbols in this paper.

Notation	Definition
$\mathcal{D}^t, \mathcal{D}^v, \mathcal{D}^e$	Dataset for training, validation & evaluation.
$\mathcal{X}, \mathcal{Y}$	Source code space & annotation space.
$x = (t_1, ..., t_l)$	Token sequence of the source code.
$(s_1, ..., s_{l'})$	Subtoken sequence of the source code.
C, Θ_C	DL model & its trainable parameters.
$y, \tilde{y}$	Annotation & prediction from the model.
$\mathcal{L}(\tilde{y}, y)$	Loss function.
Q, K, V	Query, key & value matrices in attention.
$h = (h_0, ..., h_{l'})$	Context-aware hidden states.
$\alpha = (\alpha_1, ..., \alpha_{n_h})$	Multihead attention with n_h heads.

20477481, 2024, 9, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/smr.2669 by Peking University Health, Wiley Online Library on [05/01/2025]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

3.1 | Bug detection and localization

3.1.1 | Dataset

Both bug detection and localization are supervised tasks, because the datasets are annotated. A typical dataset consists of multiple pairs of examples, that is, $\mathcal{D}=\{(x_1,y_1),...,(x_n,y_n)\}$, where n refers to the size of the dataset. A pair of example $(x,y)\in\mathcal{D}$ includes a source code piece $x\in\mathcal{X}$ and its corresponding annotation $y\in\mathcal{Y}$. The annotation space varies for different tasks, which will be defined later in this section. The source code is already tokenized, that is, $x=(t_1,...,t_l)$, where t_i is the ith token in x and l is the length of the token sequence.

3.1.2 | Model

A DL model C takes the tokenized source code x as input, and outputs $\tilde{y}\in\mathcal{Y}$ as the prediction. Note that the output format also varies for different tasks. C is supposed to produce $\tilde{y}=y$, where the prediction exactly matches the ground-truth. To obtain the optimal parameters in C is to optimize the objective function $\min_{\Theta_C}\sum_{(x,y)\in\mathcal{D}^t}\mathcal{L}(\tilde{y},y)$, where $\mathcal{L}(\cdot)$ is the loss function (usually cross entropy), which measures the similarity of $\tilde{y}$ and y.

For those C's that require different input formats (e.g., AST or graph), we assume that they process the format internally.

3.1.3 | Bug detection

We define bug detection in method-level. Bug detection is to determine whether a bug exists in the given method or function. Therefore, the annotation space for bug detection is binary, that is, $\mathcal{Y}=\{0,1\}$, where $y=1$ suggests that the corresponding function x is buggy, while $y=0$ the opposite. One often adopted approach to build a bug detector is to first generate an encoding vector of the source code via a code representation model, and then classify the encoding with a feed forward network.

3.1.4 | Bug localization

Bug localization is to determine which token subsequences cause a bug in the given code. The annotation for bug localization is defined as a sequence of binary labels, that is, $\mathcal{Y}=\{0,1\}^l$. Each token annotation y_i in $y=(y_1,...,y_l)$ is binary, where $y_i=1$ suggests that t_i participates in the bug, while $y_i=0$ the opposite. Hence, bug localization can be viewed as a sequence tagging problem on source code. Bug locators take x as input and binarily tags each t_i.

In this paper, as a very early step in this area, we focus on the fine-grained synthesized bugs, where the bug is caused by one or two tokens, for example, VarMisuse by a single variable misuse, BiOpMisuse by a single bi-operator misuse, and BoundError by a single inequality operator misuse. We propose and evaluate WELL on these datasets. The proposed weakly supervised WELL has potential to be extended to detect and locate other more complex bugs. We leave it for future work.

3.1.5 | Existing strongly supervised bug localization

Existing approaches are mainly based on the Seq2Ptr framework.[14–16] The DL model is trained on the buggy-location-annotated datasets with strong supervision. Concretely, the model encodes the source code, and computes attentions based on the token representations. The "pointer" points to the token with the highest attention score as the predicted buggy location, and for nonbuggy (clean) code, the "pointer" points to a special "clean" token inserted in the sequence.

3.2 | Attention in transformer

The attention technique is first proposed in Seq2Seq model,[42] which aims to selectively focus on parts of the source sequence during prediction. The attention function maps multiple queries and a set of key-value pairs to a weighted sum of the values, where the weight assigned to each value is computed with the query and the corresponding key.[43]

20477481, 2024, 9, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/smr.2669 by Peking University Health, Wiley Online Library on [05/01/2025]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

20477481, 2024, 9, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/smr.2669 by Peking University Health, Wiley Online Library on [05/01/2025]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

3.2.1 | Query, key, and value

The queries, the keys and the values are all vectors. The queries $Q=(Q_1,\ldots,Q_n)$ are the points of interest, the values $V=(V_1,\ldots,V_l)$ are the vectors to be weighted, and the keys $K=(K_1,\ldots,K_l)$ are the "descriptions" of V. Note that each K_i and V_i are paired. We query Q_i against the key K_j about how much portion of the value V_j should be included in the weighted-sum output.

3.2.2 | Scaled self-attention

The scaled self-attention measures how relevant the tokens in the given sequence x are to each other. Therefore, Q,K and V are all generated from X, where $X\in\mathbb{R}^{l\times d}$ is the embedding matrix of x and d is the dimension of the embedding space. Concretely, $Q=W_QX, K=W_KX$, and $V=W_VX$, where W_Q, W_K and W_V are trainable parameters. Scaled self-attention is computed as Equation (1), where $\alpha(Q,K)$ produces the probabilistic attention score, and σ is the softmax function. α_{ij} (the element of the ith row and the jth column in α) is the attention score of query Q_i against key K_j. α_{ij} reflects how relevant t_i and t_j are in x. For other types of attention, please refer to the paper.[44]

$$\text{Attention}(Q,K,V)=\alpha(Q,K)V=\sigma\left(\frac{QK^T}{\sqrt{d}}\right)V \tag{1}$$

3.2.3 | Multihead attention

The multihead attention mechanism is proposed in transformer[43] to focus on different types of information in the sequence. It consists of multiple sets of scaled self-attention, and each attention is called a head, where the parameters in each head are not shared. The outputs of these heads are concatenated and linearly transformed to integrate the information and generate the final representation. The overall formulation of multihead attention in the transformer architecture is deducted as below:

$$\begin{aligned}\text{MultiHead}(X)&=\text{Concat}(\text{head}_1,\ldots,\text{head}_{n_h})W_O,\\ \textit{where}\,\text{head}_i&=\text{Attention}_i(Q_i,K_i,V_i),\end{aligned} \tag{2}$$

where n_h is the head number, and W_O, along with W_{Qi}, W_{Ki}, W_{Vi} in the ith head, are trainable parameters.

3.2.4 | Transformer layer

The transformer architecture[43] is composed of a stack of identical transformer layers. Each layer consists of a multihead attention and a fully connected neural network. These two modules are linked through the residual connection[45] and an additional layer normalization.[46] The ith layer takes X_i as input and computes X_{i+1} for the $i+1$th layer ($X_1=X$). Please refer to "Layer 1" in Figure 2 for the detailed structure of the transformer layer.

3.3 | Pre-trained models

With the rapid increase of the amount of accessible training data and computing power, the ways of using DL models have also changed greatly. Recently, researchers have demonstrated the state-of-the-art performance of large pre-trained models on various tasks across different domains including CV and NLP. Compared with traditional training from scratch, pre-training on general tasks and fine-tuning on specific downstream tasks result in a significant performance improvement. The idea of pre-training is introduced to the field of SE lately, resulting in the CodeBERT[19] model pre-trained for source code.

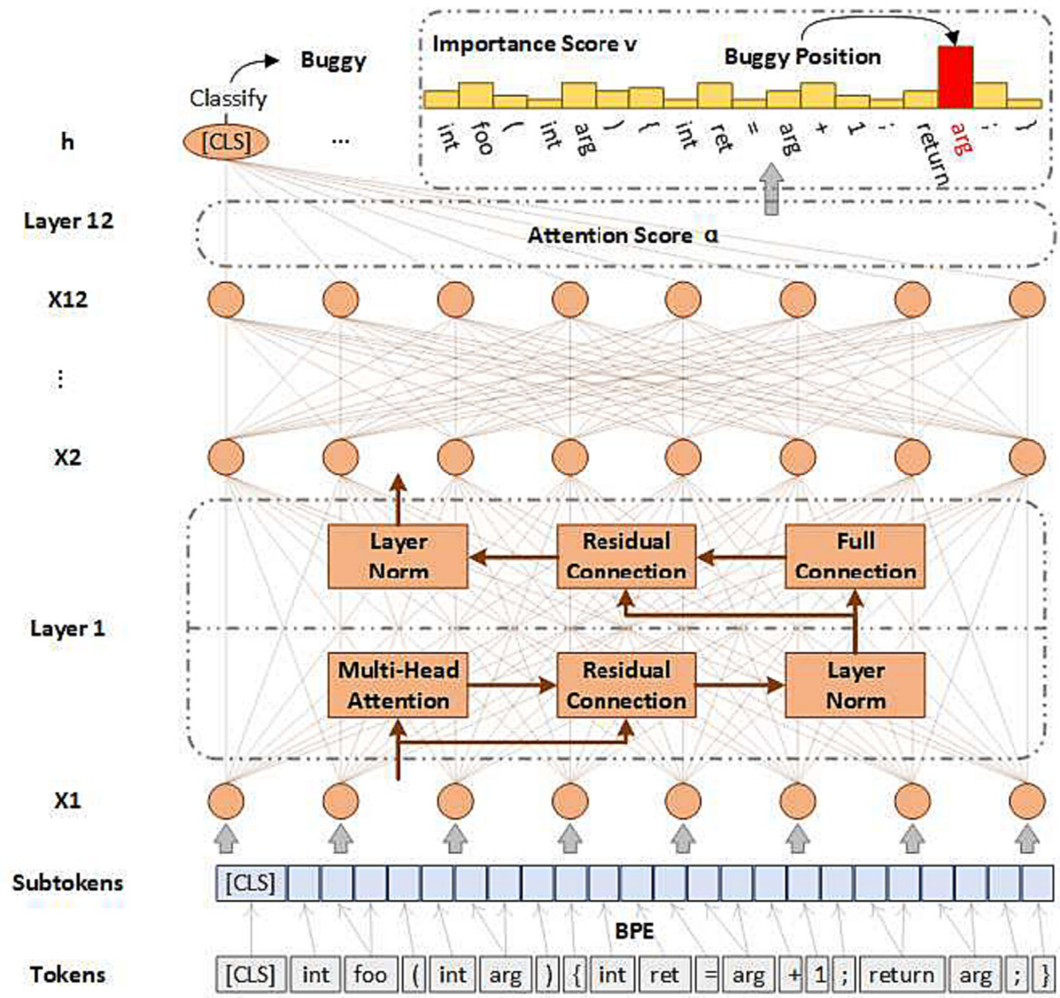


FIGURE 2 An illustrative example of WELL based on the CodeBERT backbone. We plot the computation flowchart of transformer layer only for "Layer 1" to save space. The top-left part is for bug detection, while the top right part is for bug localization.

3.3.1 | CodeBERT

Following the work of BERT[47] and RoBERTa[48] in NLP, Feng et al[19] propose CodeBERT for source code, which employs the identical architecture and model size with RoBERTa. The general architecture of CodeBERT is shown in Figure 2. Specifically, CodeBERT consists of 12 transformer layers (Layer 1 to 12). The hidden size, attention head number and feed-forward size in each layer are 768, 12, and 3072, respectively, leading to overall 125 million parameter size. CodeBERT is pre-trained on the CodeSearchNet dataset[49] with more than 8 million functions across six programming languages (i.e., Python, Java, JavaScript, Php, Ruby, and Go). After finetuning, CodeBERT is capable of performing code classification and code generation (as an encoder) tasks, and producing state-of-the-art results in various SE tasks.[50]

4 | WELL: WEAKLY SUPERVISED BUG LOCALIZATION

In this section, we illustrate the proposed WELL in detail. We first give a high-level overview of WELL in Section 4.1, which locates bugs through learning upon detection tasks in a weakly supervised manner. Then, we present details of training the model and locating bugs in Section 4.2 and Section 4.3. Finally, we extend WELL by leveraging a small amount of buggy-location-annotated data for validation when available.

2047481, 2024, 9, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/smr.2669 by Peking University Health, Wiley Online Library on [05/01/2025]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

4.1 | Overview of WELL

We first present the high-level idea of our proposed WELL, utilizing the CodeBERT model as the backbone to locate bugs in source code in a weakly supervised manner. It takes three steps to achieve weakly supervised learning: ① *finetuning* CodeBERT on bug detection datasets, ② *predicting* whether a piece of suspicious code is buggy or not, and ③ *locating* the buggy position based on the attention score. One may note that finetuning (or training) and predicting constitute the common learning paradigm of classification tasks such as bug detection. We follow this paradigm to obtain CodeBERT as bug detectors, and facilitate bug localization according to the attention score. Intuitively, if CodeBERT is capable to detect bugs, the code segment that draws the most attention should be related to the bug. Therefore, the multihead attention should reveal why and how CodeBERT detects bugs, and in this way, we may achieve weakly supervised bug localization.

WELL takes the source code token sequence as the input, and outputs the predicted buggy-or-not label and the bug location, as illustrated in Figure 2. Specifically, during localization, WELL first carries out binary classification upon the code with the finetuned CodeBERT ("Tokens" up to "h" in Figure 2). If the prediction is negative, WELL terminates, as the code is considered as clean (not buggy); otherwise, the algorithm continues. During prediction, the multihead attention scores of the last layer (α) are also retrieved. WELL aggregates the multihead attention scores to compute the token-level importance score $v = (v_1,\ldots,v_l)$, where v_i suggests the likeliness of the token t_i in the code x to cause the bug.

4.2 | Learning to detect bugs

As aforementioned, WELL finetunes the CodeBERT model upon bug detection datasets, which provide weak supervision for bug localization. As binary labeled bug detection datasets are easily accessible, CodeBERT is completely capable to learn the buggy patterns in the source code. This section illustrates the finetuning and predicting steps.

4.2.1 | Forward computation

Forward computation means prediction in DL. According to the definition of bug detection in Section 3.1, CodeBERT takes the source code $x = (t_1,\ldots,t_l)$ as input and outputs $\tilde{y}$ as the predicted label. The forward computing process is demonstrated in Figure 2. Concretely, given the source code token sequence x, we split them into subtokens $s_1,s_2,\ldots,s_{l'}$ using BPE,[51] where l' is the length of the subtoken sequence. Special token $s_0 = [\mathrm{CLS}]$ for classification is also inserted at the very beginning of the subtoken sequence. Then, the 12 transformer layers generates the context-aware vector representation $h = h_0,\ldots,h_{l'}$ of subtokens.

We preserve only h_0 as the aggregated representation of the whole piece of code, and discard other h_i's. Finally, we feed h_0 into a feed-forward neural network with softmax output, producing the predicted probabilities $p = (p_0,p_1)^T$ of the two classes (p_1 for buggy and p_0 for clean). The binary prediction $\tilde{y}$ is made based on p_1, that is, $\tilde{y} = 1$ if $p_1 \geq 0.5$; otherwise, $\tilde{y} = 0$.

4.2.2 | Training objective

We finetune CodeBERT on the training set of bug detection via back propagation and gradient descent. The loss function is the commonly adopted cross-entropy loss for classification, which is formulated as below, where Θ_C refers to the weights in the CodeBERT bug detection model:

$$\min_{\Theta_C} \mathcal{L} = \mathbb{E}_{(x,y)\in\mathcal{D}^t}(-y\log p_1 - (1-y)\log p_0). \tag{3}$$

4.3 | Localization via multihead attention

Empirically, the CodeBERT bug detector obtained in Section 4.2 makes prediction based on some certain features embedded within the code. Such features, which lead to buggy prediction, are likely to be related to the bugs. On the other hand, the multihead attention provides which parts of the source code the CodeBERT model focuses on. By analyzing the multihead attention, we may locate bugs via the CodeBERT bug detector. The algorithm is presented in Algorithm 1.

20477481, 2024, 9, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/smr.2669 by Peking University Health, Wiley Online Library on [05/01/2025]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

4.3.1 | Forward computation

In order to determine whether the input code piece is buggy or not, and to retrieve the attention scores α for further localization procedures, a forward computation (Lines 1 to 4 in Algorithm 1) is necessary.

After retrieving α, WELL performs two aggregations (multihead and subtoken) to obtain the token-level importance score v.

4.3.2 | Aggregation of multiple heads

The attention score α (as a three-dimensional tensor) includes all attention heads from the last transformer layer in CodeBERT. The ith attention head is denoted as α_i, in which the jth row (α_{ij}) suggests the probabilistic attention of s_j to all other subtokens of the code. Specifically, α_{i0} suggests the importance of each subtoken for the classification from the ith head.

In order to take all heads into consideration, we adopt average aggregation of all the n_h heads as Line 8 in Algorithm 1:

$$\alpha' = \text{AGG}_\text{H}\text{EAD}(\alpha) = \sum_{i=1}^{n_h} \frac{\alpha_{i0}}{n_h}, \tag{4}$$

where the output α' is a sequence of probabilities suggesting the importance of each corresponding subtoken.

Algorithm 1 Localization algorithm by WELL.

Require: Source code $x = t_1, t_2, \ldots, t_l$

Ensure: Predicted label y and buggy fragment x_{buggy}.

```
1:  s_1, s_2, ..., s_l' ← BPE(t_1, t_2, ..., t_l)                  ▹ Tokenize the input code
2:  h, α ← CodeBERT([CLS], s_1, s_2, ..., s_l')
3:  p ← CLASSIFY(h_0)                                             ▹ Obtain prediction with CodeBERT
4:  ỹ ← argmax_i p_i
5:  if ỹ = 0 then                                                 ▹ Source code classified as nonbuggy
6:      return CLEAN, NONE
7:  else
8:      α' ← AGG_HEAD(α)                                          ▹ Aggregate importance scores from different attention heads
9:      a, b ← ALIGN(x, s)                                        ▹ Align subtokens with code tokens
10:     v ← AGG_SUBTOKEN(a, b, α')                                ▹ Sum subtokens' importance score
11:     k ← arg max_i Σ_{j=i}^{i+N−1} v_j                         ▹ Select the position with the highest importance score
12:     return BUGGY, (t_k, ⋯, t_{k+N−1})
13: end if
```

4.3.3 | Alignment of subtokens

Due to BPE, a token t_i may be splitted into multiple subtokens, and the concatenation of these subtokens makes the original t_i that is, $t_i = \text{CONCAT}(s_{a_i}, s_{a_i+1}, \ldots, s_{b_i})$, where a_i and b_i refer to the beginning and ending indices in the subtoken sequence of the token t_i, respectively. a and b, which are vectors constituted by a_i and b_i, are retrieved by aligning the token sequence x and the subtoken sequence s as Line 9 in Algorithm 1. Concretely, BPE in CodeBERT tags the subtokens with a special character "Ġ", referring to the beginning of a token. Therefore, we perform a scanning over x and s to collect a_i and b_i for each $t_i \in x$.

4.3.4 | Aggregation of subtokens

As α' refers to the subtoken-level importance score while we require the token-level importance score, an aggregation upon the subtokens is necessary.

20477481, 2024, 9, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/smr.2669 by Peking University Health, Wiley Online Library on [05/01/2025]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

After alignment, WELL carries out additive aggregation to map the attention from the subtokens to the corresponding tokens (Line 10 in Algorithm 1), resulting in the importance score $v = (v_1, ..., v_l)$. Each v_i is computed as $v_i = \sum_{j=a_i}^{b_i} \alpha'_j$.

4.3.5 | Localization

The importance score v_i suggests how informative t_i is to the buggy prediction of the CodeBERT bug detector. In other words, those tokens with high importance scores are more likely to be related to the bug. As the bug localization task can be treated as a sequence tagging problem over the source code (Section 3.1), WELL assumes the buggy fragment to be a consecutive token subsequence of length N, where N is a hyper-parameter. With this assumption, WELL utilizes a slide window of size N to compute the importance score of all fragments, and selects the one with the largest score as the buggy fragment (Line 11 in Algorithm 1). The buggy fragment $x_{buggy} = (x_k, ..., x_{k+N-1})$ is selected as

$$k = \arg\max_i \sum_{j=i}^{i+N-1} v_j. \tag{5}$$

Please note the two assumptions in Equation (5): ① The buggy tokens are no more than N, and ① the buggy tokens are consecutive. When the assumptions are not satisfied, we could expand WELL by adopting the threshold activated inconsecutive selection strategy. We leave this to the future work.

4.4 | Extension with fine-grained supervision

So far, we have illustrated all details in WELL, including the forward computation of the CodeBERT backbone, the finetuning protocol upon buggy-or-not datasets, and the localization process. In other words, WELL does not require nor rely on any buggy location annotations for training. However, in the real world, although the fine-grained well-annotated examples are hard to collect, we can still obtain a small amount of them by all means. We introduce the extended WELL, namely, WELL-k, in this section, which leverages these fine-grained buggy-location-annotated examples for validation.

Even though multihead attention is designed to focus on different features in the input sequence,[43] recent researches have shown that only a small subset of the heads are specialized for the downstream task, while the other heads are dispensable and can even be pruned without losing much performance.[52,53] Therefore, those unimportant heads in WELL may have negative impacts after the average aggregation, which we indeed encounter during our experiments. On the other hand, aggregation of only the important heads is supposed to be competitive or even better. Therefore, WELL-k is proposed as an extension of WELL, which selects and aggregates only the top-k important attention heads.

To measure the importance of each attention head, we utilize the fine-grained well-annotated examples as validation. Instead of direct aggregation, we evaluate the bug localization performance of the ith attention head against the validation set (buggy-location-annotated), by setting $\alpha' = \alpha_{i0}$, creating WELL-H$_i$. The performance of WELL-H$_i$ is considered as the importance of the ith head.

Then, we aggregate only the top-k important heads, resulting in WELL-k. Note that WELL-1 refers to no attention head aggregation at all (using only the most important head), and WELL-n_h refers to WELL itself (aggregation of all heads). During our experiments, we evaluate WELL-1, considering the extreme case.

5 | EVALUATION

We implement WELL based on the PyTorch framework (version 1.7.1) and the transformers package (version 3.4.0). With WELL, we carry out evaluations to answer the following research questions.

5.1 | RQ1: Effectiveness of weak supervision

Can the weak supervision from bug detection enable bug localization? Furthermore, compared with strongly supervised approaches, how well can weakly supervised WELL perform?

2047481, 2024, 9, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/smr.2669 by Peking University Health, Wiley Online Library on [05/01/2025]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

5.2 | RQ2: Nontrivial bug localization

Is WELL capable to locate bugs in a more complex scenario? To what extent can weak supervision bridge bug detection and localization in a rather nontrivial environment?

5.3 | RQ3: Impact of multihead

Do all attention heads benefit weakly supervised bug localization? Does the extension design of WELL-1 make sense? Furthermore, can we improve the weakly supervised WELL with a small amount of fine-grained well-annotated data?

5.4 | RQ4: Portability to other backbone

Does the ability of localization come from the powerful transformer backbone? Is weak supervision feasible for bug localization when we switch to another weaker backbone, such as LSTM?

5.5 | Experimental settings

5.5.1 | Subject tasks and datasets

We evaluate WELL, against other approaches, with three synthetic bug detection/localization tasks, that is, VarMisuse,[13] BiOpMisuse,[16] BoundError, and a student program bug localization task (StuBug). VarMisuse[13] is a variable misuse detection and localization dataset of Python2 functions. The task is to detect whether there is a misused variable name in each function and locate it. BiOpMisuse is an operator misuse detection benchmark of Python2 functions proposed by Kanade et al.[16] Bugs are introduced by substituting one bi-operator with a wrong but type-compatible random one (e.g., "+"⇔"-", "*"⇔"/", "is"⇔"is not"). We extend this task to further bug localization by requiring the models to locate the substituted operator. BoundError is a boundary condition error detection and localization dataset of Java methods proposed in this paper. The off-by-one bug is brought into Java methods by adding/remove the equal condition in binary comparison operators. The details of obtaining this dataset are described in Appendix A. The bug types in the three synthetic datasets are different, but all of them are caused by one or two tokens (e.g., "is not"⇔"is" in BiOpMisuse).

StuBug[54] is a bug localization dataset of student programs written in the C language. The dataset is collected from 28,331 student submissions for 29 programming tasks with 231 test cases in total. During training, models learn to predict whether a program can pass a test case, and therefore, the input is a (program, test case) pair. For localization evaluation, models need to locate the buggy lines in the programs in the test set. The localization granularity is at the code line level. Please refer to the original paper[54] for more details. The statistical and other information about the datasets is listed in Table 2.

5.5.2 | Baseline models

For the synthetic datasets, state-of-the-art DL-based solutions for bug localization are mostly Seq2Ptr architecture.[14] Seq2Ptr employs the DL encoders to compute attention queried by a trainable vector upon the input code piece, and the pointer to the buggy position is generated from the attention (argmax). Different from our proposed WELL, Seq2Ptr models are trained with the bug localization dataset with strong buggy-location-annotated supervision signals. We employ previously proposed state-of-the-art models as the baseline, including GREAT[15] and

TABLE 2 Information of the subject tasks and datasets.

Dataset	Language	Train #	Valid #	Test #	Average length	Well balanced
VarMisuse	Python2	∼1.6 M	∼170k	∼886k	73	✓
BiOpMisuse	Python2	∼460k	∼49k	∼250k	123	✓
BoundError	Java	∼180k	∼26k	∼52k	146	✓
StuBug	C	∼235k	∼26k	∼17k	170	✓

20477481, 2024, 9, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/smr.2669 by Peking University Health, Wiley Online Library on [05/01/2025]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

CuBERT.[16] For experiments on StuBug, we employ NeuralBugLocator,[54] two state-of-the-art program-spectrum based approach[55,56] and one syntactic difference based solution as the baseline. The implementation details (including hyperparameters) are described in Appendix B.

5.5.3 | Evaluation metrics

For synthetic datasets, we employ *classification accuracy* (Acc_D), *precision* (P_D), and *recall* (R_D) as the evaluation metrics for bug detection and *localization accuracy* (Acc_L) for bug localization. Specifically, as bugs in the synthetic datasets are caused by only one or two tokens, we employ accuracy, that is, whether the localization model hits or misses the buggy token(s), as the performance indicator. Such localization accuracy is also employed in the previous studies.[15,16]

For StuBug, we employ line-level top-*k* ($k = 1,5,10$) localization accuracy as the metric, following the original paper.[54] If the top-*k* suspicious code lines contain the actual buggy line, the localization model is considered as accurate; otherwise, there is a mistake.

5.6 | RQ1: Effectiveness of WELL

To demonstrate WELL's effectiveness in detecting and locating bugs, we evaluate the performance of WELL on the three synthetic datasets and compare it with the baselines.

5.6.1 | Bug detection

The precision (P_D), recall (R_D), and accuracy (Acc_D) of WELL and baseline models for detecting VarMisuse, BiOpMisuse, and BoundError bugs are listed in Tables 3–5. Note that WELL-1 and WELL share the same finetuned CodeBERT backbone, the detection performance is identical. On average, WELL produces 92.85% precision, 93.21% recall, and 93.33% accuracy in detecting the three types of bugs. The performance of WELL on bug detection is significantly better than GREAT and CuBERT, which is attributed to the powerful CodeBERT backbone.

5.6.2 | Bug localization

On average, WELL and WELL-1 correctly locate 79.74% and 87.57% of bugs as shown in Tables 3–5. Compared with random picking, whose accuracy is about 1%, WELL is able to locate the three kinds of bugs, even though it has no direct supervision signals for localization during finetuning. To our surprise, the weakly supervised WELL is comparable to state-of-the-art supervised models, GREAT and CuBERT. Specifically, the localization accuracy of WELL is 6.75% higher than GREAT in VarMisuse, and 4.06% higher than CuBERT. As for BiOpMisuse and BoundError, the results of WELL-1 are only 2% lower than CuBERT. In addition, we notice that WELL and WELL-1 produce similar accuracy in VarMisuse and BiOpMisuse (with differences less than 0.5%), but WELL-1 outperforms WELL by 23.32% accuracy in BoundError. This phenomenon will be investigated and discussed in Section 5.8 later.

TABLE 3 Bug detection/localization results on VarMisuse.

Sup. sig.		Model	Detection			Localization
D[a]	L[b]		P (%)	R (%)	Acc. (%)	Acc. (%)
✓	✓	GREAT	91.38	91.69	89.91	85.53
✓	✓	CuBERT	93.53	91.76	92.69	88.22
✓	✗	WELL	**94.34**	**96.12**	**95.20**	**92.28**
✓	✗	WELL-1	**94.34**	**96.12**	**95.20**	92.08

[a]D = "Detection supervision."
[b]L = "Localization supervision."

20477481, 2024, 9, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/smr.2669 by Peking University Health, Wiley Online Library on [05/01/2025]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

5.6.3 | Case study

Figure 3 shows two cases from VarMisuse and BiOpMisuse separately. WELL predicts them as buggy correctly and locates the bugs accurately. The darker background of a token (t_i) refers to the higher importance score (v_i) from WELL of this token. The most important tokens (darkest) and the buggy locations (red circle) coincide in the figures, which, to a certain extent, shows the effectiveness of the weakly supervised WELL. On the other hand, important tokens with dark backgrounds are scarce and concentrated in the figures, meaning that CodeBERT in WELL is capable of learning the relation between the buggy position in the code and the given buggy-or-not supervised signal during finetuning. This furthermore demonstrates the feasibility and effectiveness of weak supervision in bug localization. Please refer to Appendix D for more visualized cases.

> **Answer to RQ1:** The experiment results suggest that WELL is feasible and effective in detecting and locating bugs upon the three synthetic datasets. Furthermore, the weakly supervised WELL even achieves comparable performance to the state-of-the-art supervised solutions.

TABLE 4 Bug detection/localization results on BiOpMisuse.

Sup. sig.		Model	Detection			Localization
D[a]	L[b]		P (%)	R (%)	Acc. (%)	Acc. (%)
✓	✓	GREAT	82.32	81.98	82.92	76.53
✓	✗	CuBERT	86.64	88.66	87.49	**85.14**
✓	✗	WELL	**92.70**	**90.50**	**91.71**	83.08
✓	✗	WELL-1	**92.70**	**90.50**	**91.71**	83.44

[a]D = "Detection supervision."
[b]L = "Localization supervision."

TABLE 5 Bug detection/localization results on BoundError.

Sup. sig.		Model	Detection			Localization
D[a]	L[b]		P (%)	R (%)	Acc. (%)	Acc. (%)
✓	✓	GREAT	85.87	88.11	87.08	84.77
✓	✓	CuBERT	90.12	92.36	91.12	**90.10**
✓	✗	WELL-lstm	89.00	90.40	89.63	34.96
✓	✗	WELL	**91.50**	**93.00**	**93.10**	63.87
✓	✗	WELL-1	**91.50**	**93.00**	**93.10**	87.19

[a]D = "Detection supervision."
[b]L = "Localization supervision."

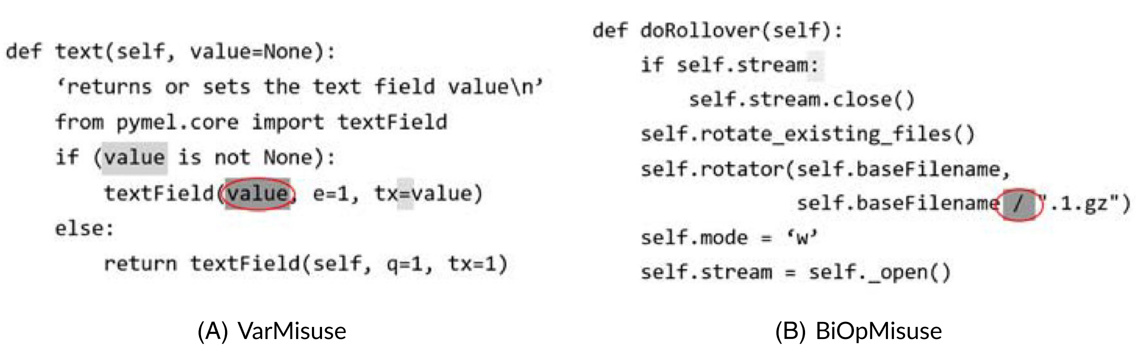


FIGURE 3 Visualization of the importance score produced by WELL. All examples are correctly handled by WELL. The red circle suggests the buggy location of ground-truth. The gray-scale of the background represents the importance score of the corresponding token by WELL.

2047481, 2024, 9, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/smr.2669 by Peking University Health, Wiley Online Library on [05/01/2025]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

5.7 | RQ2: Nontrivial bug localization

We further evaluate the performance of WELL by locating semantic bugs in student programs and compare WELL with existing state-of-the-art deep models NBL,[54] two program-spectrum-based methods (Tarantula,[55] Ochiai[56]) and one syntactic difference based technique. We evaluate the methods on 1449 programs in the test set and list the results in Table 6.

5.7.1 | Nontrivial semantic bug localization

When reporting only 1 suspicious buggy line, WELL successfully locates bugs in 421 (29.05%) programs. The accuracy of the previous state-of-the-art deep model, NBL, is only 20.29% (~9% lower). WELL also outperforms the three traditional methods significantly. For top-5 and top-10 accuracy, the superiority of WELL is still clear. WELL gives state-of-the-art performance on student programs collected from real world.

5.7.2 | Case study

Two programs in StuBug are shown in Figure 4 with the importance scores given by WELL. The first program is a solution for calculating $a^b modc$, while the line "k = k * a" may cause integer overflow. The correct fix is "k = k * a % c". WELL ranks this line to the second place. The second figure is a program for calculating the slope of a line. However, the student forgets to write the "printf" statement to output the value of slope. WELL locates the position to fix the bug (i.e., add the "printf" statement) at the highlighted line successfully.

Answer to RQ2: Compared with the existing deep solution and traditional techniques, WELL is capable to locate nontrivial semantic bugs in StuBug.

TABLE 6 Bug localization results on StuBug.

Model	Localization Result		
	Top 10	Top 5	Top 1
Tarantula	1141 (78.74%)	791 (54.59%)	311 (21.46%)
Ochiai	1151 (79.43%)	835 (57.63%)	385 (26.57%)
Diff-based	623 (43.00%)	122 (8.42%)	0 (0.00%)
NBL	1164 (80.33%)	833 (57.49%)	294 (20.29%)
WELL	**1240** (85.58%)	**931** (64.25%)	**421** (29.05%)

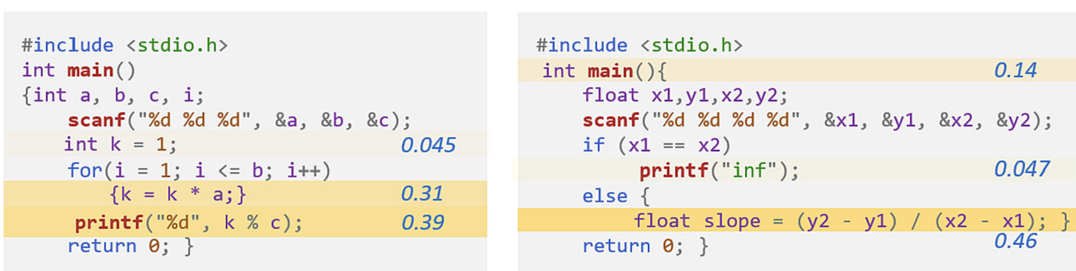


FIGURE 4 Visualization of WELL on StuBug. The three lines with highest importance scores are highlighted. The importance scores are labeled on the right with blue font.

20477481, 2024, 9, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/smr.2669 by Peking University Health, Wiley Online Library on [05/01/2025]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

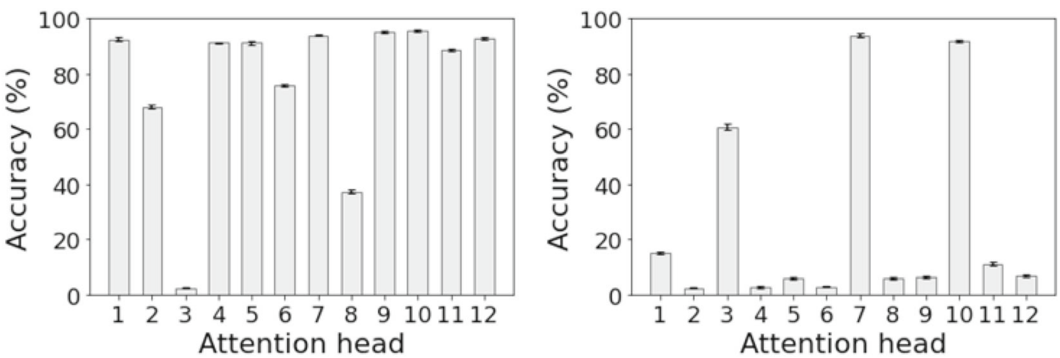
5.8 | RQ3: Impact of multihead

As aforementioned, we find that WELL abnormally loses 23.32% bug localization accuracy compared with WELL-1 in BoundError. It may be caused by the issue of multihead attention discovered in previous work.[52,53] We delve into this issue in the following paragraphs and justify the necessity of extension in WELL-1. To investigate the impact of different heads, we carry out an ablation study by randomly sampling 2000 correctly classified buggy examples from the test set to evaluate each WELL-H_i in VarMisuse and BoundError.

The localization accuracy of each WELL-H_i is shown in Figure 5. In BoundError, only three heads (3, 7, 10) are effective (>50%) for localization, while the rest are almost invalid. The results agree with the previous work that only a small subset of heads does the heavy lifting.[52,53] On the contrary, in VarMisuse, almost all heads are beneficial for localization, and only a few (3, 8) are invalid. One possible reason is that the data size of VarMisuse is much larger and WELL learns to focus on bugs better. This ablation study explains why WELL performs slightly better than WELL-1 in VarMisuse, but fails in BoundError. Because in BoundError, only several heads are effective while the others cause counteraction to the average aggregation, leading to an accuracy drop in WELL.

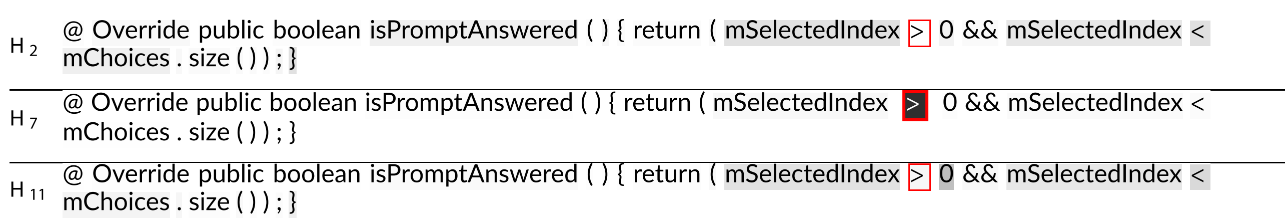
5.8.1 | Case study

Figure 6(a) presents a case visualization of WELL-H_i in BoundError. The gray-scale of the background refers to the importance score of the corresponding token, and the red box indicates the buggy location. WELL-H_7 is accurate in the demonstrated case, due to the validity of the 7th

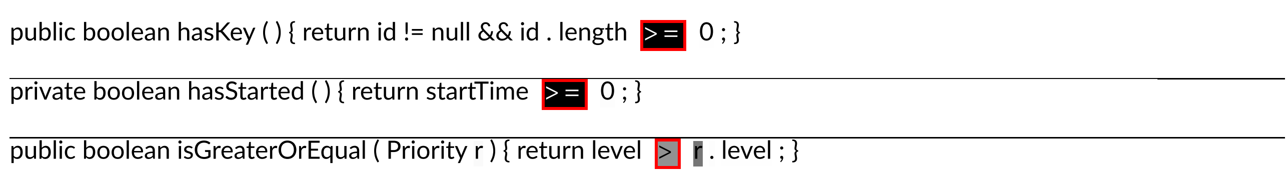


(A) VarMisuse (B) BoundError

FIGURE 5 Localization accuracy of WELL-H_i (attention head *i*). The histogram is the average of the five repeated trials, and the standard deviation is marked at the top of the histogram.



(A) Visualization ofimportance scores from WELL-H 2, H7 and H11 in BoundError. The gray-scale ofthe background indicates the importance score ofthe corresponding token, and WELL-H i predicts the token with darkest background as the buggy location. The red box refers to the ground-truth buggy location.



(B) Visualization of WELL-lstm in BoundError. The gray-scale ofthe background indicates the importance score ofthe corresponding token, and the red box refers to the ground-truth buggy location.

FIGURE 6 Visualization of WELL.

20477481, 2024, 9, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/smr.2669 by Peking University Health, Wiley Online Library on [05/01/2025]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

head according to Figure 5(b). As for the invalid heads (2 and 11), the visualized cases are distracted and erroneously predicted. In addition, WELL-H_7 is actually WELL-1 as the 7th head is the most important among all 12 heads. Therefore, the case study further demonstrates the feasibility of the extension in WELL-1. Please refer to Appendix E for more visualized cases.

Answer to RQ3: The "few-specialized" problem of multihead attention is also identified in WELL. The ablation study verifies the improvement of the WELL-1 extension, which leverages fine-grained annotations as validation to select the most important and specialized head among all heads.

5.9 | RQ4: Portability to other backbone

In the previous experiments, WELL shows great performance on bug localization. However, the effectiveness of WELL may come from the CodeBERT backbone rather than weak supervision. Therefore, we perform another ablation study, by applying weak supervision to the LSTM model, creating WELL-lstm. The backbone of WELL-lstm is a two-layer bi-directional LSTM. An attention layer[57] is placed on the top of LSTM, and we use it to compute the importance score v (no aggregation as it is single-headed). We train and evaluate WELL-lstm upon BoundError in a weakly supervised manner.

5.9.1 | Top-K localization

The localization accuracy of WELL-LSTM is 34.96%, which suggests that WELL-lstm is valid to locate BoundError bugs although it is not as effective compared with CodeBERT and strongly supervised approaches. The distracted attention may cause the rather not-effective-enough performance of WELL-lstm due to the insufficient model capability of LSTM. This is reasonable because the parameter size of LSTM (29M) is much smaller than CodeBERT (125M). Therefore, to further demonstrate the potential of weak supervision, we also evaluate the top-K accuracy, where WELL-lstm selects the top-K important segments as the predicted buggy location, and any hit among them is considered correct. The results are shown in Figure 7. The top-1 (exact) accuracy of WELL-lstm is 34.94%, much lower than WELL and WELL-1, while the top-2 accuracy rises rapidly to 76.04%. When we take four tokens into account, the accuracy is even 86.30%, which is close to the exact accuracy of WELL-1 (87.19%).

5.9.2 | Case study

Some visualized cases are presented in Figure 6(b), and more cases are shown in Appendix F.

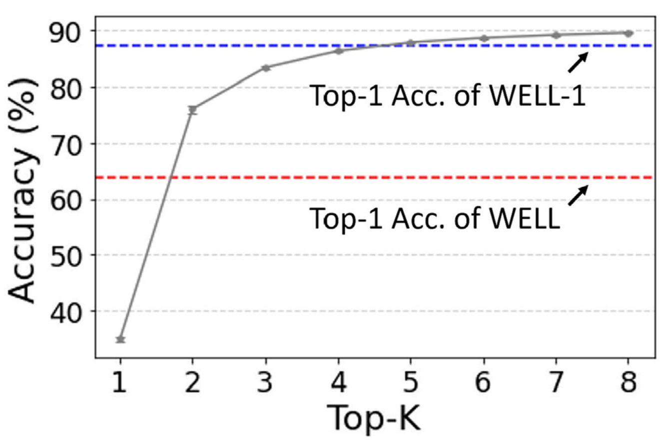


FIGURE 7 Top-K localization accuracy curve with standard deviation of WELL-lstm in BoundError. The standard deviation is too small and even imperceptible in the figure. The blue and red dashed lines are the top-1 localization accuracy of WELL-1 and WELL, respectively.

20477481, 2024, 9, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/smr.2669 by Peking University Health, Wiley Online Library on [05/01/2025]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

Although the LSTM backbone is much less powerful than CodeBERT, WELL-lstm is still capable to notice the buggy locations in code in the scenario of weak supervision. As a rough conclusion, the weakly supervised framework is portable and transferable to other attention-based models such as LSTM as well.

Answer to RQ4: Although the LSTM backbone is much less powerful than CodeBERT, WELL-lstm is still able to notice the buggy locations with only weak supervision. As a rough conclusion, the weakly supervised framework is portable and transferable to other attention-based models such as LSTM as well.

5.10 | Threats to validity

5.10.1 | Threat to external validity

The selection of subject datasets and baselines can be major threats to external validity. ① From the perspective of dataset, we perform experiments upon three synthetic datasets to demonstrate the effectiveness of WELL. To counteract the plausible trivialness, we have also evaluated WELL upon the nontrivial dataset, that is, StuBug, in **RQ2**. Results indicates that WELL is feasible to locate nontrivial semantic bugs. Still, further studies upon more complex and even real-world datasets are required to generalize our conclusion. ② From the perspective of baseline, we compare WELL with the currently state-of-the-art DL solutions (GREAT,[15] CuBERT,[16] and NBL[54]), along with traditional program-spectrum based (Tarantula[55] and Ochiai[56]) and syntactic difference-based approaches. However, to further facilitate generalizability of the conclusion, other bug localization approaches need to be compared.

5.10.2 | Threat to internal validity

The backbone selection of WELL is a threat to internal validity. The transformer architecture and the CodeBERT backbone may be too powerful and the weak supervision could be ineffective for other models. We counteract it by replacing the CodeBERT backbone with the LSTM architecture in the ablation of **RQ4**.

5.10.3 | Threat to construct validity

The evaluation metrics can be threats to construct validity. Localization accuracy is employed as the performance indicator for the synthetic datasets, which is adopted in the previous work.[15,16] As for StuBug, we employ line-level top-*k* accuracy as the evaluation metric, following the setting of the original paper.[54]

6 | CONCLUSION AND DISCUSSION

This paper proposes WELL, a weakly supervised bug localization model equipped with the powerful CodeBERT model, to alleviate the challenge of data collection and annotation. WELL is obtained on bug detection datasets without buggy-location annotations, making full usage of the more easily accessible training data. Through the in-depth evaluations, we demonstrate that WELL is capable of localizing bugs under coarse-grained supervision, and produces competitive or even better performance than existing state-of-the-art models across both synthetic and real-world datasets. Further experiments show that the weakly supervised methodology in WELL can be effectively applied to other attention-based models.

Nonetheless, there are still many things that requires in-depth discussions and explorations in the future. ① The synthesized datasets adopted in this paper are rather simple. Although we have conducted experiments upon the nontrivial StuBug dataset, more complex and even real-world datasets need to be evaluated in the future. ② The current WELL has an assumption that the buggy fragment is consecutive and has a max length of *N*. This may not be satisfied in more complex bugs. The threshold activation strategy or the inconsecutive selection strategy are plausible solutions to this issue, which worth studying later. ③ WELL relies on the attention mechanism to obtain the importance score. It may hinder the utilization upon other backbone models, because not all architectures have attention. Besides attention, there are other potential measurements,

20477481, 2024, 9, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/smr.2669 by Peking University Health, Wiley Online Library on [05/01/2025]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

such as the gradient. It is quite possible to generalize WELL to other importance metrics, and we leave these trials of different approaches to compute the importance scores for our future work. ④ A more desirable future work is to further fix bugs in the weakly supervised style, which could also be achieved with the CodeBERT backbone. Because CodeBERT is a pre-trained language model, it has the capability to produce correct code snippets. The weakly supervised bug fixing may lead desirable and promising future research work.

In short words, as an early step, we demonstrate the feasibility and effectiveness of weakly supervised bug localization in this paper, and we hope this work could introduce some new ideas and methodologies into the SE community.

ORCID

Huangzhao Zhang https://orcid.org/0000-0002-0324-4591
Zhuo Li https://orcid.org/0000-0002-0198-2304

REFERENCES

1. Mou L, Li G, Zhang L, Wang T, Jin Z. Convolutional neural networks over tree structures for programming language processing. In: Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence, February 12-17, 2016, Phoenix, Arizona, USA Schuurmans D, Wellman MP, eds. AAAI Press; 2016:1287-1293.
2. Zhang J, Wang X, Zhang H, Sun H, Wang K, Liu X. A novel neural source code representation based on abstract syntax tree. In: Proceedings of the 41st International Conference on Software Engineering, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019 Atlee JM, Bultan T, Whittle J, eds. IEEE / ACM; 2019:783-794.
3. Yu H, Lam W, Chen L, Li G, Xie T, Wang Q. Neural detection of semantic code clones via tree-based convolution. In: Proceedings of the 27th International Conference on Program Comprehension, ICPC 2019, Montreal, QC, Canada, May 25-31, 2019 Guéhéneuc Y-G, Khomh F, Sarro F, eds. IEEE / ACM; 2019:70-80.
4. Wang W, Li G, Ma B, Xia X, Jin Z. Detecting code clones with graph neural network and flow-augmented abstract syntax tree. In: 27th IEEE International Conference on Software Analysis, Evolution and Reengineering, SANER 2020, London, ON, Canada, February 18-21, 2020 Kontogiannis K, Khomh F, Chatzigeorgiou A, Fokaefs M-E, Zhou M, eds. IEEE; 2020:261-271.
5. Allamanis M, Peng H, Sutton CA. A convolutional attention network for extreme summarization of source code. In: Proceedings of the 33nd International Conference on Machine Learning, ICML 2016, New York City, NY, USA, June 19-24, 2016 Balcan M-F, Weinberger KQ, eds., JMLR Workshop and Conference Proceedings, vol. 48. JMLR.org; 2016:2091-2100.
6. Alon U, Zilberstein M, Levy O, Yahav E. code2vec: learning distributed representations of code. *Proc ACM Program Lang*. 2019;3(POPL):40:1-40:29.
7. Li J, Wang Y, King I, Lyu MR. Code completion with neural attention and pointer networks. CoRR abs/1711.09573; 2017.
8. Liu F, Li G, Wei B, Xia X, Fu Z, Jin Z. A self-attentional neural architecture for code completion with multi-task learning. In: ICPC '20: 28th International Conference on Program Comprehension, Seoul, Republic of Korea, July 13-15, 2020. ACM; 2020:37-47.
9. Liu F, Li G, Zhao Y, Jin Z. Multi-task learning based pre-trained language model for code completion. In: 35th IEEE/ACM International Conference on Automated Software Engineering, ASE 2020, Melbourne, Australia, September 21-25, 2020. IEEE; 2020:473-485.
10. Hu X, Li G, Xia X, Lo D, Jin Z. Deep code comment generation. In: Proceedings of the 26th Conference on Program Comprehension, ICPC 2018, Gothenburg, Sweden, May 27-28, 2018 Khomh F, Roy CK, Siegmund J, eds. ACM; 2018:200-210.
11. Hu X, Li G, Xia X, Lo D, Lu S, Jin Z. Summarizing source code with transferred API knowledge. In: Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence, IJCAI 2018, July 13-19, 2018, Stockholm, Sweden Lang J, ed. ijcai.org; 2018:2269-2275.
12. Alon U, Brody S, Levy O, Yahav E. code2seq: Generating sequences from structured representations of code. In: 7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019. OpenReview.net; 2019.
13. Allamanis M, Brockschmidt M, Khademi M. Learning to represent programs with graphs. In: 6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings. OpenReview.net; 2018.
14. Vasic M, Kanade A, Maniatis P, Bieber D, Singh R. Neural program repair by jointly learning to localize and repair. In: 7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019. OpenReview.net; 2019.
15. Hellendoorn VJ, Sutton C, Singh R, Maniatis P, Bieber D. Global relational models of source code. In: 8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020. OpenReview.net; 2020.
16. Kanade A, Maniatis P, Balakrishnan G, Shi K. Learning and evaluating contextual embedding of source code. In: Proceedings of the 37th International Conference on Machine Learning, ICML 2020, 13-18 July 2020, Virtual Event, Proceedings of Machine Learning Research, vol. 119. PMLR; 2020: 5110-5121.
17. Benton S, Ghanbari A, Zhang L. Defexts: a curated dataset of reproducible real-world bugs for modern JVM languages. In: Proceedings of the 41st International Conference on Software Engineering: Companion Proceedings, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019 Atlee JM, Bultan T, Whittle J, eds. IEEE / ACM; 2019:47-50.
18. Lutellier T, Pham HV, Pang L, Li Y, Wei M, Tan L. Coconut: combining context-aware neural translation models using ensemble for program repair. In: ISSTA '20: 29th ACM SIGSOFT International Symposium on Software Testing and Analysis, Virtual Event, USA, July 18-22, 2020 Khurshid S, Pasareanu CS, eds. ACM; 2020:101-114.
19. Feng Z, Guo D, Tang D, Duan N, Feng X, Gong M, Shou L, Qin B, Liu T, Jiang D, Zhou M. Codebert: A pre-trained model for programming and natural languages. In: Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: Findings, EMNLP 2020, Online Event, 16-20 November 2020 Cohn T, He Y, Liu Y, eds. Association for Computational Linguistics; 2020:1536-1547.
20. Wang S, Liu T, Tan L. Automatically learning semantic features for defect prediction. In: Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016 Dillon LK, Visser W, Williams LA, eds. ACM; 2016:297-308.
21. Choi M-J, Jeong S, Oh H, Choo J. End-to-end prediction of buffer overruns from raw source code via neural memory networks. In: Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI 2017, Melbourne, Australia, August 19-25, 2017 Sierra C, ed. ijcai.org; 2017:1546-1553.

20477481, 2024, 9, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/smr.2669 by Peking University Health, Wiley Online Library on [05/01/2025]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

22. Li Z, Zou D, Xu S, Ou X, Jin H, Wang S, Deng Z, Zhong Y. Vuldeepecker: A deep learning-based system for vulnerability detection. In: 25th Annual Network and Distributed System Security Symposium, NDSS 2018, San Diego, California, USA, February 18-21, 2018. The Internet Society; 2018.
23. Pradel M, Sen K. Deepbugs: a learning approach to name-based bug detection. *Proc ACM Program Lang*. 2018;2(OOPSLA):147:1-147:25.
24. Zhou Z-H. A brief introduction to weakly supervised learning. *Natl Sci Rev*. 2017;5(1):44-53.
25. Dai J, He K, Sun J. Boxsup: Exploiting bounding boxes to supervise convolutional networks for semantic segmentation. In: 2015 IEEE International Conference on Computer Vision, ICCV 2015, Santiago, Chile, December 7-13, 2015. IEEE Computer Society; 2015:1635-1643.
26. Papandreou G, Chen L-C, Murphy K, Yuille AL. Weakly- and semi-supervised learning of a DCNN for semantic image segmentation. CoRR abs/1502.02734; 2015.
27. Lin D, Dai J, Jia J, He K, Sun J. Scribblesup: Scribble-supervised convolutional networks for semantic segmentation. In: 2016 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2016, Las Vegas, NV, USA, June 27-30, 2016. IEEE Computer Society; 2016:3159-3167.
28. Bearman AL, Russakovsky O, Ferrari V, Li F-F. What's the point: Semantic segmentation with point supervision. In: Computer vision - ECCV 2016 - 14th European Conference, Amsterdam, The Netherlands, October 11-14, 2016, Proceedings, Part VII Leibe B, Matas J, Sebe N, Welling M, eds., Lecture Notes in Computer Science, vol. 9911. Springer; 2016:549-565.
29. Lin M, Chen Q, Yan S. Network in network. In: 2nd International Conference on Learning Representations, ICLR 2014, Banff, AB, Canada, April 14-16, 2014, Conference Track Proceedings Bengio Y, LeCun Y, eds.; 2014.
30. Zhou B, Khosla A, Lapedriza A, Oliva A, Torralba A. Learning deep features for discriminative localization. In: 2016 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2016, Las Vegas, NV, USA, June 27-30, 2016. IEEE Computer Society; 2016:2921-2929.
31. Selvaraju RR, Cogswell M, Das A, Vedantam R, Parikh D, Batra D. Grad-cam: Visual explanations from deep networks via gradient-based localization. In: IEEE International Conference on Computer Vision, ICCV 2017, Venice, Italy, October 22-29, 2017. IEEE Computer Society; 2017:618-626.
32. Wei Y, Xiao H, Shi H, Jie Z, Feng J, Huang TS. Revisiting dilated convolution: A simple approach for weakly- and semi-supervised semantic segmentation. In: 2018 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2018, Salt Lake City, UT, USA, June 18-22, 2018. IEEE Computer Society; 2018:7268-7277.
33. Ni J, Dinu G, Florian R. Weakly supervised cross-lingual named entity recognition via effective annotation and representation projection. In: Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics, ACL 2017, Vancouver, Canada, July 30 - August 4, Volume 1: Long Papers Barzilay R, Kan M-Y, eds. Association for Computational Linguistics; 2017:1470-1480.
34. Patra B, Moniz JRA. Weakly supervised attention networks for entity recognition. 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 Inui K, Jiang J, Ng V, Wan X, eds. Association for Computational Linguistics; 2019:6267-6272.
35. Lison P, Barnes J, Hubin A, Touileb S. Named entity recognition without labelled data: A weak supervision approach. In: Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics, ACL 2020, Online, July 5-10, 2020 Jurafsky D, Chai J, Schluter N, Tetreault JR, eds. Association for Computational Linguistics; 2020:1518-1533.
36. Safranchik E, Luo S, Bach SH. Weakly supervised sequence tagging from noisy rules. 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; 2020:5570-5578.
37. Lei T, Barzilay R, Jaakkola TS. Rationalizing neural predictions. In: Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing, EMNLP 2016, Austin, Texas, USA, November 1-4, 2016 Su J, Carreras X, Duh K, eds. The Association for Computational Linguistics; 2016:107-117.
38. Bastings J, Aziz W, Titov I. Interpretable neural predictions with differentiable binary variables. 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 Korhonen A, Traum DR, Màrquez L, eds. Association for Computational Linguistics; 2019:2963-2977.
39. Yu M, Chang S, Zhang Y, Jaakkola TS. Rethinking cooperative rationalization: Introspective extraction and complement control. 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 Inui K, Jiang J, Ng V, Wan X, eds. Association for Computational Linguistics; 2019: 4092-4101.
40. DeYoung J, Jain S, Rajani NF, Lehman E, Xiong C, Socher R, Wallace BC. ERASER: A benchmark to evaluate rationalized NLP models. In: Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics, ACL 2020, Online, July 5-10, 2020 Jurafsky D, Chai J, Schluter N, Tetreault JR, eds. Association for Computational Linguistics; 2020:4443-4458.
41. Jain S, Wiegreffe S, Pinter Y, Wallace BC. Learning to faithfully rationalize by construction. In: Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics, ACL 2020, Online, July 5-10, 2020 Jurafsky D, Chai J, Schluter N, Tetreault JR, eds. Association for Computational Linguistics; 2020:4459-4473.
42. Bahdanau D, Cho K, Bengio Y. Neural machine translation by jointly learning to align and translate. In: 3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, Usa, May 7-9, 2015, Conference Track Proceedings Bengio Y, LeCun Y, eds.; 2015.
43. Vaswani A, Shazeer N, Parmar N, Uszkoreit J, Jones L, Gomez AN, Kaiser L, Polosukhin I. Attention is all you need. In: Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA Guyon I, von Luxburg U, Bengio S, Wallach HM, Fergus R, Vishwanathan SVN, Garnett R, eds.; 2017:5998-6008.
44. Luong T, Pham H, Manning CD. Effective approaches to attention-based neural machine translation. In: Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing, EMNLP 2015, Lisbon, Portugal, September 17-21, 2015 Màrquez L, Callison-Burch C, Su J, Pighin D, Marton Y, eds. The Association for Computational Linguistics; 2015:1412-1421.
45. He K, Zhang X, Ren S, Sun J. Deep residual learning for image recognition. In: 2016 IEEE conference on computer vision and pattern recognition, CVPR 2016, las vegas, nv, usa, june 27-30, 2016. IEEE Computer Society; 2016:770-778.
46. Ba LJ, Kiros JR, Hinton GE. Layer normalization. http://arxiv.org/abs/1607.06450; 2016.
47. Devlin J, Chang M-W, Lee K, Toutanova K. BERT: pre-training of deep bidirectional transformers for language understanding. In: Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT 2019, Minneapolis, MN, Usa, June 2-7, 2019, Volume 1 (Long and Short Papers) Burstein J, Doran C, Solorio T, eds. Association for Computational Linguistics; 2019:4171-4186.

20477481, 2024, 9, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/smr.2669 by Peking University Health, Wiley Online Library on [05/01/2025]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

48. Liu Y, Ott M, Goyal N, Du J, Joshi M, Chen D, Levy O, Lewis M, Zettlemoyer L, Stoyanov V. Roberta: A robustly optimized BERT pretraining approach. http://arxiv.org/abs/1907.11692; 2019.
49. Husain H, Wu H-H, Gazit T, Allamanis M, Brockschmidt M. Codesearchnet challenge: Evaluating the state of semantic code search. http://arxiv.org/abs/1909.09436; 2019.
50. Lu S, Guo D, Ren S, et al. Codexglue: A machine learning benchmark dataset for code understanding and generation. arXiv preprint arXiv:210204664; 2021.
51. Sennrich R, Haddow B, Birch A. Neural machine translation of rare words with subword units. In: Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics, ACL 2016, August 7-12, 2016, Berlin, Germany, Volume 1: Long Papers. The Association for Computer Linguistics; 2016.
52. Voita E, Talbot D, Moiseev F, Sennrich R, Titov I. Analyzing multi-head self-attention: Specialized heads do the heavy lifting, the rest can be pruned. 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 Korhonen A, Traum DR, Màrquez L, eds. Association for Computational Linguistics; 2019:5797-5808.
53. Michel P, Levy O, Neubig G. Are sixteen heads really better than one? In: Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NEURIPS 2019, December 8-14, 2019, Vancouver, BC, Canada Wallach HM, Larochelle H, Beygelzimer A, d'Alché-Buc F, Fox EB, Garnett R, eds.; 2019:14014-14024.
54. Gupta R, Kanade A, Shevade SK. Neural attribution for semantic bug-localization in student programs. In: Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NEURIPS 2019, December 8-14, 2019, Vancouver, BC, Canada; 2019:11861-11871.
55. Abreu R, Zoeteweij P, van Gemund AJC. An evaluation of similarity coefficients for software fault localization. In: PRDC. IEEE Computer Society; 2006:39-46.
56. Jones JA, Harrold MJ, Stasko J. Visualization for fault localization; 2003.
57. Wang Z, Yang B. Attention-based bidirectional long short-term memory networks for relation classification using knowledge distillation from BERT. In: Dasc/picom/cbdcom/cyberscitech. IEEE; 2020:562-568.

How to cite this article: Zhang H, Li Z, Li J, Jin Z, Li G. WELL: Applying bug detectors to bug localization via weakly supervised learning. *J Softw Evol Proc*. 2024;36(9):e2669. https://doi.org/10.1002/smr.2669

APPENDIX A: BOUNDERROR CONSTRUCTION

We randomly sample 20% projects from the Github Java Corpus,[†] and extract all the Java methods using TreeSitter [‡]. The methods with more than 400 tokens are discarded as they are too long and complicated. Then we use TreeSitter to parse the methods to get the Concrete Syntax Tree (CST), and locate the subtree with type “binary expression” to find binary expressions with operators “$<=$”, “$>=$”, $<$ and $>$. The off-by-one bug is brought into the methods by replacing comparison operators (e.g., $< \Leftrightarrow <=$, “$>$”$\Leftrightarrow$“$>=$”[§]). [¶]

APPENDIX B: MODEL CONFIGURATION

We implement WELL with Python3 based on the DL framework PyTorch (ver 1.7.1) and the transformers package (version 3.4.0).[#]

WELL adopts the released base version of CodeBERT (CodeBERT-base) as the backbone.[‖] The max length is fixed to 512. We utilize the open sourced GREAT model with the same configuration reported in the original paper and train it from scratch. As for CuBERT, we reproduce the model by replacing the backbone model with CodeBERT. The reasons are as follows: ❶ The model size of CuBERT is (three times larger than CodeBERT). It is too large for our machine to finetune the model. ❷ Using the same CodeBERT backbone helps us to compare the method with WELL better. During finetuning, the learning rate is set to 4×10^{-5}, and L_2 regularization is adopted with the weight of 0.01. In each experiment, the models are trained/finetuned for 6 epochs with the batch size of 64 and we select the checkpoint with the highest accuracy on the validation set. To emphasize the supervision, the buggy locations in the training set is accessible for GREAT, CuBERT, while blocked for WELL (weak supervision). For WELL-1, we sample 1000 fine-grained labeled examples from the validation dataset to measure the importance of each attention head, which is a very small amount (less than 1% of the original training dataset size). For WELL-LSTM, we train a bi-directional LSTM model with

[†] https://groups.inf.ed.ac.uk/cup/javaGithub/.
[‡] https://tree-sitter.github.io/tree-sitter.
[§] The equal condition in the comparison is added or removed in the operator.
[¶] The original BoundError dataset is available in the open sourced repository.
[#] https://huggingface.co/transformers/.
[‖] https://huggingface.co/microsoft/codebert-base.

20477481, 2024, 9, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/smr.2669 by Peking University Health, Wiley Online Library on [05/01/2025]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

the hidden size of 600 and time step of 400. The vocabulary size is 30,000 and the embedding width is 512. In the experiment of StuBug, the results of baseline models (NeuralBugLocator, Turantula, Ochiai, and Diff-based model) are reported as in the original paper of NBL.

APPENDIX C: WELL FOR BUG FIXING

WELL is trained on bug detection datasets and can be applied for both bug detection and localization. Furthermore, WELL has the potential for unsupervised bug fixing. The backbone model of WELL is CodeBERT, which is pre-trained with masked language model task. So CodeBERT can predict the probability of the masked original token. Thus, when WELL predict a program to be buggy and find the bug location, we can mask the located buggy tokens and query CodeBERT to predict the original tokens. Theoretically, CodeBERT should recover the most probable and correct tokens. In this way, we may apply WELL for bug fixing.

However, there are still many challenges to accomplish this rough idea. For example, it is hard to determine the number of tokens to query CodeBERT to generate, that is, the number of "mask" to insert to the bug location. As an early trial, we try this method on fixing bugs in Var-Misuse dataset, in which each bug corresponds to a misused variable. We randomly sample 1000 functions and evaluate the repair accuracy. When we set the repair (sub)token number to 1, 65.9% of bugs are fixed correctly. When up to 3 (sub)tokens are taken into consideration, the number of fixed bugs grows to 81.4%. Although WELL can only repair simple bugs now, this introduces a new thought to achieve bug fixing.

APPENDIX D: MORE VISUALIZED CASES OF WELL

We list more visualized cases from WELL in BoundError in Figure D.1. Perceivable dark backgrounds in each case are sparse, indicating that the importance scores are concentrated. Figure 1A–E are correctly handled, and WELL is quite "certain" about the buggy location predictions, as the importance scores are almost concentrated only upon the actual buggy locations. Although the bug in Figure 1F is erroneously located by WELL, it still pays attention to the actual buggy position, and is in a dilemma between ">" (actual bug) or "> =" (wrong prediction). In some certain scenarios, ">" may be misused; while in others, "> =" may be misused. As the context is not provided in this case, WELL cannot make certain and correct decisions. We assume that it is hard to decide whether the equal condition should be incorporated given the limited context. And it is understandable for WELL to make mistakes in such cases.

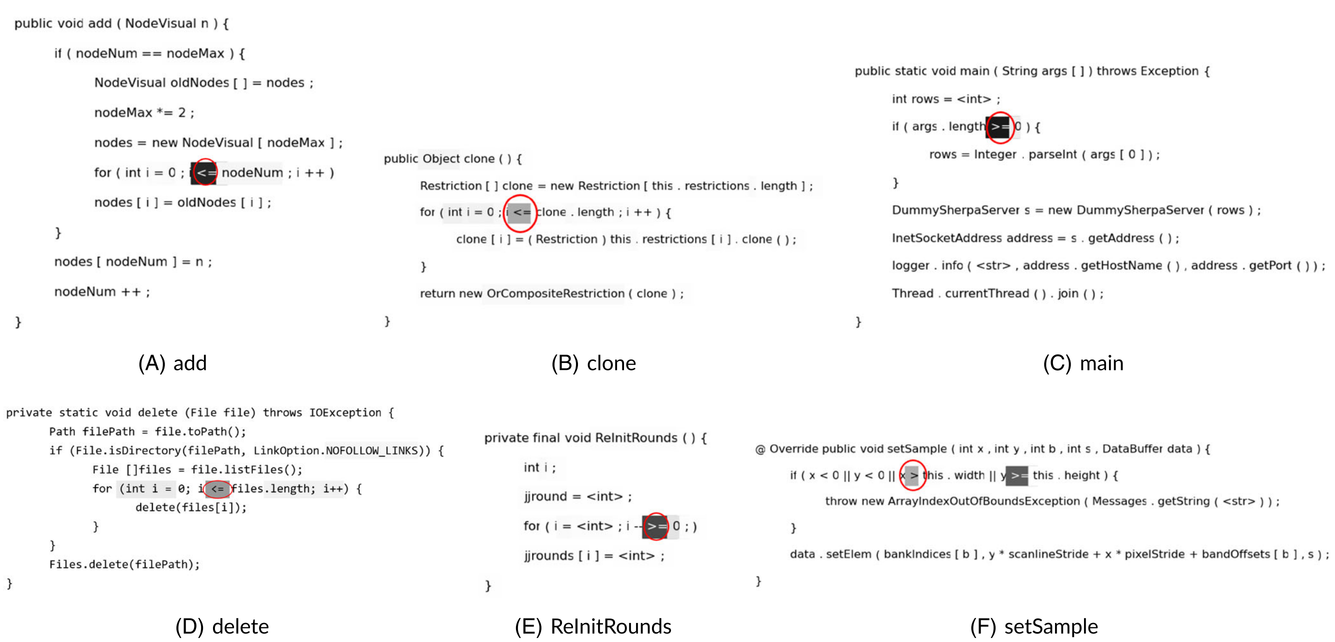


FIGURE D.1 Visualization of the importance score produced by WELL in BoundError. The red circle suggests the buggy location. The gray-scale of the background represents the importance score of the corresponding token. (a)-(e) are correctly handled by WELL, as the darkest token is coincident with the ground-truth circle; while (f) is erroneously handled, as WELL locate the bug at "> =" instead of ">."

20477481, 2024, 9, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/smr.2669 by Peking University Health, Wiley Online Library on [05/01/2025]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

APPENDIX E: MORE VISUALIZED CASES OF WELL-Hi

We present more visualized cases of WELL-H_i on different attention heads in BoundError in Figure D.2. Similar to Figure 6A, we carry out visualization upon three heads (i.e., 2, 7, and 11), where WELL-H_7 is considered as effective and valid for bug localization while WELL-H_2 and H_{11} are not according to Figure 5B. In all cases, WELL-H_7 (WELL-1) produces perceivable and concentrated importance score, and accurately locates the bugs. On the contrary, WELL-H_2 and H_{11} are distracted in many cases, and make less accurate localization. These cases to a certain extent demonstrates the issue of multihead attention, and the necessity and feasibility of extended WELL-1.

```
H2   public static boolean hasFlag ( int options , int flag ) { return ( ( options & flag ) >= 0 ) ; }
H7   public static boolean hasFlag ( int options , int flag ) { return ( ( options & flag ) >= 0 ) ; }
H11  public static boolean hasFlag ( int options , int flag ) { return ( ( options & flag ) >= 0 ) ; }
```

(A) hasFlag

```
H2   private boolean isDigit ( char ch ) { return ch >= '0' && ch < '9' ; }
H7   private boolean isDigit ( char ch ) { return ch >= '0' && ch < '9' ; }
H11  private boolean isDigit ( char ch ) { return ch >= '0' && ch < '9' ; }
```

(B) isDigit

```
H2   private void previousPage ( ) { if ( currentPage >= 0 ) { currentPage – ; updateTitle ( ) ; showPage ( currentPage ) ; } }
H7   private void previousPage ( ) { if ( currentPage >= 0 ) { currentPage – – ; updateTitle ( ) ; showPage ( currentPage ) ; } }
H11  private void previousPage ( ) { if ( currentPage >= 0 ) { currentPage – ; updateTitle ( ) ; showPage ( currentPage ) ; } }
```

(C) previousPage

```
H2   public void run ( ) { for ( int i = 0 ; i <= CNT ; i ++ ) { test ( ) ; } }
H7   public void run ( ) { for ( int i = 0 ; i <= CNT ; i ++ ) { test ( ) ; } }
H11  public void run ( ) { for ( int i = 0 ; i <= CNT ; i ++ ) { test ( ) ; } }
```

(D) run

```
H2   public void setFulfill ( String fulfill ) { _fulfill = fulfill != null && fulfill . length ( ) >= 0 ? fulfill : null ; }
H7   public void setFulfill ( String fulfill ) { _fulfill = fulfill != null && fulfill . length ( ) >= 0 ? fulfill : null ; }
H11  public void setFulfill ( String fulfill ) { _fulfill = fulfill != null && fulfill . length ( ) >= 0 ? fulfill : null ; }
```

(E) setFulfill

```
H2   public void setMaxWriteThroughput ( int maxWriteThroughput ) { if ( maxWriteThroughput <= 0 ) {
     maxWriteThroughput = 0 ; } this . maxWriteThroughput = maxWriteThroughput ; }
H7   public void setMaxWriteThroughput ( int maxWriteThroughput ) { if ( maxWriteThroughput <= 0 ) {
     maxWriteThroughput = 0 ; } this . maxWriteThroughput = maxWriteThroughput ; }
H11  public void setMaxWriteThroughput ( int maxWriteThroughput ) { if ( maxWriteThroughput <= 0 ) {
     maxWriteThroughput = 0 ; } this . maxWriteThroughput = maxWriteThroughput ; }
```

(F) setMaxWriteThroughout

FIGURE D.2 Visualization of the importance score produced by WELL-H_2, H_7 and H_{11}. The gray-scale of the background suggests the importance score of the corresponding token, and the red box refers to the buggy location. According to Figure 5B, WELL-H_7 is valid and effective and the other two are not.

20477481, 2024, 9, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/smr.2669 by Peking University Health, Wiley Online Library on [05/01/2025]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

Journal of Software: Evolution and Process — WILEY

APPENDIX F: MORE VISUALIZED CASES OF WELL-lstm

We provide more visualized cases of WELL-lstm in BoundError in Figure D.3. WELL-lstm exhibits similar behaviors in most cases, where the importance scores are concentrated on the buggy locations. However, there are also erroneous predictions, such as case 2. WELL-lstm regards "segmentEnd" instead of "< =" next to it as bug. Such incidents would somehow explain why WELL-lstm is not as effective as WELL nor WELL-1 – the LSTM backbone may be not strong enough to fully support the weakly supervised bug localization.

```
private void assertTagOpened ( String output ) { assertTrue ( output . indexOf ( "<input " ) >= − 1 ) ; }

public boolean contained ( long from , long to ) { return ( from < this . segmentStart && this . segmentEnd <= to ) ; }

public static boolean flagBit ( int b , int bit ) { return ( b & ( 1 << bit ) ) >= 0 ; }

@ Override public boolean hasChildren ( ) { return children . size ( ) >= 0 ; }

private boolean isAcceptableSize ( int width , int height ) { return ( width > _minAcceptableImageWidth ) && ( height
>= _minAcceptableImageHeight ) ; }
```

FIGURE D.3 Visualization of importance scores from WELL-lstm in BoundError. The gray-scale of the background suggests the importance score of the corresponding token, and the red box refers to the buggy location.

20477481, 2024, 9, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/smr.2669 by Peking University Health, Wiley Online Library on [05/01/2025]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License