2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)

Unifying Defect Prediction, Categorization, and Repair by Multi-task Deep Learning

Chao Ni[†], Kaiwen Yang[†], Yan Zhu[†*], Xiang Chen[‡], and Xiaohu Yang[†]
[†]Zhejiang University, China. Email: {chaoni,kwyang,yan.zhu,yangxh}@zju.edu.cn
[‡] Nantong University, China. Email: xchencs@ntu.edu.cn

***Abstract*—Just-In-Time defect prediction models can identify defect-inducing commits at check-in time and many approaches are proposed with remarkable performance. However, these approaches still have a few limitations which affect their effectiveness and practical usage: (1) partially using semantic information or structure information of code, (2) coarsely providing results to a commit (buggy or clean), and (3) independently investigating the defect prediction model and defect repair model.**

In this study, to handle the aforementioned limitations, we propose a unified defect prediction and repair framework named COMPDEFECT, which can identify whether a changed function inside a commit is defect-prone, categorize the type of defect, and repair such a defect automatically if it falls into several scenarios, e.g., defects with single statement fixes, or those that match a small set of defect templates. Technically, the first two tasks in COMPDEFECT are treated as a multiclass classification task, while the last task is treated as a sequence generation task.

To verify the effectiveness of COMPDEFECT, we first build a large-scale function-level dataset (i.e., 21,047) named Function-SStuBs4J and then compare COMPDEFECT with tens of state-of-the-art (SOTA) approaches by considering five performance measures. The experimental results indicate that COMPDEFECT outperforms all SOTAs with a substantial improvement in three tasks separately. Moreover, the pipeline experimental results also indicate the feasibility of COMPDEFECT to unify three tasks in a model.

***Index Terms*—Just-in-time Defect Prediction, Defect Categorization, Defect Repair**

I. INTRODUCTION

The software development process is evolving rapidly with frequently changing requirements, various development environments, and diverse application scenarios. It tends to release software versions in a short-term period. Such a rapid software development process and limited Software Quality Assurance (SQA) resources have formed a strong contradiction. Just-In-Time defect prediction (JIT-DP) [1]–[4] is a novel technique to predict whether a commit will introduce defects and it can help practitioners prioritize limited SQA resources on the riskiest commits during the software development process. Compared with coarse-grained level (i.e., class/file/module) defect prediction approaches, JIT-DP works at the fine-grained level to provide hints about potential defects. Though many approaches have been proposed to make a great process in JIT-DP, there still has several limitations in previous work.

Code semantic information and code structure information are not fully used. Many approaches [5]–[7] are proposed based on the commit-level metrics proposed by Kamei et al. [1]. These metrics are quantitative indicators of modified codes without considering the semantics of codes. Recently, deep learning based approaches [4], [8] consider the semantic (i.e., the code tokens' implication around its modified context) and the structure information (e.g., the relation among commits, hunks [9], modified files, modified lines, and tokens) of a change. However, the semantic information is not comprehensive [4] but only represents the modified lines and is part of the understanding of the code commit. The structure information is not real representation of code structure [8] but only represents the structure of *git diff*. Therefore, the semantic information (e.g., the code tokens' implication around its modified or unmodified context) and the structure information (e.g., data flow information of code) should be deeply excavated and fully used simultaneously since previous works [10], [11] have shown the advantages of structure information for understanding the functionality/semantic of code.

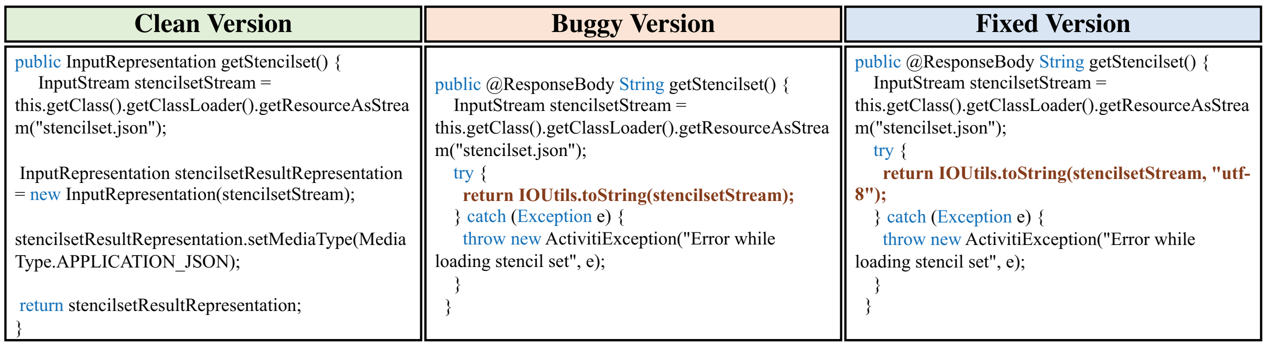
Prediction type is coarse-grained from two sides: decision and location. Most of the existing work [1], [4], [12], [13] can only predict whether a commit or a changed file is buggy or clean without more precise information about the type of defect (e.g., *Missing Throws Exception* and *Less Specific If*), which may help developers better understand the categories of the defect and consequently help fix it. Additionally, a commit or a changed file may involve several hunks which may modify a few functions. For a commit predicted as a buggy one, it may be unclear where the defect exactly exists.

Solution to automatically fix defects once identified is scarcely provided. Some approaches [1], [3] focus on defect identification, while some approaches [14], [15] focus on defect repair. However, none of the prior studies treat defect identification tasks and defect repair tasks simultaneously although they are closely linked in development activities.

In this study, to address these limitations, we propose a unified defect prediction and repair framework, COMPDEFECT, by building a multi-task deep learning model, which can ① predict whether a changed function inside a commit is defect-prone, ② categorize the type of defect, and ③ repair the defect automatically. Considering that one of these tasks, defect repair, is an important but difficult software engineering problem, we follow previous work [14], [16]–[18] by focusing on simple types of defects, such as defects with single statement fixes, or that match a small set of defect templates.

In general, COMPDEFECT consists of two stages: **an offline learning stage and an online application stage**. ① In the

[*] Yan Zhu is the corresponding author.



Clean Version

```
public InputRepresentation getStencilset() {
    InputStream stencilsetStream =
this.getClass().getClassLoader().getResourceAsStrea
m("stencilset.json");

 InputRepresentation stencilsetResultRepresentation
= new InputRepresentation(stencilsetStream);

stencilsetResultRepresentation.setMediaType(Media
Type.APPLICATION_JSON);

 return stencilsetResultRepresentation;
}
```

Buggy Version

```
public @ResponseBody String getStencilset() {
   InputStream stencilsetStream =
this.getClass().getClassLoader().getResourceAsStrea
m("stencilset.json");
   try {
     return IOUtils.toString(stencilsetStream);
   } catch (Exception e) {
     throw new ActivitiException("Error while
loading stencil set", e);
   }
 }
```

Fixed Version

```
public @ResponseBody String getStencilset() {
   InputStream stencilsetStream =
this.getClass().getClassLoader().getResourceAsStrea
m("stencilset.json");
   try {
     return IOUtils.toString(stencilsetStream, "utf-
8");
   } catch (Exception e) {
     throw new ActivitiException("Error while
loading stencil set", e);
   }
 }
```

Fig. 1: The different contents of a specific function in different states: clean version, buggy version, and fixed version.

offline learning stage, we first build a large-scale function-level dataset named Function-SStuBs4J by extracting the function body where the modified lines exist in the ManySStuBs4J dataset, which is originally collected by Rafael-Michael et al. [16] from 1,000 popular open-source Java projects and covers 16 defect patterns. In particular, we need to extract three versions of each function body context where the modified lines exist in a commit: the clean version, the buggy version, and the fixed version. The clean version means the version before the defect was introduced, the buggy version means the version when the defect was introduced into the function, and the fixed version means the version when the defect was fixed. Then, the three versions are treated as the input of CompDefect to complete two main tasks: (1) a multiclass classification task and (2) a code sequence generation task. The first task can help to identify the buggy function and categorize the type of defect, while the second task can generate the patch to repair the defect. ② In the online application stage, for a given commit, we first identify the modifications in each $hunk$. Then, for each modification, we extract the function body where the modification exists. After that, we extract the corresponding previous function body before the modification occurs. Finally, each modified function with two versions of the function body (i.e., the current version and the version before the modification introduced) are fitted into the trained CompDefect to predict its defect-proneness and subsequently repair it once identified as a defect-inducing one.

To verify the effectiveness of our proposed model CompDefect, we conduct a comprehensive study on Function-SStuBs4J with tens of state-of-the-art (SOTA) approaches on several tasks involving: defect prediction [4], [8], defect categorization [19], and defect repair [14]. Compared with SOTAs, the superiority of CompDefect is highlighted. In particular, for the defect prediction task, on average, CompDefect improves baselines (i.e., DeepJIT and CC2Vec) by 39.0%-41.7% and by 34.7%-37.3% in terms of F1-score and AUC, respectively. For the defect categorization task, on average, CompDefect also improves two types of baselines (i.e., pre-trained models and none BERT-based models) by at least 62.6% and by at least 50.1%, respectively. For the defect repair task, CompDefect still outperforms seven automatic defect repair baselines and makes an improvement by 83.1% in terms of Accuracy on average. Finally, the experimental results also indicate acceptable performances in the pipeline setting (i.e., 28.5% correct repair after correct classification). Eventually, this paper makes the following main contributions:

- **Technique.** We propose a unified defect prediction and repair framework named CompDefect for function-level software maintenance[1], which can automatically identify the defect-proneness of a changed function inside a commit, categorize the type of the defect, and repair the defect with the appropriate patch by studying the relations among three different versions of the changed method/function.
- **Dataset.** We extend and purify a new function-level simple statement dataset named Function-SStuBs4J on the basis of ManySStuBs4J. This dataset can provide more contextual information on modified functions in commits, including three versions (i.e., clean version, (non-)buggy version, and (non-)fixed version) of a function. To our best knowledge, this is the first function-level multiclass dataset that can provide comprehensive information on a changed function in a commit and its corresponding repaired patch.
- **Empirical Study**. We comprehensively investigate the value of unifying defect prediction, defect categorization, and defect repair into a unitary model. The results indicate CompDefect outperforms the state-of-the-art and better utilizes the correlation among three related tasks.

II. Background and Motivation

A. Function Quality State Relevance

A function is the basic unit of a specific implementation for a given requirement in a modern software development scenario. Meanwhile, its functionality and quality status may undergo some changes over time. Take an example from the project of *Activiti* shown in Fig. 1. There exists a function named $getStencilset$ [20] and it has three different versions at different timestamps: clean version ($Func_C$), buggy version ($Func_B$), and fixed version ($Func_F$). The buggy version represents the

[1]https://github.com/jacknichao/CompDefect

state when the defect was introduced into the function, the clean version represents the state before the defect was introduced, and the fixed version represents the state when the defect was fixed. Therefore, the definition of a function quantity state is highly related to the changing relationship between different versions of a specific function. That is, we can define a function as a buggy one by comparing its code change difference before and after. The relevance among these variants of function should be taken seriously.

B. Software Quality Assurance Task Relevance

The defects can seriously affect the functionality of the software and eventually cause huge economic losses and even threaten people's lives. Thus, many software quality assurance approaches are proposed such as software defect prediction (SDP) aims at identifying potential risks in code before the software is released, software defect categorization (SDC) aims to explain the specific defect type for developers better understanding, software defect location (SDL) helps to fight out where the defect exists, and software defect repair (SDR) helps to fix the identified defects to improve software quality for accelerating software development. All the approaches are extremely important to SQA and they also have some relationship with each other. That is, SDP and SDC help to prevent risk to end-users ahead and help developers to better understand the characteristic of defects for locating. SDL help to find the location of defect for SDR. Even, an end-to-end solution is detecting the defect (SDP) first and then repairing it subsequently (SDR). Therefore, we find that these SQA tasks are correlated with each other and the relevance among these tasks should also be taken seriously.

Motivating. *The function quality states (i.e., clean, buggy, and fixed) have intrinsic relevance with each other and the software quality assurance activities (i.e., prediction, categorization, and repair) have extrinsic relevance. Intuitively, we should pay more attention to the relevance among code, and treat SQA tasks simultaneously.*

III. APPROACH: COMPDEFECT

To dig into the relevance of both functions and SQA tasks, in this paper, we propose a novel multi-task deep learning-based approach, COMPDEFECT, to better ensure the software's qualities. Multi-task deep learning model [21] aims at training with the same dataset for multiple tasks simultaneously by using the shared representations to learn the common ideas between a few related tasks, which consequently increases the efficiency of the data and potentially accelerates learning speed for related or downstream tasks. Intuitively, we think defect prediction, categorization, and repair tasks are relevant and can benefit each other. The first two tasks (prediction and categorization) can be treated as a multiclass classification task, while the last one (repair) can be treated as a sequence generation task. Meanwhile, the classification task and the generation task can share one common encoder to extract semantic structure information of functions. When performing the generation task, the knowledge of the classification task is utilized to help generate a better code repair.

The overall framework of our approach is illustrated in Fig. 2, which contains three sub-figures. The left one (i.e., Fig. 2(a)) illustrates the structure of COMPDEFECT, while the other two represent the encoder (i.e., Fig. 2(b) is GraphCodeBERT [11]) and decoder (i.e., Fig. 2(c) is Transformer Decoder [22]) used in our proposed COMPDEFECT, respectively. For training COMPDEFECT, we need three versions of a function as the input. That is the clean version of a function, the buggy version of a function, and the fixed version of a function. Details of COMPDEFECT are presented in the following subsections.

A. Defect Prediction and Categorization: Classification Task

The defect prediction and categorization tasks aim to classify whether a function is defective and what defect types it belongs to. Therefore, we need a classifier model and adopt GraphCodeBERT [11] as the encoder (as shown in Fig. 2(b)) of COMPDEFECT, which is a pre-trained model for programming language considering the structure of code. Particularly, it utilizes the semantic-level information of code (i.e., data flow) for pretraining instead of only using the token sequence of code like CodeBERT.

The input of COMPDEFECT's encoder involves two function bodies: $Func_C$ and $Func_B$. Then, we use tree sitter [23] to transform the two functions into code tokens to build the data flow graph (DFG), which can help to construct the dependencies among variables. In particular, for a source code $C = \{c_1, c_2, \ldots, c_m\}$, COMPDEFECT first parses the source code into an abstract syntax tree (AST), which includes syntax information of the code. Besides, the leaves in AST are used to identify the sequence of variable $V = \{v_1, v_2, \ldots, v_k\}$. Therefore, the variable is treated as a node of the graph, while the relationship between v_i and v_j is treated as a directed edge from v_i to v_j of the graph $e = \langle v_i, v_j \rangle$, which means the value of v_j comes from v_i. We denote the collection of directed edges as $E = \{e_1, e_2, \ldots, e_l\}$ and consequently the graph is denoted as $G(C) = (V, E)$, which represents the dependency relationship among variables in the source code C. Then, for the clean version of the function, we concatenate clean code and the collection of variables as the input sequence $X_{clean} = \{[CLS], C_{clean}, [SEP], V_{clean}\}$ and accordingly, the buggy version of the function is transformed as $X_{buggy} = \{[CLS], C_{buggy}, [SEP], V_{buggy}\}$. $[CLS]$ is a special token in front of the whole sequence, $[SEP]$ is another special token to split two kinds of data types. After that, we get a pair (X_{clean}, X_{buggy}) that represents clean and buggy versions of a given function.

COMPDEFECT's encoder takes as input each pair (i.e., (X_{clean}, X_{buggy})) and outputs a series of contextual embedding vectors of each token $H = \{h_{[CLS]}, h_C, h_{[SEP]}, h_V\}$, in which $h_C = \{h_{c_1}, h_{c_2}, ..., h_{c_n}\}$ and $h_V = \{h_{v_1}, h_{v_2}, ..., h_{v_n}\}$. We only use the embedding vector of [CLS] for the classification task following [11], which works as the contextual embedding of the whole input. Therefore, we get the semantic contextual

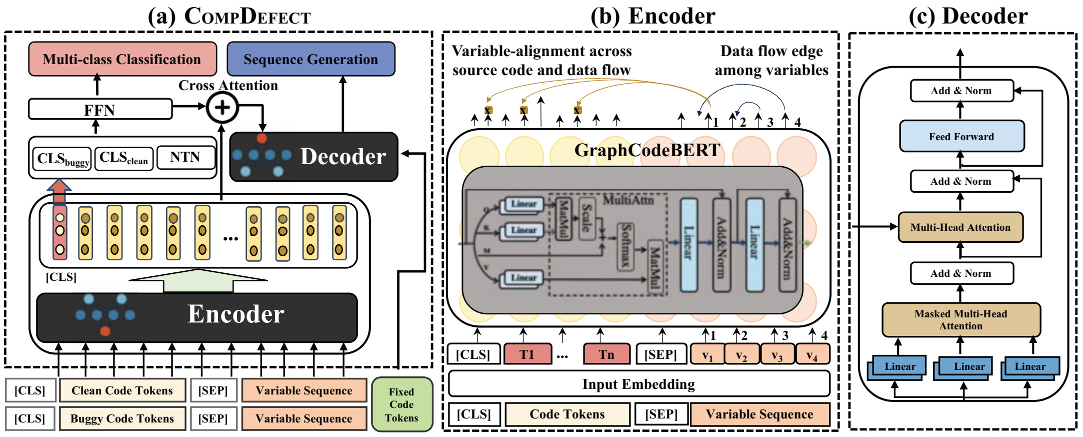


Fig. 2: (a) The overall framework of CompDefect. (b) The detail of the encoder. (c) The detail of the decoder.

representation of the clean function $h_{[CLS]_{clean}}$ and the buggy function $h_{[CLS]_{buggy}}$.

To capture the relation between the clean function (i.e. $h_{[CLS]_{clean}}$) and the buggy function (i.e. $h_{[CLS]_{buggy}}$), we adopt neural tensor network (i.e., NTN, denoted as Γ) and denote the relation as h_{NT}. The three vectors are concatenated and then fed into a fusion network consisting of multiple feed-forward networks. Finally, we get a more informative vector h_{change}, which will be fed into an MLP classifier. The classifier outputs the probability of each class.

For defect prediction and categorization task, given a pair (X_{clean}, X_{buggy}), the goal is to learn the probability of $P_\theta\left(y|(X_{clean}, X_{buggy})\right)$. This task is optimized with a cross entropy loss, denoted as $Loss_{cls}$.

B. Defect Repair: Sequence Generation Task

The defect repair task aims to generate a fixed function for an identified buggy one, directly. Therefore, we need a generation model and adopt Transformer decoder [22] as the decoder of CompDefect. The standard seq2seq architecture usually has a cross-attention mechanism connecting the encoder and decoder. To fully use the information embedded in the encoder, we replace $h_{[CLS]_{buggy}}$ with h_{change} and the final vector H_{cross} is used to interact with the decoder:

$$H_{cross} = h_{change} \oplus h_{C_{buggy}} \oplus h_{[SEP]_{buggy}} \oplus h_{V_{buggy}}$$

Notice that we use h_{change}, the fused vector containing both clean and buggy information and the relation between them, instead of $h_{[CLS]_{buggy}}$, in order to involve more knowledge of classification task when performing generation tasks, we believe that the valuable embedded information for classification task is also useful for generation task.

For the sequence generation task, the goal is to learn the probability of $P_\theta(Y|X) = \prod_{j=0}^{n} P_\theta(y_j|y_{j-1}, y_{j-2}, \ldots, y_0, (X_{clean}, X_{buggy}))$, n is the maximum length of target. This task is optimized with a stand sequence generation loss, denoted as $Loss_{gen}$. Meanwhile, we also adopt Beam search which memorizes the n best sequences up to the current state of the decoder. The successors of these memorized states are computed and sorted based on their cumulative probability. After that, the next n best sequences are passed to the next status of the decoder. As for evaluating CompDefect, we only choose the sequence with the highest probability.

C. Optimization of CompDefect

When a model has more than one task, a few task-specific objective functions need to be combined into a single aggregated one that the model tries to maximize it. Therefore, it is extremely important to exactly combine various objective functions into one that is the most suitable for multi-task learning. In CompDefect, there mainly exist two tasks: multi-class classification task and sequence generation task. Since in our usage scenario (i.e., function-level software defect prediction and defect repair), we think the two tasks are equivalently important. Thus, CompDefect addresses the multi-task optimization by balancing the individual objective functions for the two different tasks. That is, the final optimization function ($Loss(X)$) is the sum of two individual objective functions ($Loss_{cls}(X)$ for the classification task, and $Loss_{gen}(X)$ for the sequence generation task), which is the one that CompDefect tries to minimize. Formally, $Loss(X) = \alpha * Loss_{cls}(X) + \beta * Loss_{gen}(X)$ (both are set to 1 in our study).

IV. Experimental Setting

We first introduce the dataset we experiment on, then we present the baselines for different types of tasks. Following that, the evaluation metrics, and experimental settings are presented.

A. Dataset

ManySStuBs4J [16] is a collection of single-statement defect-fix changes and is annotated by whether those changes match a few defect patterns (i.e., 16 patterns [16]). Besides, considering the importance but challenge of program repair, fixing simple defects (e.g., one-line defects or defects that fall into a small set of templates) is a good way to obtain acceptable performance and ManySStuBs4J is a good starting point dataset since it is widely used in many software quality-related tasks [24], [25].

However, the details of this dataset are too simple to satisfy our task's requirement. That is, the original dataset mainly contains the types of defects, the one statement buggy code, and one statement fixed code. However, our model needs the whole information about a changed function to capture more information inside the code. Besides, our study aims at combining the defect prediction task, defect categorization task and defect repair task together, we also need to collect negative commits (i.e., clean commits) from studied projects since the original dataset only contains the positive commits (i.e., buggy commit). Therefore, we need an extended version of ManySStuBs4J.

More precisely, we extend the existing dataset ManySStuBs4J from the following aspects: ❶ **Function Body**. The original dataset only provides the buggy line number and its corresponding fixed line number in commits. We need to extract the bodies of the two types of functions for the positive instances. ❷ **Negative Functions**. We use the opposite strategy (i.e., used in ManySStuBs4J) to select negative commits for extracting negative functions since our work focuses on defect prediction which needs both positive functions and negative functions. ❸ **Multiple Versions**. We extract three versions (i.e., clean version, (non-)buggy version, and (non-)fixed version) of all functions since we believe the difference among the three versions can help the model better understand the semantic information of a function.

Here, we introduce the process of our dataset extension and data extraction. Notice that we start from the COMMIT stored in ManySStuBs4J rather than the FUNCTION. In particular, as for extracting the positive/negative functions, we follow the same steps (e.g., selecting the same Java projects, identifying non-defect-fixing/defect-fixing commits, selecting single statement changes, creating Abstract Syntax Trees, and filtering out clear refactoring) as the ManySStuBs4J took:

1) We identify bug-fixing(non-bug-fixing) commits whose commit message contains one(none) of these keywords (i.e., error, buy, fix, issue, mistake, incorrect, fault, defect, flaw, and type). That is, we use the opposite selection strategy between selecting positive commits and negative commits.
2) We follow the same strict criteria used in ManySStuBs4J to select the commits with single statement modification.
3) We identify each scope of the modification (i.e., hunk in git) in the filtered commits and extract their bug-fixing(non-bug-fixing) function bodies. That is, we split one commit into hunks and extract the function body of modification in each hunk.
4) We use the *git blame* to find where the deleted lines are introduced and consequently identify the corresponding bug-inducing(non-bug-inducing) commits. Then, we extract the corresponding bug-inducing(non-bug-inducing) function bodies.
5) We extract the corresponding clean versions (i.e., the last version in time before the bug-inducing(non-bug-inducing) functions are modified) using PyDriller [26]. Therefore, we obtain the three versions of the function body for positive/negative functions.

Meanwhile, we also design four criteria for filtering unsuitable functions in (Steps 3-5), such as: ① Modified statements exist outside a function. This work focuses on the simple scenario, that is, function-level single statement defect prediction and defect repair. Therefore, we filter out those statements that lie outside the scope of a function, for example, statements for defining a global variable or object in a class. ② Defects are introduced in newly added files or functions. COMPDEFECT needs three versions of a specific function. For the newly added defective ones, the clean version does not exist. Therefore, we filter out such cases. ③ Function names cannot be identified. On one hand, the modifications in a commit are made to the function name, and then it is difficult to solve. On the other hand, the line of the function is mapped incorrectly. In ManySStuBs4J, the authors label the line number of buggy or fixed code based on the result of AST, which may not be exactly correct with the original line in the source code file. ④ Other failed issues. When errors occur in PyDriller or in the original dataset, we cannot get the correct result.

Finally, we build the dataset and name it as **Function-SStuBs4J** for defect prediction, defect categorization, and defect repair by extracting the three versions of the function with the help of PyDriller [26] tool. The statistical information is shown in Table I.

TABLE I: The statistics of **Function-SStuBs4J**

Defect Type	# Count	% Ratio	Defect Type	# Count	% Ratio
Same Function Change Caller	381	1.81%	Same Function Less Args	441	2.1%
Change Identifier Used	1,599	7.6%	Same Function More Args	1,169	5.55%
Change Modifier	329	1.56%	Same Function Swap Args	139	0.66%
Change Numeric Literal	991	4.71%	Change Boolean Literal	330	1.57%
Change Operand	161	0.76%	Less Specific If	484	2.3%
Change Binary Operator	523	2.48%	More Specific If	570	2.71%
Change Unary Operator	338	1.61%	Missing Throws Exception	10	0.05%
Wrong Function Name	3,012	14.31%	Delete Throws Exception	32	0.15%

B. Baselines

Considering that there has no existing work which can predict whether a modification to a function may introduce defects, subsequently categorize the type of defect, and consequently repair it automatically, we make a comprehensive comparison among three types of baselines. More precisely, we consider two well-known baselines for the prediction task, consider five baselines for the categorization task which can be divided into two types: BERT-based or none BERT-based, and consider seven state-of-the-art baselines for the repair task suggested by Zhong et al. [27]). Table II gives a brief introduction to these baselines. Notice that we do not consider those defect repair tools especially the ones that need bug-triggering tests since the studied dataset does not have corresponding test cases.

C. Evaluation Metrics

To evaluate the effectiveness of COMPDEFECT, we consider the following performance measures:

Precision is the fraction of correctly classified functions among the predicted ones, which can be calculated as $\frac{TP}{TP+FP}$.

TABLE II: Baselines used in the comparison

Baselines	Tasks			Brief Description	Venue
	P	C	R		
DeepJIT [4]	✔			a CNN model to learn high-dimensional semantic features for both commits message and commit code	MSR 2019
CC2Vec [8]	✔			adopt a Hierarchical Attention Network to capture the hierarchical structure inside a commit	ICSE 2020
BERT [19]		✔		a pre-train deep bidirectional representation from the unlabeled text and consists of 12-layer transformers	arXiv 2018
RoBERTa [28]		✔		a robustly optimized version of BERT	arXiv 2019
CodeBERT [29]		✔		a pre-trained model on the basic of BERT for the programming language	arXiv 2020
TextCNN [30]		✔		a simple but well-perform CNN with one layer of convolution on top of word vectors obtained from an unsupervised neural language model	EMNLP 2014
FastText [31]		✔		a lightweight library for efficient learning of word representations and sentence classification	arXiv 2016
SequenceR [14]			✔	proposes a novel buggy context abstraction process to organize the fault localization information into a representation	TSE 2019
CoCoNut [32]			✔	uses ensemble learning on the combination of CNN and a new context-aware neural machine translation architecture to fix bugs in multiple programming languages	ISSTA 2020
Mashhadi et al. [33]			✔	a variant of CodeBERT-based approach	MSR 2021
Tufano et al. [34]			✔	An RNN-based bug fixing model by abstracting identifiers and literals in the buggy code to simplify the input and output	TOSEM 2019
CODIT [35]			✔	a two-stage approach with LSTM-based Neural Machine Translation Models	TOSEM 2020
Recoder [36]			✔	a syntax-guided decoder to generate edits on the AST of the buggy function/method	FSE 2021
Edits [37]			✔	a model to generate patch by inserting and deleting tokens of buggy code	ASE 2020

$*$"**P**": Defect Prediction; "**C**": Defect Categorization; "**R**": Defect Repair.

Recall measures how many defective functions can be correctly classified, which is defined as $\frac{TP}{TP+FN}$.

F1-score is a harmonic mean of $Precision$ and $Recall$ and can be calculated as: $\frac{2\times P\times R}{P+R}$.

AUC: the Area Under the receiver operator characteristics Curve (AUC) is also used to measure the discriminatory power of CompDefect and baselines, i.e., the ability to differentiate between defective or non-defective functions. AUC calculates the area under the curve plotting the true positive rate (TPR) versus the false positive rate (FPR) while applying multiple thresholds to determine if a function is defect-inducing or not.

Accuracy evaluates the performance of how many functions can be correctly classified. It is calculated as $\frac{TP+TN}{TP+FP+TN+FN}$.

D. Empirical Setting

We implement our CompDefect in Python with the help of the Pytorch framework and pre-trained model on Huggingface. The pre-trained GraphCodeBERT model is used as the encoder for embedding training samples and uses Transformer decoder [22] as the generator of fixed code. Besides, each version function is embedded as a 768-dimensional vector. During the training phase, the parameters of CompDefect are optimized using Adam with a batch size of 32. We also use $ReLu$ and $tanh$ as the activation function. A dropout of 0.1 is used for dense layers before calculating the final probability. The maximum number of epochs in our experiment is 50. The models (i.e., CompDefect and baselines) with the best performance on the validation set are used for our evaluations.

As for dataset split, we use 80%, 10%, and 10% of CompDefect as training data, validation data, and testing data, respectively. Notice that, for each part of the data, we keep the distribution among each type of function as same as the original one.

V. Results and Analysis

To comprehensively evaluate the effectiveness of CompDefect, we investigate the following three research questions.

- **RQ-1:** How does CompDefect perform on defect prediction compared with state-of-the-art baselines?
- **RQ-2:** How does CompDefect perform on defect categorization compared with state-of-the-art baselines?
- **RQ-3:** How does CompDefect perform on defect repair compared with the state-of-the-art baselines?

A. [RQ-1]: Defect Prediction

Objective. Just-in-time (JIT) defect prediction has received much attention in software engineering and many state-of-the-art approaches are proposed [4], [8], [12]. These approaches are built from the simple model (e.g., CBS+) on manually designed features to the complex model (e.g., DeepJIT) on semantic features and have made great progress in the JIT scenario. As for CompDefect based on neural network, it can also identify whether a commit is a defect-inducing one by predicting if the modification to a function will introduce a defect. Therefore, we want to make a comparison between CompDefect with those SOTA semantic features-based approaches.

Experiment Design. We treat recently proposed DeepJIT and CC2Vec as the baseline methods. There are two differences between CompDefect and the two baselines: 1) the two methods can only estimate the defect-proneness of a commit at the commit level and a commit may contain a few hunks which change a few functions, while CompDefect estimates whether a function in a hunk of a commit is defect-inducing. That is, CompDefect can estimate the defect-proneness of a commit at the hunk(function)-level, which means CompDefect is more fine-grained than the baseline ones. 2) the two baselines can only predict a commit as a defect-inducing one or a clean one, while CompDefect can not only predict a commit defect-proneness but also can categorize the types of defects.

Considering the two differences and to make a fair commit-level comparison, we make the following two hypothesizes for COMPDEFECT: 1) a function in a hunk of a commit will be treated as a buggy one if COMPDEFECT predicts it as a non-clean one; 2) a commit is predicted as buggy one if there exists at least one function in a hunk of a commit predicted by COMPDEFECT as buggy one, otherwise the commit is predicted as a clean one.

Meanwhile, considering the difference in the granularity of two baselines (i.e., commit-level), we also make adjustments to the dataset splitting. Particularly, as for training data (i.e., 80% from the original split dataset), we identify the unique commits from both positive and negative functions to build the commit-level training data for the two baselines. Similarly, we adopt the same method to build the validation data and testing data. Therefore, all methods (i.e., DeepJIT, CC2Vec, and COMPDEFECT) are compared on the newly built dataset at the commit level. We adopt the widely used performance measures (i.e., Precision, Recall, F1-score, and AUC) to evaluate the difference among those methods.

TABLE III: Comparison among DeepJIT, CC2Vec and Ours

Approach		Precision	Recall	F1-score	AUC
DeepJIT		0.504	0.352	0.414	0.513
CC2Vec		0.464	0.345	0.396	0.492
COMPDEFECT		**0.750**	**0.619**	**0.679**	**0.785**
Improv.	*DeepJIT*	32.8%	43.2%	39.0%	34.7%
	CC2Vec	38.2%	44.2%	41.7%	37.3%

Results. The evaluation results are reported in Table III and the best performance is highlighted in bold. According to the results, we find that our approach COMPDEFECT has a significant advantage over DeepJIT and CC2Vec on all performance measures. In particular, COMPDEFECT obtains 0.679 and 0.785 in terms of F1-score and AUC, which improves DeepJIT and CC2Vec by 39.0% and 41.7%, by 34.7% and 37.3% in terms of F1-score and AUC, respectively. As for Precision and Recall, COMPDEFECT also has a large improvement. Specifically, COMPDEFECT improves DeepJIT and CC2Vec by 32.8% and 38.2%, by 43.2% and 44.2% in terms of Precision and Recall, respectively. Besides, we surprisingly find that in our scenario, CC2Vec performs a little worse than DeepJIT, which, to some extent, means CC2Vec cannot capture more information than DeepJIT for the existing state-of-the-art techniques can utilize.

Answer to RQ-1: COMPDEFECT *can accurately identify the defect-proneness of commit and outperform state-of-the-arts significantly.*

B. [RQ-2]: Defect Categorization

Objective. Although many approaches [1], [2], [8], [38] have been proposed for JIT-DP, these approaches can only predict a commit or a changed file as defect-inducing or not. They cannot categorize the type of defect. Different from previous work, COMPDEFECT can categorize the type of defect the defect-inducing function belongs to. We totally consider 16 types of defects [16]. Reporting the type of defect rather than "buggy-or-clean" can help developers better understand such a defect. COMPDEFECT makes certain progress in the JIT-DP scenario even if we only consider the one-statement function-level modification setting.

TABLE IV: Comparison among five baselines and COMPDEFECT on categorizing the types of defect

Approach		Precision	Recall	F1-score	AUC_ovo	AUC_ovr
BERT		0.301	0.084	0.083	0.693	0.731
RoBERTa		0.227	0.086	0.085	0.712	0.751
CodeBERT		0.226	0.115	0.124	0.695	0.754
TextCNN		0.223	0.090	0.091	0.685	0.726
FastText		0.334	0.230	0.265	0.657	0.725
COMPDEFECT		**0.401**	**0.295**	**0.319**	**0.723**	**0.776**
Improve	BERT	33.1%	250.5%	283.8%	4.3%	6.1%
	RoBERTa	76.9%	245.1%	274.8%	1.6%	3.3%
	CodeBERT	77.8%	157.5%	158.5%	4.0%	2.9%
	TextCNN	79.9%	226.8%	251.3%	5.5%	6.8%
	FastText	20.2%	28.4%	20.7%	10.0%	7.0%
	Avg.	57.6%	181.7%	197.8%	5.1%	5.2%

*"OVO": stands for one-vs-one; "OVR": stands for one-vs-rest.

Experiment Design. To verify COMPDEFECT's effectiveness in categorizing defects, we choose three BERT-based baselines: BERT, RoBERTa, and CodeBERT, which are widely used in natural language processing and software engineering [39], [40], and then be used for downstream tasks (e.g., multiclass classification). We use these pre-trained models (i.e., "bert-base-uncased", "roberta-base" and "microsoft/codebert-base") from Huggingface. For a fair comparison, the input of these baselines is the same as the input of COMPDEFECT (i.e., buggy version and clean version). Additionally, since the limitation of the maximum length of baselines' input, we use the same strategy in COMPDEFECT to concatenate the two functions vertically to assemble the whole input. Meanwhile, to further investigate the effectiveness of our approach, we also compare COMPDEFECT with another two none BERT-based but well-performed multiple classification approaches (i.e., FastText and TextCNN) since no existing models proposed in JIT-DP for the multi-classification task. We directly use their replications to conduct this experiment to reduce the internal threat.

We also consider the same performance measures used in RQ-1. These classification metrics are defined for binary cases by default. When applying these binary metrics to multiclass, we adopt the "macro" averaging strategies, which are widely adopted in prior work [40]–[43].

Results. The evaluation results are reported in Table IV and the best results are highlighted in bold. On average, we find that COMPDEFECT outperforms baselines by 57.6%, 181.7%, and 197.8% in terms of Precision, Recall, and F1-score, respectively. In particular, COMPDEFECT improves BERT by 33.1%, 250.5%, and 283.8% in terms of Precision, Recall, and F1-score, respectively. Compared with RoBERTa, COMPDEFECT improves RoBERTa by 76.9%, 245.1%, and 274.8% in terms of Precision, Recall, and F1-score, respectively, which means that even though COMPDEFECT and RoBERTa have the same architecture, COMPDEFECT can learn more information with its related task (i.e., defect fix). Compared with CodeBERT, CompDefect improves CodeBERT by 77.8%, 157.5%, and 158.5% in terms of Precision, Recall, and F1-score, respectively, which also means CompDefect can benefit from multi-task learning. As for TextCNN, CompDefect improves it by 79.9%, 226.8%, 251.3% in terms of Precision, Recall, and F1-score. As for FastText, CompDefect also obtains better performance and improves it by 20.2%, 28.4%, and 20.7%, in terms of Precision, Recall, and F1-score, respectively. In terms of AUC, CompDefect still performs the best. All results indicate the priority of CompDefect in categorizing the types of defects.

Meanwhile, we also present the detailed results of each approach on classifying the types of function as shown in Table V, in which we show the numbers of functions that are correctly classified by each approach and highlight the best one with a light gray background color. Additionally, we show the ratio of correctly classified functions by FastText and CompDefect in the last two columns since they perform better, and highlight the top-5 performances. According to the results, we can obtain the following observations: (1). CompDefect performs the best in classifying the non-clean functions and can almost work well on all types of defects except two types (i.e., both "Delete Throws Exception" and "Same Function Swap Args" are incorrectly classified.). (2). Almost all baselines except FastText perform badly on classifying non-clean functions but perform better on clean ones, which means these baselines cannot distinguish the intrinsic difference among these types of defects. (3). No methods can identify the type of "Same Function Swap Args", which indicates the difficulty in understanding the characteristic of this type of defect. By analyzing the definition of this type, it does seem to be hard to understand the difference since the swapped two parameters are the same type. (4). Both BERT-based approaches (e.g., CompDefect) and none BERT-based approaches (e.g., FastText) have varying abilities in classifying the types of defects.

Answer to RQ-2: CompDefect *has a good performance on distinguishing different types of defects, especially for "Wrong Function Name", "Change Modifier", "Change Numeric Literal", "Same Function Less Args" and "Missing Throws Exception".*

C. [RQ-3]: Defect Repair

Objective. Defect repair research is active and mostly dominated by techniques based on static analysis [44] and dynamic analysis [17]. Even existing approaches have achieved promising performance, currently, automated defect repair is limited to simple cases, mostly one-line patches [17], [18]. Recently, defect repair tools based on machine learning, especially for deep learning technology are proposed [14], [32]–[37], which promote the further development of defect repair, and mainly focus on one-line path scenario. Therefore, we want to evaluate the performance difference between the SOTA baselines and CompDefect.

Experiment Design. Though defect categorization and defect repair should be intrinsically connected in practice (e.g., CompDefect), all baselines are proposed on the hypothesis that the defects are correctly identified and located. That is, these baselines cannot identify whether a function is defect-inducing and cannot categorize the type of defect in such a function. Therefore, for a fair comparison, we compare CompDefect with baselines on all defective functions with the assumption that the defective lines are correctly identified.

We filter all negative functions (i.e., clean functions) in the original dataset and keep only positive functions (i.e., buggy functions, 10,508). Meanwhile, different approaches may have various pre-processing (e.g., $abstraction$) on a given buggy function. However, not all these operations can be successfully executed even using the scripts provided by the authors of all baselines. Therefore, for each baseline, the final evaluation size of positive functions may not always be the same, and we remove the functions which are incorrectly pre-processed and keep the left as the whole dataset. Then, for each baseline, we split all these kept positive functions into 80%, 10%, and 10% as training data, validating data, and testing data, respectively.

In addition, the split of all positive functions differs from the one used to train our approach CompDefect. That is, the testing data of the positive function and the training data in RQ-2 may overlap. Thus, we need to remove the overlapping for a fair comparison between baselines and CompDefect.

Here, we take one of the baselines ("SequenceR") as an example to explain the setting. We filter negative functions in the original dataset and keep only positive functions. We refer to it as Function-SStuBs4J$_{pos}$, which contains 16 types of positive functions. Besides, SequenceR conducts an *abstraction* operation on the function. However, some functions in Function-SStuBs4J$_{pos}$ cannot be executed successfully with the tool provided by SequenceR. Thus, we filter out these functions and finally Function-SStuBs4J$_{pos}$ has positive functions with 14 types. We split 80%, 10%, and 10% of Function-SStuBs4J$_{pos}$ as training data (7,474), validation data (934), and testing data (934), respectively, and the distribution among each type of function as same as the original one.

Moreover, for fully evaluating the capability of CompDefect, we directly evaluate the CompDefect trained on original training data (i.e., training data from Function-SStuBs4J, denoted as Training$_{ori}$) on the testing data of Function-SStuBs4J$_{pos}$, denoted as Testing$_{pos}$. However, since Function-SStuBs4J$_{pos}$ and Function-SStuBs4J are not the same split, Training$_{ori}$ may contain the positive functions in Testing$_{pos}$. So, for a fair comparison, we identify the intersection of Training$_{ori}$ and Testing$_{pos}$, and remove the intersection from Testing$_{pos}$ and denote it as Testing$_{pos\_fltr}$. Finally, we adopt the widely used measure, Accuracy, to evaluate their performances.

Results. The evaluation results are reported in Table VI and the best results are highlighted in bold. The first column shows the SOTA approaches, and the second column illustrates the setting including the used data size as well as the indicator of whether baseline approaches adopt the *abstraction* operation

TABLE V: Performance of all approaches on each category

Defect Category	**# Test**	**# BERT**	**# RoBERTa**	**# CodeBERT**	**# FastText**	**# TextCNN**	**# CompDefect**	**% FastText**	**% CompDefect**
Same Function Change Caller	38	0	0	0	4	0	**7**	10.5%	18.4%
Changed Identifier Used	160	0	0	10	**50**	9	45	31.3%	28.1%
Change Modifier	33	1	1	3	9	0	**14**	27.3%	42.4%
Change Numeric Literal	99	8	10	21	**38**	5	**38**	38.4%	38.4%
Change Operand	16	0	0	0	0	0	**3**	0	18.8%
Change Binary Operator	53	2	2	6	**12**	3	10	22.6%	18.9%
Change Unary Operator	34	0	0	0	**3**	0	**3**	8.8%	8.8%
Wrong Function Name	301	81	81	121	142	103	**160**	47.2%	53.2%
Same Function Less Args	44	0	0	0	11	0	**13**	25.0%	29.5%
Same Function More Args	117	2	3	11	26	5	**27**	22.2%	23.1%
Same Function Swap Args	14	0	0	0	0	0	0	0	0
Change Boolean Literal	33	0	0	0	**6**	0	5	18.2%	15.2%
Less Specific If	49	0	0	0	**4**	0	**4**	8.2%	8.2%
More Specific If	57	0	0	1	7	0	**11**	12.3%	19.3%
Missing Throws Exception	1	0	0	0	0	0	**1**	0	100.0%
Delete Throws Exception	3	0	0	0	**1**	0	0	33.3%	0
Clean	1,054	**1,051**	1,044	1,008	902	1,040	837	85.6%	79.4%
All	**2,106**	**1,145**	**1,141**	**1,181**	**1,215**	**1,165**	**1,178**		

or not. Following that, we list the performance of baselines independently. Finally, the following four columns describe the filtered testing size (i.e., remove the intersection of Training$_{ori}$ and Testing$_{pos}$), and the compared performances between baselines and CompDefect.

According to the results, we find that on Testing$_{pos}$: (1) These approaches (i.e., Tufano et al, SequenceR, CODIT, and Recoder) which adopt the *abstraction* operation perform similarly in terms of accuracy (0.101-0.136). Meanwhile, these approaches (i.e., Edits, CoCoNut, and Mashhadi et al.) which do not adopt the *abstraction* operation perform similarly (except Edits) with the accuracy performance of 0.225-0.230. (2) Generally, methods without *abstraction* have a better performance than the ones with *abstraction*.

TABLE VI: Comparison between baselines and **CompDefect** on defect repair

Approaches	**Setting**		**Testing$_{pos}$**	**Testing$_{pos\_fltr}$**		
	Abs.	**# Train/Valid/Test**	**Acc.**	**# Size**	**Acc. (BLs)**	**Acc. (Ours)**
Tufano et al.	✔	7,042/880/881	0.132	194	0.139	**0.247**
SequenceR		7,474/934/934	0.136	179	0.193	**0.240**
CODIT		7,395/923/925	0.101	220	0.109	**0.246**
Recoder		5,801/726/724	0.104	157	0.115	**0.231**
Edits	✘	8,406/1,050/1,052	0.066	1,052	0.066	**0.231**
CoCoNut		8,406/1,050/1,052	0.225	1,052	0.225	**0.231**
Mashhadi et al.		8,406/1,050/1,052	0.230	1,052	0.230	**0.231**

$*$ *Abs.*: abstraction, *Acc.*: accuracy, *BLs*: Baselines.

Besides, on Testing$_{pos\_fltr}$, we achieve the following observations: (1) Four methods with *abstraction* operation perform similarly to the ones on Testing$_{pos}$ setting and achieve a better accuracy performance of 0.109-0.193. (2) CompDefect performs best and improves Tufano et al., SequenceR, CODIT, and Recoder by 77.7%, 24.4%, 125.7%, and 100.9%, respectively. (3) CompDefect also performs better than the three methods without *abstraction* and improves them by 2.7%-250.0%. (4) Methods including CompDefect without *abstraction* operation seem to perform better on the defect repair task. (5) Compared with Mashhadi et al. which is a CodeBERT-based method and fine-tuned on the same dataset used in this paper, we find that CompDefect and Mashhadi et al. have comparable performance, which indicates the superiority of the pre-trained model on the defect repair task.

Answer to RQ-3: CompDefect *can perform well in repairing defects and pre-trained models have the advantage in the defect repair task. Meanwhile, the abstraction operation may negatively impact the performances of models when repairing defects.*

VI. Discussion

A. *The pipeline performance of* CompDefect

Currently, since there have no existing approaches that can address three tasks (defect prediction, categorization, and repair) simultaneously, we treat the three tasks separately and compare CompDefect with tens of state-of-the-art baselines on these tasks, independently. However, the practice is that the three tasks should be performed in a row. That is, the evaluation of CompDefect's bug repair should be conducted on the correctly classified results. Notice that, in our setting, the predicting task and categorizing are intrinsically connected since we classify whether a commit is defective by categorizing whether one of the modified functions in the commit is a non-clean one. We want to figure out the effectiveness of CompDefect in the pipeline setting (repairing defects on correctly categorized functions) on the testing dataset.

Table VII shows the detailed experimental results. In particular, the first two columns show defect types and their corresponding numbers. The following two columns present the number of correctly classified functions and correctly repaired, independently. Then, the next two columns list the numbers of correctly repaired functions after correctly classified as well as their ratio. Finally, we present the overall performance when functions are correctly classified and repaired.

According to the results, we obtain the following observations: (1) On average, about two-seventh (i.e., 0.285) of buggy functions can be correctly repaired after being correctly classified. (2) Defect repair is more difficult than defect categorization and the number of correctly fixed functions

TABLE VII: The pipeline performance of **COMPDEFECT**

Defect Category	**# Test**	**# Classification**	**# Repair**	**# Classification&Repair**	**% Ratio**	**% Overall**
Same Function Change Caller	38	7/38	6/38	2/38	0.286	0.053
Changed Identifier Used	160	45/160	35/160	19/160	**0.422**	**0.119**
Change Modifier	33	14/33	9/33	9/33	**0.643**	**0.273**
Change Numeric Literal	99	38/99	18/99	11/99	0.289	0.111
Change Operand	16	3/16	2/16	0/16	0	0
Change Binary Operator	53	10/53	7/53	6/53	**0.600**	**0.113**
Change Unary Operator	34	3/34	3/34	1/34	0.333	0.029
Wrong Function Name	301	160/301	111/301	97/301	**0.606**	**0.322**
Same Function Less Args	44	13/44	5/44	3/44	0.231	0.068
Same Function More Args	117	27/117	24/117	14/117	**0.519**	**0.120**
Same Function Swap Args	14	0/14	1/14	0/14	0	0
Change Boolean Literal	33	5/33	7/33	1/33	0.200	0.030
Less Specific If	49	4/49	10/49	1/49	0.250	0.020
More Specific If	57	11/57	5/57	2/57	0.182	0.035
Missing Throws Exception	1	1/1	0/1	0/1	0	0
Delete Throws Exception	3	0/3	0/3	0/3	0	0
ALL	**1,052**	**341/1,052**	**243/1,052**	**166/1,052**	**0.285**	**0.158**

is less than the number of correctly classified functions in each type of defect. (3) COMPDEFECT has a varying performance in classifying and repairing different types of defects. In particular, COMPDEFECT performs well on these types (i.e., "Change Modifier" (0.643), "Wrong Function Name" (0.606), "Change Binary Operator" (0.600), "Same Function More Args" (0.519), and "Changed Identifier Used" (0.422)), but performs badly on four types (i.e., "Change Operand", "Same Function Swap Args", "Missing Throws Exception" and "Delete Throws Exception"). (4) Through a deeper analysis of the performance difference, we find that:

- Data size has a heavy impact on performance. We find that defect types with better performance have larger numbers of training sizes than those with bad performance. For example, "Wrong Function Name" has 2,410 training samples, while "Change Operand", "Missing Throws Exception", and "Delete Throws Exception" have 129, 8, and 26 training samples, respectively.
- Function size (LOC) has an impact on performance. For smaller size functions (e.g., <10), COMPDEFECT can capture their semantic features well and repair them with varying operations. For example, COMPDEFECT can repair "Change Modifier" defect by appropriately adding or removing modifiers to functions though it has a median size training data (i.e., 263).
- Defect complexity has an impact on performance. As for the complex types of defects, COMPDEFECT can't understand the complex types of defects well, which is also hard for developers. For example, "Same Function Swap Args" means fixing the defect by only swapping the order of parameters with identical types. Therefore, if multiple function parameters are of the same type, developers can easily swap two of them without realizing if they do not accurately remember what each argument represents.

Answer to Discussion 1: COMPDEFECT *has a varying performance in classifying and repairing different types of defects and achieves an acceptable performance in the pipeline setting (i.e., repairing defects after correctly classified). Besides, the more training data the model has, the better performance the* COMPDEFECT *achieves.*

B. Ablation Study

We propose a unified model for predicting, categorizing, and repairing defects by adopting the multi-task technology with the combination of two separate technologies: GraphCodeBERT and Decoder. Therefore, we conduct an ablation study to investigate the effectiveness of the combination between GraphCodeBERT and Decoder on these studied tasks. In particular, we directly use GraphCodeBERT to address the defect categorization task (function level) and defect prediction tasks (commit level) and directly use a Decoder to address the defect repair task (function level). The experimental settings are the same as the ones in previous RQs.

The results are presented in Table VIII and the best performances are highlighted in bold. According to the results, we find that COMPDEFECT, a combination of GraphCodeBERT and Decoder, can achieve better performance than the ones obtained by a single approach on almost all tasks considering several performance metrics (except precision on defect prediction). In particular, COMPDEFECT improves GraphCodeBERT by 1.3%-10.1% and 2.1%-49.1% on the defect prediction task and defect categorization task, respectively. Meanwhile, COMPDEFECT also improves Decoder by 20.3% on the defect repair task. Therefore, combining GraphCodeBERT and Decoder is superior to using them directly.

Answer to Discussion 2: *GraphCodeBERT and Decoder can achieve good performance on independent tasks, but combining them appropriately with multiple-task learning techniques (e.g.,* COMPDEFECT*) can obtain better performance.*

C. Threats to Validation

Threats to Internal Validity mainly lie in the potential faults in the implementation of our model. To minimize such threats, we not only implement these methods by pair

TABLE VIII: The ablation study of COMPDEFECT

Method Type	Commit-level (Binary Classification)				Function-level (Multiple Classification)				
	Precision	Recall	F1-score	AUC	Precision	Recall	F1-score	AUC_ovo	AUC_ovr
COMPDEFECT	0.750	**0.619**	**0.679**	**0.785**	**0.401**	**0.295**	**0.319**	**0.723**	**0.776**
GraphCodeBERT	**0.767**	0.562	0.648	0.775	0.338	0.203	0.214	0.708	0.755

Method Type	Function-level (Repair)	Overall	Prediction	Category	Repair
	Accuracy				
COMPDEFECT	**0.231**	COMPDEFECT *vs. Single Task*	↗	↗	↗
Decoder	0.192	*Improvement*	**1.3%-10.1%**	**2.1%-49.1%**	**20.3%**

programming but also make full use of the pre-trained models such as GraphCodeBERT [11] and BERT [19]. Besides, we directly use the original source code of baselines and the same hyperparameters used in the original method are adopted in our paper. All of the datasets used in our study are publicly available from previous work [16], and we extend this dataset for our investigated scenario.

Threats to External Validity mainly lie in the studied projects. To reduce such threats, we opted to select high-popularity Java projects. The popularity of a project is determined by computing the sum of z-scores of forks and stars [16]. However, all studied projects are open-source projects, it is still unknown whether our COMPDEFECT can work well on commercial projects.

Threats to Construct Validity mainly lie in the adopted performance metrics in our evaluations. To minimize such threats, we adopt a few performance metrics widely used in existing work. In particular, we totally consider five performance metrics including Accuracy, Precision, Recall, F1-score, and AUC.

VII. RELATED WORK

A. Just-in-Time Defect Prediction

JIT defect prediction has been an active research topic in recent years since it can identify defect-inducing commit at a fine-grained level at check-in time [2], [6], [38], [45]–[52]. Kamei et al. [1] proposed 14 change-level metrics from five dimensions to build an effort-aware prediction model. Jiang et al. [13] proposed a personalized defect prediction by building a separate prediction model for each developer. Following that, Yang et al. [7], [53] subsequently proposed two approaches (i.e., Deeper and TLEL) for JIT defect prediction. Recently, a few deep learning-based approaches [4], [8] have been proposed for JIT defect prediction. Zeng et al. [54] revisited the deep learning-based JIT defect prediction models and proposed a simplistic model LApredict.

Different from the existing work (i.e., commit-level or file-level JIT defect prediction), our model COMPDEFECT focus on function-level single-statement JIT defect prediction, which is a more fine-grained defect identification task. Besides, previous work only gives two coarse-grained outputs: defect-inducing or clean. COMPDEFECT can categorize the type of defect-inducing functions.

B. Defect Repair

Defect repair [15] is also an active research topic and many machine learning-based approaches are proposed to achieve great progress [33], [34], [55]–[59]. However, the current state of automated defect repair is limited to simple small fixes, mostly one-line patches [17], [18]. Gupta et al. [60] proposed a defect repair tool named DeepFix for fixing compiler errors. Ahmed et al. [61] proposed a better approach, TRACER, for compiler errors. Martin et al. [62] proposed DeepRepair to leverage the learned code similarities to select repair ingredients from code fragments that are similar to the buggy code. Wang et al. [63] proposed a transformer-based deep learning method named TEA for accurate and effective defect repair. Chen et al. [64] proposed a unified and graph-based view of the program learning approach named PLUR. Lutellier et al. [32] proposed CoCoNut, which combines convolutional neural networks and a new context-aware neural machine translation architecture.

Considering the complexity of defect repair, similar to previous work, COMPDEFECT also focuses on the one-line code fix scenario. However, different from the existing work, the input of COMPDEFECT is the source code of the changed function in a commit and it aims to provide a foundation for connecting defect identification and defect repair. COMPDEFECT can categorize the type of defect and can fix it at check-in time.

VIII. CONCLUSION

We propose a unified defect prediction and repair framework COMPDEFECT, which can identify whether a function changed in a commit is defect-prone, categorize the type of defect, and repair such a defect automatically. Generally, the first two tasks in COMPDEFECT are treated as a multiclass classification task, while the last one is treated as a sequence generation task. The whole input of COMPDEFECT consists of three versions (clean, buggy, and fixed) of a specific function. Moreover, we build a new function-level dataset (**Function-SStuBs4J**) on the basis of ManySStuBs4J to evaluate the performance of COMPDEFECT. By comparing with state-of-the-art baselines in various settings, COMPDEFECT can achieve superior performance on classification and defect repair.

ACKNOWLEDGEMENTS

This research is supported by the National Natural Science Foundation of China (No. 62202419), the Fundamental Research Funds for the Central Universities (No. 226-2022-00064), the Ningbo Natural Science Foundation (No. 2022J184), and the State Street Zhejiang University Technology Center.

References

[1] Y. Kamei, E. Shihab, B. Adams, A. E. Hassan, A. Mockus, A. Sinha, and N. Ubayashi, "A large-scale empirical study of just-in-time quality assurance," *IEEE Transactions on Software Engineering*, vol. 39, no. 6, pp. 757–773, 2013.

[2] J. Liu, Y. Zhou, Y. Yang, H. Lu, and B. Xu, "Code churn: A neglected metric in effort-aware just-in-time defect prediction," in *Proceedings of the 11th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*. IEEE Press, 2017, pp. 11–19.

[3] C. Pornprasit and C. Tantithamthavorn, "Jitline: A simpler, better, faster, finer-grained just-in-time defect prediction," in *Proceedings of the International Conference on Mining Software Repositories (MSR)*, 2021, p. To Appear.

[4] T. Hoang, H. K. Dam, Y. Kamei, D. Lo, and N. Ubayashi, "Deepjit: an end-to-end deep learning framework for just-in-time defect prediction," in *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*. IEEE, 2019, pp. 34–45.

[5] Q. Huang, E. Shihab, X. Xia, D. Lo, and S. Li, "Identifying self-admitted technical debt in open source projects using text mining," *Empirical Software Engineering*, vol. 23, no. 1, pp. 418–451, 2018.

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

[7] X. Yang, D. Lo, X. Xia, and J. Sun, "Tlel: A two-layer ensemble learning approach for just-in-time defect prediction," *Information and Software Technology*, vol. 87, pp. 206–220, 2017.

[8] T. Hoang, H. J. Kang, D. Lo, and J. Lawall, "Cc2vec: Distributed representations of code changes," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, 2020, pp. 518–529.

[9] "Git," 2023. [Online]. Available: https://git-scm.com/

[10] S. Cao, X. Sun, L. Bo, R. Wu, B. Li, and C. Tao, "Mvd: Memory-related vulnerability detection based on flow-sensitive graph neural networks," pp. 1456–1468, 2022.

[11] D. Guo, S. Ren, S. Lu, Z. Feng, D. Tang, S. Liu, L. Zhou, N. Duan, A. Svyatkovskiy, S. Fu, M. Tufano, S. K. Deng, C. B. Clement, D. Drain, N. Sundaresan, J. Yin, D. Jiang, and M. Zhou, "Graphcodebert: Pre-training code representations with data flow," 2021.

[12] Q. Huang, X. Xia, and D. Lo, "Supervised vs unsupervised models: A holistic look at effort-aware just-in-time defect prediction," in *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2017, pp. 159–170.

[13] T. Jiang, L. Tan, and S. Kim, "Personalized defect prediction," in *Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering*. IEEE Press, 2013, pp. 279–289.

[14] Z. Chen, S. J. Kommrusch, M. Tufano, L.-N. Pouchet, D. Poshyvanyk, and M. Monperrus, "Sequencer: Sequence-to-sequence learning for end-to-end program repair," *IEEE Transactions on Software Engineering*, 2019.

[15] L. Gazzola, D. Micucci, and L. Mariani, "Automatic software repair: A survey," *IEEE Transactions on Software Engineering*, vol. 45, no. 1, pp. 34–67, 2017.

[16] R.-M. Karampatsis and C. Sutton, "How often do single-statement bugs occur? the manysstubs4j dataset," in *Proceedings of the 17th International Conference on Mining Software Repositories*, 2020, pp. 573–577.

[17] M. Wen, J. Chen, R. Wu, D. Hao, and S.-C. Cheung, "Context-aware patch generation for better automated program repair," in *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*. IEEE, 2018, pp. 1–11.

[18] R. K. Saha, Y. Lyu, H. Yoshida, and M. R. Prasad, "Elixir: Effective object-oriented program repair," in *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2017, pp. 648–659.

[19] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," *arXiv preprint arXiv:1810.04805*, 2018.

[20] "Activiti example," 2023. [Online]. Available: https://github.com/Activiti/Activiti/commit/c015d11303339f50254a10be7335fd33546911ab

[21] M. Crawshaw, "Multi-task learning with deep neural networks: A survey," *arXiv preprint arXiv:2009.09796*, 2020.

[22] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," in *Advances in neural information processing systems*, 2017, pp. 5998–6008.

[23] tree sitter, 2023. [Online]. Available: https://tree-sitter.github.io/tree-sitter/

[24] M. Jin, S. Shahriar, M. Tufano, X. Shi, S. Lu, N. Sundaresan, and A. Svyatkovskiy, "Inferfix: End-to-end program repair with llms," *arXiv preprint arXiv:2303.07263*, 2023.

[25] B. Lin, S. Wang, M. Wen, and X. Mao, "Context-aware code change embedding for better patch correctness assessment," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 31, no. 3, pp. 1–29, 2022.

[26] D. Spadini, M. Aniche, and A. Bacchelli, "PyDriller: Python framework for mining software repositories," in *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering - ESEC/FSE 2018*. New York, New York, USA: ACM Press, 2018, pp. 908–911. [Online]. Available: http://dl.acm.org/citation.cfm?doid=3236024.3264598

[27] W. Zhong, H. Ge, H. Ai, C. Li, K. Liu, J. Ge, and B. Luo, "Standup4npr: Standardizing setup for empirically comparing neural program repair systems," in *37th IEEE/ACM International Conference on Automated Software Engineering*, 2022, pp. 1–13.

[28] Y. Liu, M. Ott, N. Goyal, J. Du, M. Joshi, D. Chen, O. Levy, M. Lewis, L. Zettlemoyer, and V. Stoyanov, "Roberta: A robustly optimized bert pretraining approach," *arXiv preprint arXiv:1907.11692*, 2019.

[29] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang, and M. Zhou, "CodeBERT: A pre-trained model for programming and natural languages," in *Findings of the Association for Computational Linguistics: EMNLP 2020*. Online: Association for Computational Linguistics, Nov. 2020, pp. 1536–1547.

[30] Y. Kim, "Convolutional neural networks for sentence classification," in *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing*. ACL, 2014, pp. 1746–1751.

[31] A. Joulin, E. Grave, P. Bojanowski, M. Douze, H. Jégou, and T. Mikolov, "Fasttext.zip: Compressing text classification models," *CoRR*, vol. abs/1612.03651, 2016. [Online]. Available: http://arxiv.org/abs/1612.03651

[32] T. Lutellier, H. V. Pham, L. Pang, Y. Li, M. Wei, and L. Tan, "Coconut: combining context-aware neural translation models using ensemble for program repair," in *Proceedings of the 29th ACM SIGSOFT international symposium on software testing and analysis*, 2020, pp. 101–114.

[33] E. Mashhadi and H. Hemmati, "Applying codebert for automated program repair of java simple bugs," in *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*. IEEE, 2021, pp. 505–509.

[34] M. Tufano, C. Watson, G. Bavota, M. D. Penta, M. White, and D. Poshyvanyk, "An empirical study on learning bug-fixing patches in the wild via neural machine translation," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 28, no. 4, pp. 1–29, 2019.

[35] S. Chakraborty, Y. Ding, M. Allamanis, and B. Ray, "Codit: Code editing with tree-based neural models," *IEEE Transactions on Software Engineering*, 2020.

[36] Q. Zhu, Z. Sun, Y.-a. Xiao, W. Zhang, K. Yuan, Y. Xiong, and L. Zhang, "A syntax-guided edit decoder for neural program repair," in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2021, pp. 341–353.

[37] Y. Ding, B. Ray, P. Devanbu, and V. J. Hellendoorn, "Patching as translation: the data and the metaphor," in *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2020, pp. 275–286.

[38] X. Chen, Y. Zhao, Q. Wang, and Z. Yuan, "Multi: Multi-objective effort-aware just-in-time software defect prediction," *Information and Software Technology*, vol. 93, pp. 1–13, 2018.

[39] G. Zhipeng, X. Xin, L. David, G. John, and Z. Thomas, "Automating the removal of obsolete todo comments," in *Proceedings of the 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2021, pp. 1–1.

[40] S. Pan, L. Bao, X. Ren, X. Xia, D. Lo, and S. Li, "Automating developer chat mining," in *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2021, pp. 854–866.

[41] D. Arya, W. Wang, J. L. Guo, and J. Cheng, "Analysis and detection of information types of open source software issue discussions," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 2019, pp. 454–464.

[42] A. Wood, P. Rodeghero, A. Armaly, and C. McMillan, "Detecting speech act types in developer question/answer conversations during bug repair," in *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2018, pp. 491–502.

[43] Q. Huang, X. Xia, D. Lo, and G. C. Murphy, "Automating intention mining," *IEEE Transactions on Software Engineering*, vol. 46, no. 10, pp. 1098–1119, 2018.

[44] S. Mechtaev, J. Yi, and A. Roychoudhury, "Angelix: Scalable multiline program patch synthesis via symbolic analysis," in *Proceedings of the 38th international conference on software engineering*, 2016, pp. 691–701.

[45] A. Mockus and D. M. Weiss, "Predicting risk of software changes," *Bell Labs Technical Journal*, vol. 5, no. 2, pp. 169–180, 2000.

[46] S. Young, T. Abdou, and A. Bener, "A replication study: just-in-time defect prediction with ensemble learning," in *Proceedings of the 6th International Workshop on Realizing Artificial Intelligence Synergies in Software Engineering*, 2018, pp. 42–47.

[47] G. G. Cabral, L. L. Minku, E. Shihab, and S. Mujahid, "Class imbalance evolution and verification latency in just-in-time software defect prediction," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 2019, pp. 666–676.

[48] M. Yan, X. Xia, Y. Fan, A. E. Hassan, D. Lo, and S. Li, "Just-in-time defect identification and localization: A two-phase framework," *IEEE Transactions on Software Engineering*, 2020.

[49] C. Ni, W. Wang, K. Yang, X. Xia, K. Liu, and D. Lo, " The Best of Both Worlds: Integrating Semantic Features with Expert Features for Defect Prediction and Localization," in *Proceedings of the 2022 30th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, 2022.

[50] C. Ni, X. Xia, D. Lo, X. Yang, and A. E. Hassan, "Just-in-time defect prediction on javascript projects: A replication study," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 31, no. 4, pp. 1–38, 2022.

[51] Z. Wan, X. Xia, A. E. Hassan, D. Lo, J. Yin, and X. Yang, "Perceptions, expectations, and challenges in defect prediction," *IEEE Transactions on Software Engineering*, 2018.

[52] S. Tabassum, L. L. Minku, D. Feng, G. G. Cabral, and L. Song, "An investigation of cross-project learning in online just-in-time software defect prediction," in *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*. IEEE, 2020, pp. 554–565.

[53] X. Yang, D. Lo, X. Xia, Y. Zhang, and J. Sun, "Deep learning for just-in-time defect prediction," in *2015 IEEE International Conference on Software Quality, Reliability and Security*. IEEE, 2015, pp. 17–26.

[54] Z. Zeng, Y. Zhang, H. Zhang, and L. Zhang, "Deep just-in-time defect prediction: how far are we?" in *ISSTA '21: 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ACM, 2021, pp. 1–1.

[55] H. Hata, E. Shihab, and G. Neubig, "Learning to generate corrective patches using neural machine translation," *arXiv preprint arXiv:1812.07170*, 2018.

[56] N. Jiang, T. Lutellier, and L. Tan, "Cure: Code-aware neural machine translation for automatic program repair," in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 2021, pp. 1161–1173.

[57] S. Chakraborty and B. Ray, "On multi-modal learning of editing source code," in *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2021, pp. 443–455.

[58] N. Bui, Y. Wang, and S. C. H. Hoi, "Detect-localize-repair: A unified framework for learning to debug with codet5," in *Findings of the Association for Computational Linguistics: EMNLP 2022, Abu Dhabi, United Arab Emirates, December 7-11, 2022*, Y. Goldberg, Z. Kozareva, and Y. Zhang, Eds. Association for Computational Linguistics, 2022, pp. 812–823. [Online]. Available: https://aclanthology.org/2022.findings-emnlp.57

[59] M. Allamanis, H. Jackson-Flux, and M. Brockschmidt, "Self-supervised bug detection and repair," *Advances in Neural Information Processing Systems*, vol. 34, pp. 27 865–27 876, 2021.

[60] R. Gupta, S. Pal, A. Kanade, and S. Shevade, "Deepfix: Fixing common c language errors by deep learning," in *Thirty-First AAAI Conference on Artificial Intelligence*, 2017.

[61] U. Z. Ahmed, P. Kumar, A. Karkare, P. Kar, and S. Gulwani, "Compilation error repair: for the student programs, from the student programs," in *Proceedings of the 40th International Conference on Software Engineering: Software Engineering Education and Training*, 2018, pp. 78–87.

[62] M. White, M. Tufano, M. Martinez, M. Monperrus, and D. Poshyvanyk, "Sorting and transforming program repair ingredients via deep learning code similarities," in *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2019, pp. 479–490.

[63] W. Wang, C. Wu, L. Cheng, and Y. Zhang, "Tea: Program repair using neural network based on program information attention matrix," *arXiv preprint arXiv:2107.08262*, 2021.

[64] Z. Chen, V. J. Hellendoorn, P. Lamblin, P. Maniatis, P.-A. Manzagol, D. Tarlow, and S. Moitra, "Plur: A unifying, graph-based view of program learning, understanding, and repair," *Advances in Neural Information Processing Systems*, vol. 34, pp. 23 089–23 101, 2021.