Code Defect Detection using Pre-trained Language Models with Encoder-Decoder via Line-Level Defect Localization

Jimin An[1*], **YunSeok Choi**[2*], **Jee-Hyong Lee**[1†]
[1]Sungkyunkwan University, Suwon, South Korea
[2]Hankuk University of Foreign Studies, Seoul, South Korea
als398@skku.edu, ys.choi@hufs.ac.kr, john@skku.edu

Abstract

Recently, code Pre-trained Language Models (PLMs) trained on large amounts of code and comment, have shown great success in code defect detection tasks. However, most PLMs simply treated the code as a single sequence and only used the encoder of PLMs to determine if there exist defects in the entire code. For a more analyzable and explainable approach, it is crucial to identify which lines contain defects. In this paper, we propose a novel method for code defect detection that integrates line-level defect localization into a unified training process. To identify code defects at the line-level, we convert the code into a sequence separated by lines using a special token. Then, to utilize the characteristic that both the encoder and decoder of PLMs process information differently, we leverage both the encoder and decoder for line-level defect localization. By learning code defect detection and line-level defect localization tasks in a unified manner, our proposed method promotes knowledge sharing between the two tasks. We demonstrate that our proposed method significantly improves performance on four benchmark datasets for code defect detection. Additionally, we show that our method can be easily integrated with ChatGPT.

Keywords: code defect detection, line-level defect localization, unified multi-task training

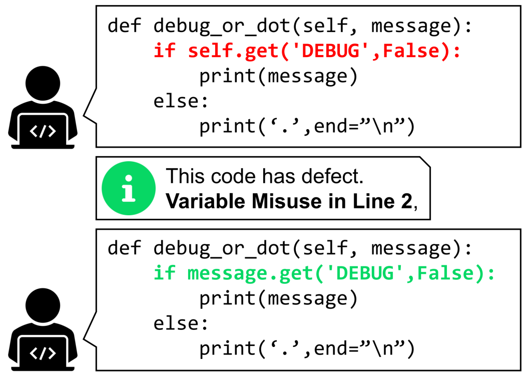
1. Introduction

Code defect detection is the process of identifying errors, bugs, or potential issues in software code. These defects can lead to functional errors in the software, and result in software threats and vulnerabilities. The purpose of code defect detection is to discover these flaws in advance and correct them, thereby enhancing the quality of the software.

Pre-trained Language Models (PLMs) for programming language have achieved significant success in code defect detection tasks (Feng et al., 2020a; Ahmad et al., 2021; Wang et al., 2021a; Guo et al., 2022). These models learned the context and patterns of programming language from large datasets of source code in pre-training stage. Then, the encoder of PLMs is fine-tuned on labeled datasets where code are tagged as either containing defects or being defect-free.

However, most PLMs for code defect detection tasks focused solely on classifying whether code has defects or not. They just treated the entire code as a single sequence input and predicted the defects based on the overall context of the code. It fails to provide detailed information about where exactly in the code the bugs or defects exist, and what their causes might be. Simply classifying whether the code has defects makes developers manually find out which line in the code is vulnerable.

For code defect detection, it is important to know where the defect is in the code. As shown in Figure 1, it helps developers to identify precise vulnerable sections within the source code, facilitating rapid resolution of security flaws. If a specific line is predicted to be vulnerable in a software project, it can prompt code reviews or modification efforts targeted at that particular line. Line-level code defect predictions enhance overall software reliability.



```
def debug_or_dot(self, message):
    if self.get('DEBUG',False):
        print(message)
    else:
        print('.',end="\n")
```

This code has defect.
Variable Misuse in Line 2,

```
def debug_or_dot(self, message):
    if message.get('DEBUG',False):
        print(message)
    else:
        print('.',end="\n")
```

Figure 1: An example of code defect detection. Finding vulnerable lines in the code helps developers quickly fix defective code.

For a more analyzable and explainable approach to code defect detection, it is crucial to identify which lines contain defects. However, most PLMs for programming language have simply used a whole code sequence as input during the pre-training stage. When given code and comments as input, they are concatenated with special tokens. [CLS] token is added at the beginning, [SEP] token is placed between the code and comment, and

[*]Equal contribution
[†]Corresponding author

[END] token is placed at the end of the sequence. Finally, during the fine-tuning stage, the input sequence is concatenated with [CLS] token at the beginning of the code and is fed to the classifier. The entire code sequence as a whole input makes it challenging to focus on line-level details, which are crucial for tasks like code defect detection.

To address these aspects of code defect detection, we treat the code line-by-line and detect defective lines by classifying whether each line of the code exists as a defect. In this paper, we propose a novel method for code defect detection that integrates line-level defect localization into a unified training process. We utilize code PLMs with an encoder-decoder architecture, which is capable of understanding the context of programming languages. First, we insert [LINE] token at the end of each line to process the code as sequences based on each line. When the input is given to the PLMs, the encoder of PLMs learns for two tasks. The first task is a code defect detection task that identifies code defects in the code based on the [CLS] token. The second task is a line-level defect localization task that learns to identify which lines contain defects based on all [LINE] tokens. Furthermore, to utilize the characteristic that both the encoder and decoder in PLMs process information differently, we leverage both the encoder and decoder of the PLMs for line-level defect localization. The decoder of PLMs is trained to generate lines with defects. By employing a unified training method for the three processes, our approach promotes knowledge sharing between code defect detection and line-level defect localization. We demonstrate that our proposed method significantly improves defect detection performance on four benchmark datasets.

We summarize the contributions of this paper in three aspects: (1) We introduce a method to process the code as sequences based on each line for line-level defect localization. (2) We utilize code PLMs with an encoder-decoder architecture into a unified training process for code defect detection task. (3) We demonstrate that our proposed method can be easily integrated with ChatGPT.

2. Related Work

Previous research for code defect detection can be categorized into sequence-based approaches, which extract vulnerability patterns from code sequences, and graph-based approaches, which utilize the hierarchical structure of code. For sequence-based approaches, Li et al. (2018) used the Bidirectional LSTM (Hochreiter and Schmidhuber, 1997) method and learned vulnerability patterns extracted from code through APIs and libraries to detect defects. Li et al. (2022) extracted sequential patterns using the flattened Abstract Syntax Tree (AST) of the code using Bi-GRU model (Cho et al., 2014). For a graph-based approach, Zhou et al. (2019) encoded code into a graph structure using Graph Neural Networks. However, these approaches were only trained to predict code defects using limited datasets, and they have limitations in performance.

As the pre-trained models based on the Transformer architecture (Vaswani et al., 2017) have achieved great success in natural language understanding tasks (Devlin et al., 2019; Radford et al., 2019; Liu et al., 2019; Lewis et al., 2020; Clark et al., 2020; Raffel et al., 2020), the methods for extending natural language-based methods to code have recently been proposed in code understanding tasks (Kanade et al., 2020; Feng et al., 2020a; Guo et al., 2020; Ahmad et al., 2021; Wang et al., 2021a; Guo et al., 2022; Wang et al., 2023). Kanade et al. (2020) proposed a method that was pre-trained on a massive amount of Python code to obtain contextual embeddings of source code. They also introduced three benchmark datasets such as Variable-Misuse Classification, Wrong Binary Operator, Swapped Operand for defect detection tasks. Feng et al. (2020a) proposed CodeBERT, a pre-trained language model, based on BERT (Devlin et al., 2019), to learn cross-modal representation of both program language and natural language. Ahmad et al. (2021) proposed PLBART to support both code generation tasks using encoder-decoder model BART (Lewis et al., 2020). Wang et al. (2021a) proposed CodeT5, a pre-trained encoder-decoder model based on T5 (Raffel et al., 2020), to facilitate generation tasks for programming language, and recently proposed an expended CodeT5 model, CodeT5+ (Wang et al., 2023). However, they only predict whether there are defects in the entire code, and they do not predict which specific lines contain defects.

Recently, Large Language Models (LLMs) have been proposed with massive model sizes and extensive training data. ChatGPT (Brown et al., 2020; OpenAI, 2024) has the capability to understand various topics and contexts, thereby demonstrating high performance in diverse generation tasks. However, its performance in classification tasks such as code defect detection is not as high as that of traditional PLMs designed specifically for programming languages.

3. Method

In this section, we introduce a unified framework that utilizes both the encoder and decoder of PLMs for code defect detection. Figure 2 shows the architecture of our proposed method. Our proposed method aims to train two tasks: code defect de-

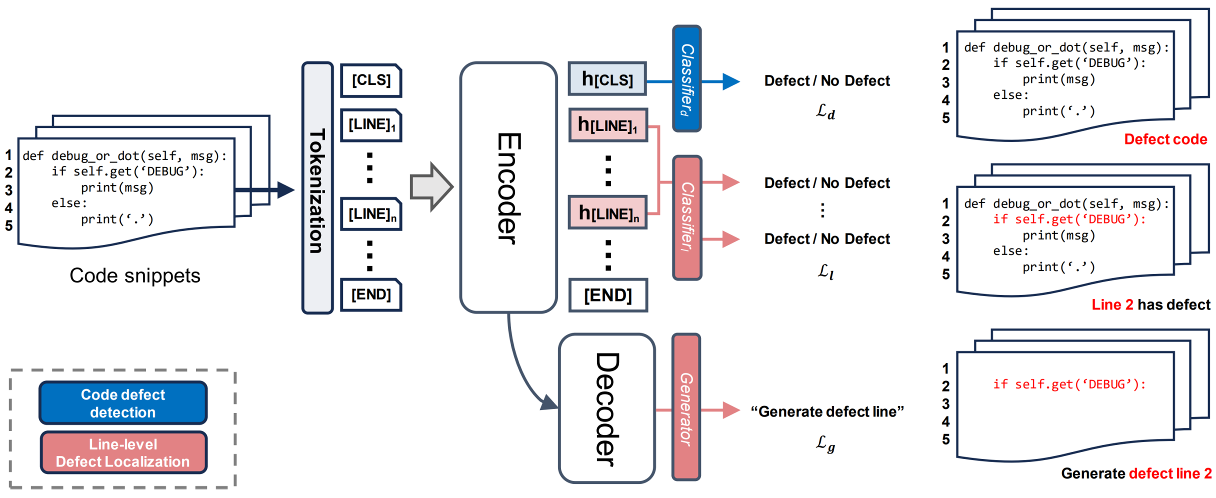


Figure 2: The framework of our proposed method. Our proposed method utilizes both the encoder and decoder of PLMs to simultaneously train code defect detection and line-level localization tasks in a unified manner.

tection and line-level defect localization. First, we employ the encoder of PLMs with the primary goal of classifying code defect detection (Section 3.1). Second, we focus on identifying the specific locations of defects within the code, classifying defective lines on a line-by-line basis (Section 3.2). For utilizing both the encoder and decoder of PLMs which process information differently, we additionally conduct the task of generating defective lines in the decoder for line-level localization (Section 3.3). Our goal is to enhance code defect detection by simultaneously training on these three processes within a unified framework (Section 3.4).

We provide a detailed explanation of our training processing in the following subsection.

3.1. Encoder: Code Defect Detection

In the pre-training stage, PLMs incorporate the [CLS] token at the beginning of each sequence to facilitate the learning of sequence embeddings. This [CLS] token serves as a crucial anchor, capturing the context of the code sequence and representing it as an embedding. As the embedding of this token represents context information of the entire input, it is used during the fine-tuning stage. In code defect detection task, the embedding of the [CLS] token enables PLMs to classify whether the code contains defects.

Given code C, we aim to train the encoder of PLMs that identify defects within code. In this phase, we obtain the embedding value h_i of C. The embedding of the [CLS] token is then fed into the first classifier for code defect detection as follows:

$$I = \langle \text{[CLS]}; \text{C}; \text{[END]} \rangle \tag{1}$$

$$h = \text{Encoder}(I) \tag{2}$$

$$logit_d = \text{Classifier}_d(h_{\text{[CLS]}}) \tag{3}$$

where [CLS] and [END] tokens are special tokens representing the classification token and the end of sequence respectively, $h_{\text{[CLS]}}$ refers to the embedding of the [CLS] token, and $logit_d$ is the output from the classifier for code defect detection.

Finally, the loss L_d is computed as follows:

$$\mathcal{L}_d = \text{CrossEntropy}(logit_d, label_d) \tag{4}$$

where $label_d$ is the ground truth label whether the code contains a defect or not.

3.2. Encoder: Line-level Defect Localization

Previous works for code defect detection using PLMs only learn from the embedding of the [CLS] token corresponding to code C. They simply regarded the entire code sequence as a whole input. These approaches summarize the code with a single [CLS] token. However, most code defects derive from specific parts of a line, such as variables or operators. Thus, training solely based on a single [CLS] token may not effectively capture detailed information about the precise locations of defects or bugs in the code. To capture more context information about the code, we introduce a line-level defect localization method that allows the model to detect defects on a line-by-line basis, thereby enabling it to learn the exact location information related to defects and bugs.

Given code C $= \{c_1, c_2, ..., c_n\}$ consisting of n lines, which c_i means the i-th line of the code, we

Datasets	Train	Valid	Test	Lines (defective)
Devign	17.8K (9.7K/8.2K)	2.2K (1.3K/0.9K)	2.3K (1.3K/1.0K)	49.6 (6.2)
Variable-Misuse	700.6K (350.3K/350.3K)	8.2K (4.1K/4.1K)	378.4K (189.2K/189.2K)	10.2 (1.0)
Wrong Binary Operator	459.4K (229.7K/229.7K)	8.2K (4.1K/4.1K)	251.8K (125.9K/125.9K)	18.4 (1.0)
Swapped Operand	236.2K (118.1K/118.1K)	8.2K (4.1K/4.1K)	131.0K (65.5K/65.5K)	22.5 (1.0)

Table 1: Statistic of the four benchmark datasets. (/) represents the number of datasets with defects and without defects, respectively. (defective) means the average number of defective lines in a code.

aim to train a model that detects defects on a line-by-line basis. We insert line tokens to delineate each line for a code C as shown in Figure 2. This effectively segments the code at every newline. Then, using the encoder of PLMs, we obtain embeddings for each individual line. The embeddings of the n line tokens are fed into a classifier for line-level defect localization. The process of obtaining $logit_i$ for the i-th line c_i is as follow:

$$I = \langle \text{[CLS]}; \text{[LINE]}_1; c_1; ..., \text{[LINE]}_n; c_n; \text{[END]} \rangle \quad (5)$$

$$h = \text{Encoder}(I) \quad (6)$$

$$logit_{l_i} = \text{Classifier}_l(h_{\text{[LINE]}_i}) \quad (7)$$

where [LINE]_i is a special token representing the start of code i-th line, $h_{\text{[LINE]}_i}$ refers to the final embedding of the [LINE]_i token for the encoder of PLMs, and $logit_{l_i}$ is the output of i-th line from the classifier for line-level defect localization.

The loss for localizing defects in each line of the code is as follows:

$$\mathcal{L}_l = \sum_{i=1}^{n} \text{CrossEntropy}(logit_{l_i}, label_{l_i}) \quad (8)$$

where $label_{l_i}$ is the ground truth label whether the i-th line of code is defective or not.

As shown in Figure 2, we simultaneously conduct code defect detection and line-level defect localization using the encoder of PLMs. The whole code sequence C is trained through the classifier for code defect, and the classifier for line-level defect is trained to determine the absence of defects on a line-by-line basis. By integrating to train on two tasks, a single PLM encoder shares the context information of the code.

3.3. Decoder: Line-level Defect Localization

For classification tasks, classifiers are commonly trained using only the encoder of PLMs. The encoder is well-suited to understanding the context of the input sequence for classification. To utilize the generation capability of the decoder as well as the encoder, we aim to train both the encoder and decoder of PLMs for line-level defect localization.

If the encoder detects whether each line has a defect, the decoder is designed to generate which line contains the defect. Both the encoder and decoder aim to find the defective line, but they differently process information through each process. The code embedding h obtained from the encoder is fed to the decoder. The decoder generates a defective line gen_i. The generated line and defective line are calculated to compute the loss as follows:

$$gen_i = \text{Decoder}(h) \quad (9)$$

$$\mathcal{L}_g = \text{GenerationLoss}(gen_i, sent_i) \quad (10)$$

where $sent_i$ is the ground truth defective line in the code.

3.4. Unified Multi-task Training

Instead of training each task separately, we aim to learn across three processes in a unified manner, leading to a better transfer of knowledge across tasks. We obtain the loss in the encoder for classifying the entire code's defect, the loss for classifying each line as a defect or not, and the loss in the decoder for generating defective lines in the code. Then we compute the final loss by summing the three losses, each multiplied by a weight factor. The final loss for our proposed method is presented as follows:

$$\mathcal{L}_{final} = w_1 * \mathcal{L}_d + w_2 * \mathcal{L}_l + w_3 * \mathcal{L}_g \quad (11)$$

where w_1, w_2, and w_3 are the weights of each loss.

4. Experiment Setup

Dataset We conduct experiments on four public benchmark datasets for the defect detection task. Zhou et al. (2019) introduced the Devign dataset, which is one of the benchmark datasets in CodeXGLUE (Lu et al., 2021). Kanade et al. (2020) presented three benchmark dataset, Variable-Miuse, Wrong Binary Operator, and Swapped Operand. Table 1 shows the detailed statistics of the datasets.

- **Devign** consists of functions from large open-source C projects. Its objective is to predict whether the code is vulnerable to software systems or not. We removed datasets from which we cannot obtain line-level defect information.

Datasets	Devign		VM		WBO		SO	
Models	Acc.	F1.	Acc.	F1.	Acc.	F1.	Acc.	F1.
CuBERT (Kanade et al., 2020)	-	-	94.04	-	89.90	-	92.20	-
CodeBERT (Feng et al., 2020a)	63.73	51.51	93.21	93.03	90.66	90.27	91.06	90.77
CodeT5 (Wang et al., 2021a)	62.87	58.39	93.82	93.74	88.12	87.75	91.78	91.70
CodeT5+ (Wang et al., 2023)	63.40	62.59	93.28	93.21	89.08	88.62	92.70	92.61
UniXcoder (Ahmad et al., 2021)	63.18	47.57	93.95	93.85	90.35	90.11	93.73	93.66
CodeT5 (ours)	**65.44**	62.68	**95.43**	**95.38**	90.53	90.29	93.88	93.80
CodeT5+ (ours)	64.91	**63.97**	95.08	95.02	91.56	91.35	93.69	93.62
UniXcoder (ours)	64.29	58.37	95.36	95.30	**92.49**	**92.31**	**94.22**	**94.16**

Table 2: Comparison of our proposed method with the baseline models on the four benchmark dataset. We selected CodeT5, CodeT5+, and UniXcoder, which are the state-of-the-art (SOTA) PLMs with the encoder-decoder architecture, as our baselines. The best result is in **boldface**, and the next best is underlined.

- **Variable-Misuse Classification (VM)** is a dataset to determine whether two variables in a code are mistakenly swapped.
- **Wrong Binary Operator (WBO)** is a dataset to check if a binary operator is erroneously replaced with another operator.
- **Swapped Operand (SO)** is a dataset to verify whether the operands of a binary operator are incorrectly swapped.

Evaluation Metrics For code defect detection, we use Accuracy, F1-score, AUC-ROC curve (Bradley, 1997), and PR-AUC (Davis and Goadrich, 2006). For line-level defect selection, we use Accuracy and F1-score.

- **Accuracy** is the ratio of correctly predicted instances to the total number of instances in the dataset.
- **F1-score** is the harmonic mean of precision and recall.
- **AUC-ROC** is that the ROC curve plots the true positive rate against the false positive rate for various threshold values, and AUC gives the area under the ROC curve.
- **PR-AUC** is the precision-recall curve plots precision against recall for various thresholds.

Baselines We compare our proposed approach with PLMs for programming language, such as CuBERT (Kanade et al., 2020), CodeBERT (Feng et al., 2020a), CodeT5 (Wang et al., 2021b), CodeT5+ (Wang et al., 2023), and UniXCoder (Guo et al., 2022). They are fine-tuned using only the [CLS] token of each code, as a general training method for code defect detection. We refer to the CuBERT reported by Kanade et al. (2020), which is a PLM specifically for Python language.

Implementation Details We selected various code PLMs, CodeT5, CodeT5+, and UniXcoder, which are encoder-decoder architectures, as our framework. Our framework has three main tasks: (1) Code Defect Detection in Encoder, (2) Line-level Defect Localization in Encoder, and (3) Line-level Defect Localization in Decoder. The input is the code C $= [c_1, ..., c_n]$ with [CLS] token and [LINE] tokens to delineate each code line c_k. While other tokens or different special tokens can be considered, we chose the [SEP] token that preserved the overall meaning of the code in C and Python Language. We set the batch size to 8 and the learning rate is 2e-5, the maximum source length to 512 tokens. VM/WBO/SO and Devign fine-tuning for 2 and 10 epochs, respectively. Also, we set the maximum length of the [LINE] token to 50, 15, 20 and 30 for Devign, VM, WBO and SO, which are similar to the average line length of the code in each dataset.

The loss weight values and ground truth label are as follows:

(1) Fine-tuning with [CLS] token and target is no-defect(0) or defect(1). We set the loss weight w_1 for $\mathcal{L}_d$ to 1.

(2) Fine-tuning with [LINE] tokens and target is no-defect(0) or defect(1) with each line. We set the loss weight w_2 for $\mathcal{L}_l$ to 0.1 and 0.5 for Devign and VM/WBO/SO datasets, respectively.

(3) If code C has a defect in k-th line c_k, the decoder generate the the k-th line. Otherwise, the decoder generates the "No Defect Found". We set the target sequence length to 50 tokens and loss weight w_3 for $\mathcal{L}_g$ to 0.1.

5. Experiment Result

5.1. Main Result

Table 2 shows the comparison of our proposed method with baselines on four benchmark datasets

	Devign		VM		WBO		SO	
	Acc.	F1.	Acc.	F1.	Acc.	F1.	Acc.	F1.
Baseline: (1)	62.87	58.39	93.82	93.74	88.12	87.75	91.78	91.70
Ours: (1)+(2)	63.18	58.88	94.76	94.70	89.13	88.84	92.35	92.28
Ours: (1)+(3)	63.67	56.01	94.12	94.06	89.97	89.68	92.80	92.72
Ours: (1)+(2)+(3)	**65.44**	**62.68**	**95.43**	**95.38**	**90.53**	**90.29**	**93.88**	**93.80**

Table 3: Ablation study on three learning processes of our proposed method, (1) Code Defect Detection in Encoder (2) Line-level Defect Localization in Encoder, and (3) Line-level Defect Localization in Decoder. The best result is in **boldface**, and the next best is underlined. We chose CodeT5 as the baseline.

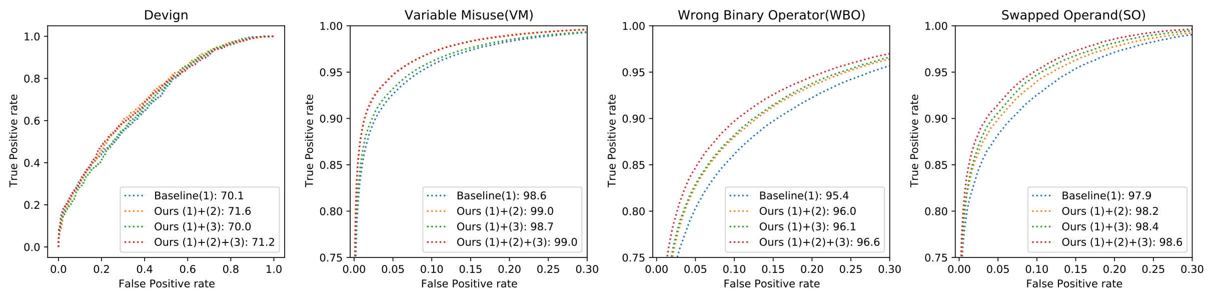


Figure 3: Comparison results of AUC-ROC curve on the four benchmark datasets. The X-axis and Y-axis represent the False Positive Rate and True Positive Rate, respectively. A higher AUC score indicates better classification performance to distinguish between defective and non-defective classes.

for the code defect detection tasks. First, the Devign dataset contains defects that may be used to attack software systems, such as resource leaks and use-after-free, so the task is very difficult compared to other datasets. Among the baselines, the CodeT5+ showed higher scores with an Accuracy and F1-score of 63.40 and 62.59. When we applied our proposed method with CodeT5+, the score increased by 2.38% and 2.20%, respectively. Also, while UniXcoder demonstrated high Accuracy, its F1-score was significantly lower. This indicates an increased number of False Negative predictions, leading to a decreased Recall and hence lower F1-score. It implies that the model often misclassifies actual defective code as non-defective. By applying our method, the increase in F1-score is more significant in all baselines. Finally, CodeT5 with our proposed method shows the best performance in the Devign dataset.

For the datasets of Variable-Misuse (VM), Wrong Binary Operator (WBO), and Swapped Operand (SO), CuBERT and UniXcoder show higher performance than other baselines. CuBERT is trained on only Python programming language, so it shows better performance in the datasets, which consist of the code in Python datasets. In UniXcoder, code representation is obtained based on the Abstract Syntax Tree, enabling it to better capture structural information. Also, CodeBERT, which is only composed of a PLM encoder, showed higher performance on the WBO dataset compared to other baselines. However, all baselines tended to show a lower F1-score compared to Accuracy. By applying our proposed method to these baselines, both Accuracy and F1-score improved and resulted in more consistent defect detection. Finally, CodeT5 with our proposed method shows the best performance in VM dataset, and UniXcoder with our proposed method achieves state-of-the-art in WBO and SO datasets.

5.2. Ablation Study

In Table 3, we present an ablation study to investigate the impact of line-level defect localization learning in the encoder and decoder respectively on the code defect detection performance. We chose CodeT5 as our baseline. Baseline (1) represents the performance of the fine-tuning method using only the [CLS] token. Ours (1)+(2) indicates the results when only the encoder of PLMs is used for both code defect detection and line-level defect localization. Ours (1)+(3) represents the results when the encoder of PLMs is only trained for code defect detection and the decoder of PLMs is trained for line-level defect localization. Lastly, we present the results of our final proposed method (1)+(2)+(3) which combines all three training processes.

First, for ours (1)+(2), we observed an increase in both Accuracy and F1-score compared to the baseline. We treated the code as line-by-line based on the [LINE] token rather than just a single sequence. This demonstrates that by training the encoder to detect code defects and simultaneously identify the

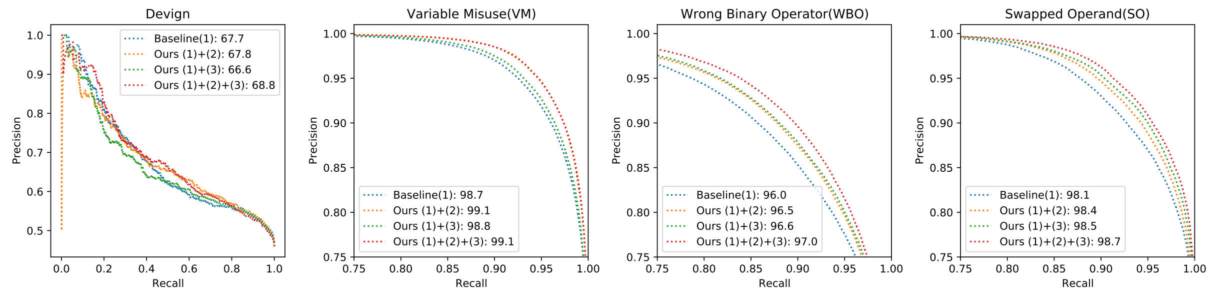


Figure 4: Comparison results of PR-AUC curve on the four benchmark datasets. The X-axis and Y-axis represent the Recall and Precision, respectively. A higher PR-AUC score indicates better classification performance to correctly classify defective code.

Models	CodeT5		CodeT5+		UniXcoder	
Dataset	Acc.	F1.	Acc.	F1.	Acc.	F1.
Devign	78.8	11.2	78.2	7.8	80.4	13.8
VM	99.0	93.5	98.7	91.3	98.8	92.2
WBO	98.5	87.2	98.4	85.2	98.9	90.4
SO	99.4	94.5	99.2	93.1	99.4	94.9

Table 4: Performance of the encoder of PLMs for Line-level Defect Localization on the four benchmark datasets.

Models	ChatGPT		ChatGPT+Ans		Ours	
Dataset	Acc.	F1.	Acc.	F1.	Acc.	F1.
Devign	51.6	52.7	81.0	82.6	65.3	62.5
VM	56.3	55.5	87.0	88.0	95.0	94.8
WBO	54.8	49.4	83.7	85.0	90.4	89.7
SO	55.2	49.2	83.1	84.6	93.6	93.1

Table 5: Comparison of our proposed method with ChatGPT to detect defect code on four datasets sampled with 1000 instances.

Models	ChatGPT		ChatGPT+Ours	
Dataset	BLEU	CodeBLEU	BLEU	CodeBLEU
Devign	63.2	65.1	65.3	67.6
VM	60.5	64.3	62.6	65.8
WBO	74.8	77.9	78.7	80.1
SO	72.1	75.2	77.1	80.2

Table 6: Comparison of our proposed method with ChatGPT to fix repair code on defective code in terms of BLEU (Papineni et al., 2002) and CodeBLEU (Ren et al., 2020) score.

specific lines with defects, the model can better pinpoint defective code.

For ours (1)+(3), while there was an overall performance improvement compared to the baseline, we noticed a significant increase in Accuracy but a notably lower F1-score in the Devign dataset. This is because the model does not focus on line-level but instead tries to locate the defect sequence based on the entire code, leading to a higher number of False Negative predictions compared to (1)+(2).

Lastly, for our final proposed method (1)+(2)+(3), we observed a significant increase in both Accuracy and F1-score across all datasets. By utilizing code PLMs with an encoder-decoder architecture that understands the context of programming languages, we trained on code defect detection and line-level defect localization tasks using an integrated learning method. This unified approach promotes knowledge sharing between the encoder and decoder.

Figure 3 shows the AUC-ROC curve results for four benchmark datasets. For all datasets, our method achieved the highest AUC score. Especially, for our final approach (1)+(2)+(3), The results increase by 3.00%, 0.41%, 1.26%, and 0.72% compared to the baseline (1), respectively. We demonstrated that our method effectively classified whether the code has defects and consistently performed well across all thresholds.

In Figure 4, we present the PR-AUC curve for four benchmark datasets. For all datasets, our proposed method achieved the highest PR-AUC score. A good PR-AUC curve performance indicates that the model is effectively predicting the defective class and the model is correctly predicting a large number of defective samples while minimizing the number of false positives. We showed that our proposed method has the excellent predictive capability of the model.

5.3. Line-level Defect Localization

We analyze how well the encoder of PLMs performed on line-level defect localization, and evaluate based on Accuracy and F1-score for lines that contain defects. Table 4 presents the performance of the encoder for Line-level Defect Localization. In the Devign dataset, which predicts vulnerabilities in a software system rather than just syntactical errors, the result shows lower performance than other datasets. Additionally, the dataset contains

ChatGPT	ChatGPT+Ours
Prompt: Repair the following defect code snippet. In your response, output the fixed code only. (Code)	**Prompt**: Repair the following defect code snippet. In your response, output the fixed code only. Defect candidate Line: n-th line in this code. (Code)

Code 1:
def update_thumbs(self, nameList):
    invalid = (set(self.thumbList) - set(nameList))
    if len(nameList) > 0:
        with self.thmblock:
            for thumbkey in invalid
                self.thumbList.remove(thumbkey)
                del self.thumbDict[thumbkey]
                self._tkf_highlight.discard(thumbkey)
        self.reorder_thumbs()
	Code 1:
	def update_thumbs(self, nameList):
	    invalid = (set(self.thumbList) - set(nameList))
	    if len(invalid) > 0:
	        with self.thmblock:
	            for thumbkey in invalid
	                self.thumbList.remove(thumbkey)
	                del self.thumbDict[thumbkey]
	                self._tkf_highlight.discard(thumbkey)
	        self.reorder_thumbs()

Code 2:
def _guess_media_encoding(self, source):
    info = source.byteStream.info()
    if 'Content-Type' in info:
        for param in self.getplist():
            if param.startswith('charset='):
                return param.split('=', 1)[1].lower()
	Code 2:
	def _guess_media_encoding(self, source):
	    info = source.byteStream.info()
	    if 'Content-Type' in info:
	        for param in info.getplist():
	            if param.startswith('charset='):
	                return param.split('=', 1)[1].lower()

Table 7: A qualitative example of the VM dataset on repair task.

multiple defective lines within a single code, making accurate prediction challenging. However, by training to identify which lines contain defects simultaneously, we can observe that the model better represents code for the code defect detection task. Furthermore, for the VM, WBO, and SO datasets, the model is notably effective at predicting lines with defects. It helps developers to identify precisely vulnerable sections within the source code.

5.4. Comparison with ChatGPT

To evaluate how well the Large Language Model, ChatGPT, performs in code defect detection, we sample 1000 instances from each dataset, and compare the defect detection performances of ChatGPT and our proposed method as shown in Table 5. When we simply ask ChatGPT to detect the defect code, ChatGPT shows a very low performance, around 50% for each dataset. In order to maximize the performance of ChatGPT, we perform additional experiments by providing defect line information for each code in prompts. The ChatGPT+Ans in Table 5 show the results with the ground truth defect information.

We notify that our proposed method shows significantly higher performance even than ChatGPT with golden defect line information. Our proposed method performs better on the VM, WBO, and SO datasets. Interestingly, on the Devign dataset, ChatGPT with the ground truth of defect line information performs the best. This may arise from the small size of the Devign dataset. As shown in Table 1, the size of Devign is approximately one over several tens of others. So, a model may learn relatively small amount of information from the dataset. Since ChatGPT was trained on a large corpus, it can perform better on the small size of datasets using its knowledges from the pre-training dataset. This suggests that we may benefits from the broad knowledge of a pre-trained language model even for code intelligence tasks.

5.5. Integration with ChatGPT

Since ChatGPT was designed as a general-purpose generative AI, it showed limitations in classification tasks like code defect detection. However, we can combine our proposed method to ChatGPT to enhance its generative capabilities. We conduct another experiments to generate repaired code by combining ChatGPT with our model. We sample 100 defect code instances from each dataset, and we additionally provide the predicted line-level defect information by our method in prompts. The repair performance is shown in Table 6.

When we simply ask ChatGPT to repair the defective code (ChatGPT in Table 6), the BLEU scores are 63.2%, 60.5%, 74.8%, and 72.1%, respectively, on the four benchmark datasets. However, when we provide ChatGPT with the predicted defect locations by our method (ChatGPT+Ours), the scores significantly improve to 65.3%, 62.6%, 78.7%, and 77.1%. This demonstrated that we can enhance the generative capabilities of large language models using our proposed method.

Table 7 shows qualitative examples of the repair task with ChatGPT. We provide the task description in the prompt for ChatGPT as shown in the left column, and provide the task description with the predicted defect locations for ChatGPT+Ours. When we simply instruct ChatGPT to repair the defective code, ChatGPT struggle to identify the defective line, which lead to unsuccessful repair attempts. However, when we provide information about the predicted defective line, ChatGPT successfully repairs the defective code.

6. Conclusion

We introduced a novel method for code defect detection with line-level defect localization in a unified manner. By segmenting the code based on lines and leveraging both the encoder and decoder of PLMs, we achieved a more detailed and interpretable defect detection mechanism. We demonstrated that our evaluations on four benchmark datasets showed the superiority of our method in code defect detection. Moreover, the interaction of our approach with generative AI, specifically ChatGPT, broadens its applicability and potential in real-world scenarios. In future work, we plan to conduct research on an integrated model that simultaneously performs code defect detection and defect repair based on line-level defect information.

7. Acknowledgments

This work was partly supported by Institute of Information & communications Technology Planning & Evaluation (IITP) grant funded by the Korea government (MSIT) (No.2019-0-00421, AI Graduate School Support Program, Sungkyunkwan University, and No.2022-0-01045, Self-directed Multimodal Intelligence for solving unknown, open domain problems)

8. Bibliographical References

Wasi Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2021. Unified pre-training for program understanding and generation. In *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 2655–2668, Online. Association for Computational Linguistics.

Berkay Berabi, Jingxuan He, Veselin Raychev, and Martin Vechev. 2021. Tfix: Learning to fix coding errors with a text-to-text transformer. In *International Conference on Machine Learning*, pages 780–791. PMLR.

Andrew P. Bradley. 1997. The use of the area under the roc curve in the evaluation of machine learning algorithms. *Pattern Recognition*, 30(7):1145–1159.

Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. 2020. Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. 2021. Evaluating large language models trained on code.

KyungHyun Cho, Bart van Merrienboer, Dzmitry Bahdanau, and Yoshua Bengio. 2014. On the properties of neural machine translation: Encoder-decoder approaches. *CoRR*, abs/1409.1259.

Kevin Clark, Minh-Thang Luong, Quoc V Le, and Christopher D Manning. 2020. Electra: Pre-training text encoders as discriminators rather than generators. *arXiv preprint arXiv:2003.10555*.

Jesse Davis and Mark Goadrich. 2006. The relationship between precision-recall and roc curves. In *Proceedings of the 23rd international conference on Machine learning*, pages 233–240.

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. 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, Volume 1 (Long and Short Papers)*, pages 4171–4186, Minneapolis, Minnesota. Association for Computational Linguistics.

Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. 2020a. CodeBERT: A pre-trained model for programming and natural languages. In *Findings of the Association for Computational Linguistics: EMNLP 2020*, pages 1536–1547, Online. Association for Computational Linguistics.

Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. 2020b. Codebert: A pre-trained model for programming and natural languages. *arXiv preprint arXiv:2002.08155*.

Daya Guo, Shuai Lu, Nan Duan, Yanlin Wang, Ming Zhou, and Jian Yin. 2022. UniXcoder: Unified cross-modal pre-training for code representation. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 7212–7225, Dublin, Ireland. Association for Computational Linguistics.

Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, et al. 2020. Graphcodebert: Pre-training code representations with data flow. *ArXiv preprint*, abs/2009.08366.

Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long short-term memory. *Neural computation*, 9(8):1735–1780.

Harshit Joshi, José Cambronero Sanchez, Sumit Gulwani, Vu Le, Gust Verbruggen, and Ivan Radiček. 2023. Repair is nearly generation: Multilingual program repair with llms. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 37, pages 5131–5140.

Aditya Kanade, Petros Maniatis, Gogul Balakrishnan, and Kensen Shi. 2020. Learning and evaluating contextual embedding of source code. In *International conference on machine learning*, pages 5110–5121. PMLR.

Mike Lewis, Yinhan Liu, Naman Goyal, Marjan Ghazvininejad, Abdelrahman Mohamed, Omer Levy, Veselin Stoyanov, and Luke Zettlemoyer. 2020. BART: Denoising sequence-to-sequence pre-training for natural language generation, translation, and comprehension. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pages 7871–7880, Online. Association for Computational Linguistics.

Zhen Li, Deqing Zou, Shouhuai Xu, Hai Jin, Yawei Zhu, and Zhaoxuan Chen. 2022. SySeVR: A framework for using deep learning to detect software vulnerabilities. *IEEE Transactions on Dependable and Secure Computing*, 19(4):2244–2258.

Zhen Li, Deqing Zou, Shouhuai Xu, Xinyu Ou, Hai Jin, Sujuan Wang, Zhijun Deng, and Yuyi Zhong. 2018. VulDeePecker: A deep learning-based system for vulnerability detection. In *Proceedings 2018 Network and Distributed System Security Symposium*. Internet Society.

Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. 2019. Roberta: A robustly optimized bert pretraining approach.

Benjamin Livshits and Thomas Zimmermann. 2005. Dynamine: finding common error patterns by mining software revision histories. *ACM SIGSOFT Software Engineering Notes*, 30(5):296–305.

Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin Clement, Dawn Drain, Daxin Jiang, Duyu Tang, et al. 2021. Codexglue: A machine learning benchmark dataset for code understanding and generation. *arXiv preprint arXiv:2102.04664*.

Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. 2022. Codegen: An open large language model for code with multi-turn program synthesis. *arXiv preprint arXiv:2203.13474*.

OpenAI. 2024. Gpt-4 technical report.

Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. 2002. Bleu: a method for automatic evaluation of machine translation. In *Proceedings of the 40th annual meeting of the Association for Computational Linguistics*, pages 311–318.

Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, et al. 2019. Language models are unsupervised multitask learners. *OpenAI blog*, 1(8):9.

Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J Liu. 2020. Exploring the limits of transfer learning with a unified text-to-text transformer. *The Journal of Machine Learning Research*, 21(1):5485–5551.

Shuo Ren, Daya Guo, Shuai Lu, Long Zhou, Shujie Liu, Duyu Tang, Neel Sundaresan, Ming Zhou, Ambrosio Blanco, and Shuai Ma. 2020. Codebleu: a method for automatic evaluation of code synthesis.

Yeonhee Ryou, Sangwoo Joh, Joonmo Yang, Sujin Kim, and Youil Kim. 2022. Code understanding linter to detect variable misuse. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, pages 1–5.

Alexey Svyatkovskiy, Shao Kun Deng, Shengyu Fu, and Neel Sundaresan. 2020. Intellicode compose: Code generation using transformer. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1433–1443.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. *Advances in neural information processing systems*, 30.

Song Wang, Devin Chollak, Dana Movshovitz-Attias, and Lin Tan. 2016. Bugram: bug detection with n-gram language models. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, pages 708–719.

Yue Wang, Hung Le, Akhilesh Deepak Gotmare, Nghi DQ Bui, Junnan Li, and Steven CH Hoi. 2023. Codet5+: Open code large language models for code understanding and generation. *arXiv preprint arXiv:2305.07922*.

Yue Wang, Weishi Wang, Shafiq Joty, and Steven C.H. Hoi. 2021a. CodeT5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pages 8696–8708, Online and Punta Cana, Dominican Republic. Association for Computational Linguistics.

Yue Wang, Weishi Wang, Shafiq Joty, and Steven CH Hoi. 2021b. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. *arXiv preprint arXiv:2109.00859*.

Yaqin Zhou, Shangqing Liu, Jingkai Siow, Xiaoning Du, and Yang Liu. 2019. Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks. *Advances in neural information processing systems*, 32.

9. Language Resource References

Aditya Kanade, Petros Maniatis, Gogul Balakrishnan, and Kensen Shi. 2020. Learning and evaluating contextual embedding of source code. In *International conference on machine learning*, pages 5110–5121. PMLR.

Yaqin Zhou, Shangqing Liu, Jingkai Siow, Xiaoning Du, and Yang Liu. 2019. Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks. *Advances in neural information processing systems*, 32.